

UNIVERSITETET I OSLO
Institutt for informatikk

Kompendium nr. 51
til IN210

Dino Karabeg og
Rune Djurhuus

22. august 1999



Innhold

1 Forord	5
2 Polyscopic scientific teaching and writing	7
2.1 Motivation	7
2.2 Anecdote One	7
2.3 Analysis of anecdote One	8
2.4 Anecdote Two	8
2.5 Analysis of anecdote Two	8
2.6 Conclusion	8
3 Forelesningslysark og -notater	11
3.1 Lecture 1	12
3.2 Lecture 2	34
3.3 Lecture 3	56
3.4 Lecture 4	80
3.5 Lecture 5	106
3.6 Lecture 6	132
3.7 Lecture 7	160
3.8 Lecture 8	184
3.9 Lecture 9	210
3.10 Lecture 10	236
3.11 Lecture 11	274
3.12 Lecture 12	290
3.13 Lecture 13	318
4 Oppgaver, ledetråder og løsninger	341
4.1 Problems	341
4.2 Hints	350
4.3 Solutions	356
5 Kommentarer til læreboka G&J	377
5.1 Innledning	377
5.2 Kommentarer til Kapittel 1	377
5.3 Kommentarer til kapittel 2	381
5.4 Kommentarer til kapittel 3	384
5.5 Kommentarer til kapittel 4	391
5.6 Kommentarer til kapittel 5	393
5.7 Kommentarer til kapittel 6	394

Kapittel 1

Forord

Dette er kompendium nr. 51 til bruk i kurset IN210 – Algoritmer og effektivitet – ved Institutt for informatikk, Universitetet i Oslo.

Kapittel 2 inneholder to anekdoter som belyser Dino Karabegs *polyskopiske* tilnærming til algoritmeteori og praksis. Kapittel 3 inneholder lysark (foiler) og kommentarer, basert på forelesningene gitt av Dino Karabeg høsten 1998. Kapittel 4 inneholder oppgaver, ledetråder og løsningsforslag. Siste kappitel er noen kommentarerer til læreboka, opprinnelig skrevet til forelesningene våren 1990.

Kompendiet er redigert av Rune Djurhuus, men mesteparten av stoffet kommer fra Dino Karabeg, med unntak av oppgaver med norsk tekst i kapittel 4 (Stein Krogdahl, Ellen Munthe-Kaas) og læreboknotatene i kapittel 5 (Stein Krogdahl).

Kompendiet finnes tilgjengelig i pdf-format på “nettet”. Se nærmere informasjon på <http://www.ifi.uio.no/~in210/> området.

Kapittel 2

Polyscopic scientific teaching and writing

2.1 Motivation

We look at the following questions: What sort of problems in scientific teaching and writing arise from the habitual flat-and-linear approach to information representation? Can those problems be resolved by using polyscopic modeling?

By studying those questions we want to

- point to the advantages of the polyscopic modeling approach (see <http://www.ifi.uio.no/~id/>) to teaching science and scientific writing over the common approaches.
- motivate our work on designing the Algorithm Theory class (IN210) and the related "Lectures in Algorithm Theory" compendium and book according to the guidelines set by the polyscopic modeling methodology.

We base this inquiry on a pair of anecdotes.

2.2 Anecdote One

"We don't want theory for the sake of theory."

I heard those words from a chairman of an American computer science department. I was interviewing for a position and those words were his greeting as I arrived from the airport. I was a novice theoretician. I felt like a cowboy-movie character riding into a city and being received by the sheriff with a gun in his hand and a "*We don't like strangers!*"

Anti-theory attitudes were not uncommon in both universities and businesses. When I was beginning with my doctorate people warned me against studying theory. I assured them that I had no intention to become a theoretician. But as it turned out I had no other choice. I found in theory the natural way to study the practical questions *scientifically*. I saw in theory a *model* of practice, not "theory for the sake of theory". Theory was for me no more and no less than organized knowledge about practice. I considered the algorithm theory the natural way to understand computing. Why, then, was theory so little appreciated among the non-theoreticians?

2.3 Analysis of anecdote One

If one opens an algorithm theory textbook one will easily understand why the good chairman acted as he did. There is no mention of real-world issues in a typical algorithm theory textbook. There are only definitions, lemmas and theorems. The kind of stuff that may interest a theoretician, but hardly anyone else.

One might conclude that we, the theoreticians, are to blame. Is it not our duty to give our field and our work a proper representation? Is it not up to us to motivate the theory so that any student or chairman can understand its meaning and relevance? Why don't we do that?

In a typical flat, linear, *mono-scopic* textbook (and in the corresponding style of lecturing) one needs to choose a single way of writing, a single point of view, a single level of detail. In a theory textbook, what could that single choice possibly be but - the theory itself?! Those definitions and those theorems really *have to be there!* But an unfortunate consequence of that natural choice is that the modeling aspect, the meaning and the relevance of the theory, the way the theory organizes our ideas about computers and computation, the way it gives us a systematic way of thinking about our field - the issues that are probably far more interesting and more relevant for a typical student than the theorems themselves - end up being left out.

2.4 Anecdote Two

"I don't read research papers. I call up my friends by phone and they tell them to me."

This I heard from a mathematician who was one of the leading people in his field. Similar attitudes have been expressed surprisingly often.

I mean, similar attitudes were often heard *among the researchers*. *Surprisingly* often because - if a researcher wouldn't read the research papers in his field, who in the world would?

2.5 Analysis of anecdote Two

Behind a typical result in a theoretical field there is usually an idea, an insight, a "trick" which makes the result possible. Such an idea can be told in an informal phone conversation in just a few minutes. But when one writes a paper about the result, one aims to obtain the sharpest possible version of it. One writes in formal terms. The simple, elegant ideas become burried in formalism and difficult to recover.

Why can't the researchers simply give an informal description of the key insights in the research article, exactly as they would in a phone conversation with a colleague friend? If they did, even the people who are not their friends would be able to understand the article without unnecessary trouble. The obvious reason is again the tradition of writing in a single style, of looking from a single view point.

2.6 Conclusion

By writing a polyscopic algorithm theory textbook we make a *model* for polyscopic scientific writing and teaching. Several styles of writing and several angles of looking are represented. We give, of course, the most relevant definitions

and theorems. But we also explain their practical implications, the *modeling aspect* of the theory. We give a high-level view of the theory which shows how the theory is structured. We extract the key techniques which make the main results possible and we explain them by using very simple examples. The algorithm theory becomes transparent, as if seen from different sides, projected on different planes, each projection showing a single aspect, a simple image.

Kapittel 3

Forelesningslysark og -notater

Dette kapitlet inneholder lysarkene (foilene) fra forelesningene i IN210 slik de ser ut for øyeblikket (august 1999). I tillegg har vi tatt med utfyllende kommentarer til hvert lysark basert på forelesningene som Dino Karabeg holdt høsten 1998. Vi har plassert kommentarene til hvert enkelt lysark direkte etter lysarket, slik at man lett kan følge med på lysarket mens man leser kommentarene.

Kommentarene er et redigert sammendrag av forelesningene. Det betyr at den fortellende, ofte litt ordrike, forelesningsstilen skinner igjennom. Materialet ville helt sikkert hatt godt av å redigeres enda mer, men fordi vi kun har hatt polynomisk tid til rådighet, må det vente til neste versjon av kompendiet.

Ideen er at du som tar kurset kan velge om du vil lese om forelesningen på forhånd og/eller bruke kommentarene som forelesningsnotater. Dermed vil du forhåpentligvis få større utbytte av forelesningene.

En annen årsak til at vi her presenterer forelesningene i skriftlig form er at pensum i kurset bare delvis dekkes av hver av de to lærebøkene. Garey og Johnsons "Computers and Intractability" er teoretisk meget sterk, men gir ikke alltid den store oversiktsforståelsen. David Harel's "Algorithmics" er lettlest og meget praktisk anvendt, men den har ikke den nødvendige teoretiske dybden som vi krever på et videregående universitetsemne.

Dino Karabeg legger stor vekt på å presentere informasjon fra flere synsvinkler som til sammen gir en helhetlig forståelse. Vi håper at denne måten å undervise på kan fungere som en bro mellom den praktisk-orienterte verdenen og teoriverdenen.

3.1 Lecture 1

IN210 – lecture 1

Administrative information

- **Teacher:** Dino Karabeg (Email: dino@ifi, Room: 3344, Phone: 22 85 27 02)
 - **Teaching material:**
 - Garey & Johnson: Computers and Intractability
 - D. Harel: Algorithmics
 - Compendium no. 51
 - **Information:**
<http://www.ifi.uio.no/~in210/>
 - **Problem sessions** (gruppeøvelser):
Prepare in advance!
 - **Midterm** (oblig): An old exam
 - **Final** (eksamen): December 2rd 1999
- NB!
- Theory \leftrightarrow proof
- Theory $\not\leftrightarrow$ opinion
- Answer = (sketch of a) proof

G&J:

1

Algo:

1

Autumn 1999

2 of 12

Welcome to IN210 – Algorithms and Efficiency. There are three books to read: The textbook in the earlier years was Garey and Johnson: Computers and Intractability. The problem with G&J is that it is not really a textbook, it is a professional book on NP-completeness and it covers only one part of the course. So there will be parts which you don't have in G&J, quite a few of them, and of course my approach is completely different from what you have in G&J.

So we will supplement G&J with David Harel's Algorithmics. This book supplements what you don't have in G&J, which are the general insights, the connection with the real world. It is a very interesting book to read. It talks about quite a few things, some of them are part of the course, some of them are not. I recommend that you read this book.

We also have the new version of the compendium which is our first attempt to combine the best from the two textbooks, namely theory and practice, into a new kind of book written in a modular (polyscopic) style. There you have foils and notes based on last year's series of lectures. You also have homework problems with hints and solutions, together with some old exams.

At <http://www.ifi.uio.no/in210/> you will find among other things announcements, a lecture plan, and the description of the "pensum" (what goes into the course and what is not part of it). You should read the announcement file ("Beskjeder") each week.

We will have problem sessions ("gruppeøvelser"), one midterm ("obliga-

torisk oppgave") and the exam. I would emphasize and recommend that you prepare in advance for the problem sessions. You should read the problems and try to solve them on your own. And even if you cannot, this would be extremely useful for you, because when you come to the group sessions, you will listen in a completely different way if you have tried to solve the problems on your own, then if you did not. You can imagine that if you try to solve the problem, then there is a little question mark that forms in your brain, it's like, almost like, a little hole that needs to be filled in. And then when information comes, this hole is ready, so you can receive the information.

A lot of researchers, really good professional people, when they enter a new area, they don't just start reading what other people have done. They just read some papers until they have understood what the problems are in the area. Then they leave the papers and try to solve the problems on their own. Typically, they cannot because they have not advanced yet to that level. But in this way they prepare themselves for actually understanding what the whole thing is about. I recommend strongly that you do something similar.

The midterm is going to be around week 9 of the course, and it will be an old exam – it will be a take-home exam. So you will take this exam home, and it will give you an idea of what the real exam will look like.

There is a major point that regards all of these exams and problem sessions, namely that the theory, which is what we are dealing with in this class, is about proofs not opinions. Everything that one states in theory, one proves. So that means that whenever I am asking for an answer, I am really asking for a sketch of a proof from you, not your opinion, not just the answer, not just 'Yes' or 'No'. Sometimes I am asking for a sketch of a proof, sometimes I am asking for the complete proof, but in either way when I read your answer, I should form an impression that you would know how to prove your answer. That is extremely important. So when you study, try to understand how you would prove things, and when you answer me questions, show me that you *can* prove things.

IN210 – lecture 1**Lecture 1 overview**

- Our approach
- The subject matter - what this is all about
- Historical introduction
- Problems and their models

Autumn 1999

3 of 12

Here is what today's lecture will be about:

I will first tell you a little bit about the approach to algorithm theory which we are going to follow. This approach differs, I would say quite a bit, from what is usual, and this is one of the reasons why we are making this compendium, and eventually turn it into a book.

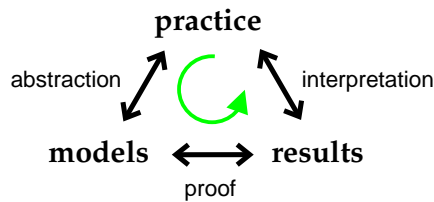
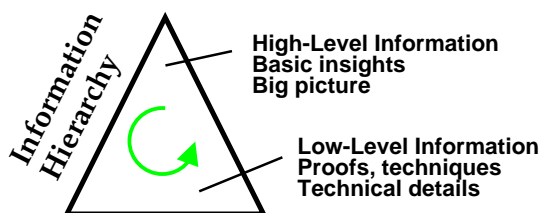
Then I will tell you what the course will cover, not in detail but kind of give you a broad idea what this whole thing is all about that we will be talking about in the class.

Next I will give a historical introduction into this subject matter. I will tell you how exactly this subject developed historically, and by doing so I will tell you what the interesting issues really are, the main interesting issues. So then we will have both: We will have an idea of how this subject developed and also what the main problems are and who are the main people who dealt with those problems – very, very briefly.

And in the end we will deal with a little bit of some technical issues, and we will start looking into the theory.

(This is a blank page)

IN210 – lecture 1

Our approach**Modeling****Perspective**

Lectures → Mainly high-level understanding

Group sessions → Practice skills: proofs, problems

Studying strategy: Don't memorise pensum – try to understand the whole!

Autumn 1999

4 of 12

Two words characterize our present approach to algorithm theory: The word 'modeling' and the word 'perspective'. It is quite usual that algorithm theory – and any kind of theory – is just theory. 'Just theory' being definitions, theorems, lemmas – technical things. We will however emphasize the practical side of the issue, or the actual reality of computing and its connection with the theory. So we will consider theory as a model of what happens in practice.

To get from the practice to theory – to models – we use abstraction, or formalization. So by using abstraction looking at theory, we ignore what is irrelevant, what maybe varies from one situation to another. As if looking through our eyebrows, we try to see the broad features – what is essential. And based on what is essential and broad, we make definitions, or formal models, and in that way we represent what we see in practice by formal, or mathematical, or defined things – formal models. So it is like we are moving from a real world to an abstract world.

And then once we are in the abstract world we have the benefit of having things well defined, having methodologies well defined. Then we live there for a while, we make theorems, gain insights, use formal proofs. And then we step back into practice and interpret those results and insights. We try to see what they really mean – practically.

So this whole cycle from practice to modeling back to practice, is one of the two most important things for us, for our approach. And I recommend to you that whenever we are dealing with abstract things, models – whenever we are in the abstract world, please see practice through it. We are not just interested in

theory for the sake of theory; we want to understand the reality through it. And vice versa: When we are talking about the real world issues, then we want to see through them, into theory, and see how exactly they are reflected in theory. So these two – practice and theory – they are inseparable, they reflect one another.

The word perspective means a kind of a compact, simple, transparent picture of the whole. Ultimately we would like to build a perspective on our whole subject matter – perspective on algorithmics, on this thing called the algorithm, or a perspective on computation. So we want to understand it as a whole. And I will be using this ideogram of the triangle. You can imagine that it represents a little pyramid, and this pyramid represents a hierarchy of information.

On the top of this pyramid we find high level information. These are the basic questions, the basic insights, and we want to know first of all, and above all, what those are. What are really the main issues? And there on the top level we want to form some kind of big picture of the whole thing. What it is really that we are talking about.

In order to answer these high level questions, in order to form the big picture, we will have to descend down the information hierarchy to the technical level, to the lower level, where we find definitions, proofs, techniques – all sort of things. So, on that level we will be spending quite a bit of time, and also on that level we practice something which is no less important than what happens up there for our goal: We practice the technical skills.

So two things are really expected from you. One: high level, global understanding of the issue. Two: technical proficiency. Because understanding, being able to ask questions, is not enough for a theory education: You need to be able to do theory, which means do proofs, create models, understand them, understand the results. So in the group sessions you will be dealing quite a bit with this low level. I will be dealing mainly with the high level in the class, but still quite a bit with the low level. Now it is your task to put this all together into a perspective, into a consistent whole.

I am repeating the modeling story really. What I am talking about now is just technically how this modeling will happen.

About the studying strategy for you: I recommend that you don't really think of the pensum, or the contents of the course, too much. Usually one thinks: "OK, so from this page to that page I have to learn, and then those pages and those pages, and then I have to memorize all these things, be able to write them down on an exam, and then maybe later I just purge, I erase them from my memory, and I take in something else." That's not a very good way of studying, not very useful.

The alternative, that I do recommend, is that you begin from the perspective, understand what the essential questions are, and then on the lower level understand what the essential techniques are that you can use for answering those questions, and then focus on those first. Because that is really what I am going to be asking on the exam. Not only I will be asking the essential questions, but also I will try to test whether you understand what those essential questions are. So you want to understand what is essential and what is not. And that's one way of saying that you do have the perspective, you have a good, compact understanding of the whole. So try to study by putting together a good understanding of the whole, as opposed to memorizing some kind of pensum.

IN210 – lecture 1



Subject matter

How to **solve** information-processing **problems efficiently**.



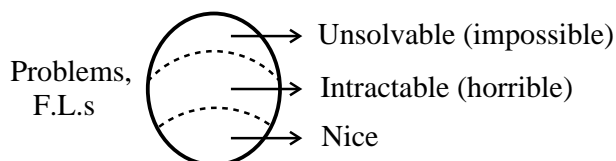
abstraction
formalisation
modeling

Problems \rightsquigarrow interesting, \rightsquigarrow formal
natural languages
problems (F.L.s)

(Ex. MATCHING, SORTING, T.S.P.)

Solutions \rightsquigarrow algorithms \rightsquigarrow Turing
machines

Efficiency \rightsquigarrow complexity \rightsquigarrow complexity
classes



Autumn 1999

5 of 12

I will use this triangle symbol in the upper-right corner of the foil to illustrate where we are in the "pyramid of information". Now we rise to the highest level of our pyramid – we speak about the most general things. We try to tell what this class is about. And we can say that the class is about one question, which is: How to solve information-processing problems efficiently.

Well, that's a general kind of question that informatics deals with. We want to be able to solve information-processing problems. Probably by using a computer, but I don't even insist on that, because a human can be a computer. A human can use the same kind of procedure that a computer uses, and solve the problem, and the issues are still more or less the same.

What is unique for this class, or for theory classes in general, is that we are using abstraction or formalization or modeling to deal with this general issue. And I am using this little snake-like arrow, \rightsquigarrow , to represent this abstraction or formalization or modeling. I have given three keywords here and we are going to apply abstraction to those keywords in order to arrive to models that are going to be the bread and butter of this course – the basic, basic tools.

So first of all we have 'problems'. There are all kinds of problems in this world, all kinds of problem that you want to solve on a computer. And putting the problems together and talking about all of them on the same time, is a challenge. We use abstraction, and we represent the real-world things like people and distances by mathematical objects, such as numbers and graphs. And by doing a little bit of sifting – seeing what is important and what is not, or separating the wheat from the chaff, as they say – we arrive at the so called interesting

or natural problems, which are studied in algorithm theory, such as the matching problem, the sorting problem, the Traveling Sales Person problem, etc. And then we use abstraction a little more, abstract from those natural problems to a concept which is called formal languages. Formal languages will represent problems for us. So they will be a formalization of the first essential term, the first essential object in reality that we are studying, which is the problems.

Then we talk about how to solve those problems. What is a solution? And we will see that the solution for us is an algorithm, it is a procedure, it is a sequence of steps that solves all the instances of a problem. And then we will formalize the algorithm further by saying that algorithms are really Turing Machines. So TMs will be something that will represent solutions for us.

And then the third thing: We don't want just any kind of solution, we want good solutions, efficient solutions. How can we tell that a solution is efficient? We will be able to talk about efficiency by comparing the complexity of our algorithms – their running time, how much resources the algorithms take – to the complexity of the problems that those algorithms are solving. If those two complexities match, then we say that we have a good, or efficient, solution. If they don't then the chances are that our algorithm is not so good. So this issue of efficiency – or complexity – will be a central issue that we will be studying, a central thing that we will be trying to evaluate when we study the algorithms as solutions.

So one of our goals will be to classify problems into complexity classes, which means into sets of problems which are of about the same difficulty and have sort of the same structure. The problems that live within a certain class will naturally be attacked by more or less the same algorithmic approach. They will be of the same or similar difficulty. So they will be a natural kind of group of problems of a certain physiognomy. So by structuring problems into these groups, we will be able to gain insights about the nature of problems, about the nature of their solutions, and organize those insights very nicely.

I will be using an egg kind of ideogram to represent all problems. So when I draw an egg, that means the set of all problems that exist in this world, or all formal languages when we talk about the problems in the formal way. And then we will see that we are dividing these problems into classes, and very broadly speaking, on the most general level, we will be talking about unsolvable problems. We will see that some problems really don't have any solution. Those will be unsolvable problems. And then among the solvable problems there are problems which we call intractable. Intuitively speaking they are hopeless problems, hopeless in the sense that we cannot really solve them exactly. And then we have nice problems on the bottom, and those nice problems they live in a class which is called P – polynomial time solvable.

So on the broadest most general level we will be interested in dividing problems into these three categories, understanding the techniques that are used. So this foil pretty much defines the most global, the most basic perspective on our class. So I hope you can have this page written/printed somewhere very nicely and maybe put it on your refrigerator, and every time you go to this class, try to see where we are exactly on this map.

IN210 – lecture 1



Historical introduction

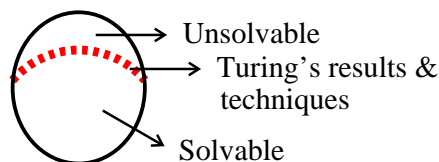
In mathematics (cooking, engineering, life)
solution = algorithm

Examples:

- $\sqrt{253} =$
- $ax^2 + bx + c = 0$
- Euclid's g.c.d. algorithm — the earliest non-trivial algorithm?

\exists algorithm? \rightarrow metamathematics

- K. Gödel (1931): nonexistent theories
- A. Turing (1936): nonexistent algorithms (article: "On computable Numbers . . . ")



Autumn 1999

6 of 12

Now we move to the algorithm. If we understand an algorithm as a procedure, as a sequence of steps which solves a certain problem or produces a certain thing, then we can say that the algorithm is in fact one of the most basic things that exists in this world. In mathematics as well as in cooking, as well as in engineering, or in life in general, the best solutions are algorithms. And what an algorithm gives you, as a kind of a solution, is that it solves not only just one instance of the problem, not only just one situation, but basically all situations, all problems of a certain kind. And it gives you the ability to deal with the problem without really thinking, without really worrying, without any strength. You just follow the recipe: 1-2-3-4-5, five steps. In five steps you have the solution. It's like in cooking: Who would ever come up with those complicated things like baking a bread? But you just open the book. It says: Step 1: Put some flour into a bowl. Step 2: Add some salt. Step 3: Add some water. Step 4: Mix well. And so on. The recipe tells you what to do, you don't have to think. It's very convenient.

In mathematics there are all kinds of recipes, like we learn very early at school how to compute the square root of a number. Or given a little polynomial how to find the roots of the polynomial, the x -values for which this evaluates to zero. We know those things.

One of the earliest non-trivial algorithms that one mentions historically is the Euclid's Greatest Common Divisor-algorithm, and actually this algorithm was proven correct by Euclid, and this happened about 400 years before Christ. So algorithms have been around for a very, very long time. And one of the big tasks of mathematics and mathematicians is to construct algorithms. And this

is what they have been doing for ages. There are lots of algorithms and if you think, you will see that actually a good part of your education is algorithms.

Mathematicians construct algorithms. Sometimes they are successful, in which case they write a paper, become famous. A lot of times they just struggle and try, and they cannot really make an algorithm for the problem. For example, for the quadratic polynomial we know how to compute the roots, but for the general polynomial we don't. And beyond a certain degree of the polynomial, we just don't know how to compute the zeros.

The natural question arises then: Given some problem, is there really an algorithm for the problem? A mathematician with this kind of knowledge is maybe able to say: "Look, this problem doesn't really have an algorithm. So I should not spend the rest of my life trying to find an algorithm. I may as well forget it and move on to some more useful and tractable problem." This ability to actually prove that there is no algorithm for a given problem, this has been given to us relatively recently.

With a certain development in mathematics, which was actually a very interesting development, people became interested in something which is called metamathematics just at the very beginning of this century. The year 1900 was actually the beginning of metamathematics. Metamathematics is about mathematics studying itself. It's like mathematics looking at itself and trying to decide, trying to prove, what mathematics can really do.

Perhaps the most famous result in metamathematics is Gödel's Incompleteness Theorem, where he has proven that certain very basic theories in mathematics, which people were trying to construct for long time without succeeding, don't really exist. This was a major, major result in 1932. And we are dealing with this result in a more advanced version of this course, IN394, which is usually thought in the spring and for 'hovedfag' students and Ph.D students. Here we leave this aside and only mention that Gödel developed a certain proof technique and used it in this proof, which allowed Alan Turing a little bit later to show that certain problems don't really have algorithms. He wrote a paper called "On Computable Numbers" in 1936 and proved that certain numbers cannot be computed, or in other words: That certain problems don't have corresponding algorithms.

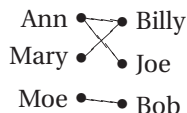
So Turing's results and techniques allowed us to make the first big separation we are interested in: To divide the world of all problems into the unsolvable ones and those that do have solutions. This is obviously totally essential for our understanding of algorithms and problems, and naturally this is going to be the first issue that we will be dealing with in this class.

So the first thing for us is uncomputability, or unsolvability. We will see those techniques that allow us to separate between unsolvable and solvable problems. You will gain insight into the unsolvable problems and unsolvability, so that when you come across a problem you can pretty much recognize right away whether this problem is unsolvable or not, because you will know the physiognomy of the unsolvable problems – what they look like. And that is of course extremely important because you will be creating lots of algorithms and programs in your life. It is very important that you are able to know that certain problems don't have algorithms, that they cannot be solved by programs. And also that you know what sort of problems those are, what they look like, so you can avoid them.

IN210 – lecture 1



- Von Neumann (ca. 1948): first computer
- Edmonds (ca. 1965): an algorithm for
MAXIMUM MATCHING



Edmonds' article rejected based on existence of trivial algorithm: Try all possibilities!

Complexity analysis of trivial algorithm (using approximation)

- $n = 100$ boys
- $n! = 100 \times 99 \times \dots \times 1 \geq 10^{90}$ possibilities
- assume $\leq 10^{12}$ possibilities tested per second
- $\leq 10^{12+4+2+3+2} \leq 10^{23}$ tested per century
- running time of trivial algorithm for $n = 100$ is $\geq 10^{90-23} = 10^{67}$ centuries!

Compare: "only" ca. 10^{13} years since Big Bang!

Autumn 1999

7 of 12

All this development of theory of algorithms happened really before the first computer was made around 1948 by John von Neumann and some other people. So computer science began before the computer was around. But once the computers came around then of course it was a lot more interesting and important to study algorithms and computer science and the theory of algorithms. Now they had this perfect mechanical slave that could do algorithms, that could do a lot of computation for them, if they could only put him into business, give him a reasonable way of doing things. So after 1948 the construction of algorithms became even more important than it was before. People were constructing algorithms left and right.

Around 1965 a researcher named Edmonds made an algorithm for a problem called maximum matching, wrote a paper and sent the paper to a journal. Let me tell you what the maximum matching problem is about: You have a set of girls and a set of boys, n of each. You have some compatibility constraints which tell you who can marry whom, or who likes whom. You assume that these relationships are mutual so if Ann likes Billy, Billy likes Ann too. So in this example Ann and Billy they are compatible. Ann and Joe, and Mary and Billy, but Mary and Joe are not compatible. They cannot be married. Edges represent compatibility in this graph, nodes represent people. And then we want to marry as many pairs of compatible people as we possibly can.

This is somewhat a non-trivial problem. Suppose Ann and Billy get married. They like each other and just get married without asking anybody. Then maybe Moe and Bob get married, and that's all right. But the problem exists with Ann and Billy because that doesn't give us the maximum possible matching. Mary and Joe have nobody to marry. So in order to get the maximum matching Ann and Billy have to divorce, and then Ann gets to marry Joe and Mary gets to marry

Billy, and then we have a maximum matching.

There are some insights here. There is this trivial kind of algorithms where you just start somewhere and then add more things to that. Edmonds' algorithm is not one of those trivial algorithms. If you have a kind of a partial solution, then you have to correct this partial solution in a clever way in order to actually arrive at the best solution eventually.

So Edmonds' algorithm was one of those non-trivial algorithms, and it took quite a bit of work. So that's why he wrote a paper about it. The paper was however rejected. The referee, the person who read the paper and evaluated it, was a mathematician. There were not many computer scientists around at that time. So this mathematician rejected the paper based on basically just one sentence. He said: "Why bother with this complicated algorithm when there is a trivial algorithm that solves the problem, namely you try all possibilities?"

What are all the possible solutions? Well, you try to marry Ann with Billy, with Joe, with Bob, and for each of those you marry Mary with one of the remaining guys, and then Moe with the guy who remains unmarried. So you sort of circle the boys and just try all possible solutions. For each possible matching you count how many people get married and keep track of the best. It's simple, it's easy, it solves the problem, why bother with complicated algorithms? Said the referee.

However Edmonds he did not give up. He actually wrote back to the referee, explaining why his algorithm was a good algorithm, and eventually he got to publish the paper. His argument was accepted. I don't know what he wrote, but we may guess that he wrote something along the following lines.

The argument is based on complexity analysis, which is the second big part of our course. We analyze the complexity of the algorithm and of the problem. I will be doing it very, very sketchy. In complexity analysis we use approximations quite a bit, and this will be almost abusing approximation. So think that we have 100 boys and 100 girls. We ask how many possible solutions there are that the naive algorithm will go through, and we see that there are $n!$ (n factorial) of them. There are 100 boys that Ann can marry, and then for each of those, 99 remaining boys that Mary can marry, and so on. So there is $100!$ possible solutions. Ignoring the 10 smallest numbers in this product, we can say that this is definitely bigger than 10^{90} – that's what you get when you multiply 10 ninety times.

We assume that each of these possibilities can be checked unbelievably fast, so in 1 picosecond we can test one. It's not possible even with the fastest computer, but we say that that's the limit. So we assume that in fact at most 10^{12} possibilities can be tested per second. We have 3600 seconds in one hour, we say that's less than 10^4 . And then we have 24 hours in one day, we say that's less than 10^2 . We have 365 days in a year, that's less than 1000. And then we have 100 years in a century.

So if we can look at 10^{12} possibilities per second, we can look at 10^{23} possibilities in one century, using the fastest computer. And since we have 10^{90} possibilities, then the trivial algorithm would take 10^{67} centuries to compute. How big is this? Well, obviously none of us will live long enough to get the output. In fact our universe is only 10^{13} years old, because that's the estimated time since the Big Bang.

This trivial algorithm is a really nice algorithm – try all possibilities, why not? Well, good luck! Nobody can wait this long, and in fact there has not been enough time since the beginning of our universe to complete even a simple run that solves an instance of the medium size 100.

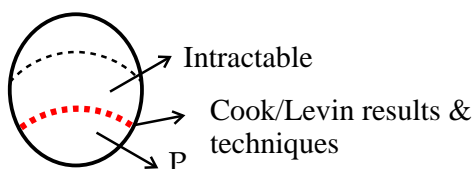
The bottom line is that there are algorithms, there are solutions which are just plain useless, because they take unbelievably much time.

IN210 – lecture 1



Edmonds: Mine algorithm is a **polynomial-time** algorithm, the trivial algorithm is **exponential-time!**

- \exists polynomial-time algorithm for a given problem?
- Cook / Levin (1972): **\mathcal{NP} -completeness**



Autumn 1999

8 of 12

So what Edmonds was saying in his response is: "Mine is not one of those useless algorithms as the one that you are proposing, my dear referee. My algorithm is a polynomial-time algorithm and your algorithm is exponential time." So these two keywords, 'polynomial time' and 'exponential time', will be used over and over again in this class to distinguish between good algorithms and bad algorithms, useful algorithms and practically useless algorithms.

Since we are theoreticians, we want to ask the obvious questions and answer them. The next obvious question that arises is: Given a problem, is there a nice algorithm, is there a polynomial-time algorithm that solves the problem? Actually Edmonds was not the first guy to speak about polynomial-time. It seems that John von Neumann was the first guy. He was a very clever guy. He really came up with all sorts of basic notions in informatics, and one of them is polynomial-time. So there was a lot of interest in constructing polynomial-time algorithms around that time, but still there was no methodology for proving that such an algorithm does not exist for a certain problem. Sometimes people were successful in constructing polynomial-time algorithms, a lot of times they were not. But when they were not, there were just nothing one could do but to give up.

This was the situation until a theory was developed in 1971-72, called the \mathcal{NP} -completeness theory, which in a certain way allows us to prove that certain problems don't have good algorithms. We can prove that certain problems most probably can not be solved efficiently by algorithms. Those problems we will call intractable. This is the second big issue which we will be dealing with quite

a bit in this class. We will look at this methodology, this bunch of techniques and results that were developed first by Cook and Levin and then some other people, that allows us to differentiate between efficient solvable problems and those intractable ones.

In this class we will be also dealing quite a bit with methods for solving in some way or other those intractable problems. Because intractable problems they do exist in this world, they have to be dealt with. If you naively just try to solve them with a real algorithm, that won't work. So if you come across one such problem, then instead of trying to solve it with an algorithm – give a precise, complete solution – you rather prove that the problem is intractable. And then you use one of those alternative techniques for dealing with the problem, without really solving it exactly, because you know that the exact solution will take an exponential time, which is just not the kind of time that we have in our lives.


IN210 – lecture 1



Problems, formal languages



All the world's information-processing problems

Ex. compute salaries, control Lunar module landing


 graphs, numbers ...

“Interesting”, “natural” problems

MATCHING
TSP
SORTING

 inp.  outp.

Functions *(sets of I/O pairs)*

 output=
YES/NO

Formal languages *(sets of 'YES-strings')*

Problem = set of strings (over an alphabet).
Each string is (the encoding of) a YES-instance.

Autumn 1999

9 of 12

So this up to here, was an introduction. We were on the top of our information hierarchy, on top of the pyramid. Now we descend a little bit towards the bottom, just to get a taste of it and see how the first of our basic notions can be formalized. We look into the ways of formalizing the notion of a problem, and we arrive at the formal notion of 'the formal language', which models the real world notion of a problem.

There are all kinds of problems which one solves with algorithms or computers, for example one computes salary, or computers control complicated machinery like the landing of the lunar module or airplanes. So there are all kinds of things there, and the question is: How can one actually put all those things into one pot, talk about all of them in a unified way? How can we study problems theoretically?

Well, we use abstraction. We say: Look, what is really important about the problem is not whether the problem talks about salaries or ages of people or heights of trees. All those things are numbers. One problem is asking for the same thing about trees as some other problem is about people. We don't care about such distinctions. We say: Numbers. So we abstract real world things and properties by representing them by mathematical, abstract, people-invented objects such as numbers. Another good object is graphs. When we gain some experience in this business of representing real world things with abstract mathematical objects, then we see that there are not so many interesting or really different problems in this world. There are quite a few, but a manageable number.

So by using abstraction, we arrive at a certain set of interesting problems.

Common examples are the maximum matching problem – the one that we have just seen – the traveling sales person's problem, the sorting problem, and so on. And we will be seeing quite a few of those problems in this class. Typically those interesting problems are formulated as input/output pairs, or more mathematically speaking, as functions. What you want to know when you define a problem is: Given an input, the instance of a problem, what should the output be, what should the solution be, what is the answer? And then those question/answer pairs, all of them together, they make up a problem. So to define a problem, you define this function in some way or the other.

We abstract further by saying: We don't really care about all sorts of outputs here. We just care about those problems which have a single bit of information as output, which have an yes/no-answer. So we represent all problems by such problems that have 'Yes' or 'No' as answers. How could we do this? Well, take sorting for example. The sorting problem is ordering a certain sequence of numbers or names or whatever.

So an input to the sorting problem is a bunch of things in an arbitrary order. The output that is desired is the same bunch of things, nicely ordered. But you can turn this problem into a yes/no kind of question by saying: "Here is the input. Is it ordered?" And the problem consists in checking whether the order is the right order. This problem is certainly not more difficult than actually producing the solution. It may be easier, but in a certain way we can say that this yes/no-problem reasonably well represents the original problem.

So just about always we are able to represent the real world, interesting problems by these yes/no kind of problems, and from yes/no-problems we arrive at a notion of formal languages very easily. There a problem becomes just a set of strings over an alphabet. Now we talk a little more about this.

IN210 – lecture 1



Def. 1 *Alphabet* = finite set of symbols

Ex. $\Sigma = \{0, 1\}$; $\Sigma = \{A, \dots, Z\}$

Coding: binary \leftrightarrow ASCII

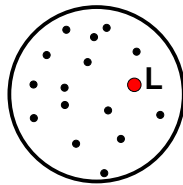
Def. 2 Σ^* = all finite strings over Σ

$\Sigma^* = \{\epsilon, 0, 1, 00, 01, \dots\}$ — in **lexicographic order**

Def. 3 A *formal language* L over Σ is a subset of Σ^*

L is the set of all “YES-instances”.

Set of all
problems



Here we are at the bottom of our pyramid, dealing completely with technical things. We show this on the foil by the green triangle in the upper-right corner. I wouldn't say that this is completely the bottom because in the theory world these are very, very basic concepts.

We start with some definitions. An alphabet is defined as finite set of symbols, where symbols are some objects. Common alphabets are characters a to z which we use for writing, and which is the alphabet that we learn first in the school. In computer science a common alphabet is $\{0, 1\}$. This is the kind of alphabet that the computers use. And then there are all kinds of other alphabets in this world. We can extend this a-z alphabet with small letters and special characters and digits from 0 to 9 and so on.

But by and large it is irrelevant which alphabet we are using, because we can represent one alphabet by another by using what is called 'coding'. You are probably familiar with what is called the ASCII-code which allows us – or which allows the computers – to represent the real world characters that exist on the keyboard, by 0's and 1's. We use a sequence of eight 0's and 1's per character, and that gives us enough characters to represent each of this real world characters. So by using coding techniques any alphabet, even if the alphabet is fairly large, can be represented by a very small alphabet such as 0 and 1. We just use a sufficient but constant number of 0's and 1's to represent each of the characters in the big alphabet. So then we can focus on a simple alphabet, say $\{0, 1\}$, and that's what we will be doing most of the time in this class, because that's what the computers are doing.

Sigma-star will represent all finite strings over the alphabet, namely all possible strings or sequences that one can construct from these characters. Here is what sigma-star is if the alphabet is 0 and 1: It's first the empty string, epsilon, and then 0, 1, 00, 01, and so on. This way of listing the strings over the $\{0, 1\}$ -alphabet is called the lexicographic order. It is the natural way to order the strings. Notice that all of them are here. You order the strings first by lengths, and then according to the rule which we apply when we look in a dictionary.

By definition, a formal language over the alphabet Sigma is a subset of Sigma-star. In other words: A formal language over an alphabet is a set of strings over that alphabet. This set can be finite or infinite. Now, how does this represent a problem? It is very simple: We take those strings that are the yes-instances of the problem – the questions whose answer is yes. We take them together into a set, and this set represents the problem. This set is a formal language.

In the sorting problem – or the formal language that corresponds to the sorting problem – we take all the sorted sequences, their representations, some kind of encoding. The sorted instances are the 'Yes'-instances. We put them all together and we say: This is the sorting problem. This is the formal language that corresponds to the sorting problem.

We can do this pretty much with any problem in the world. We can represent all problems with formal languages. And formal languages, what are they? They are just sets of strings. And this allows us to do this piece of magic: To put all problems in this world under one hat, into this one set which is the set of all sets of strings. And each problem is a point in this big set. Each problem is a formal language. So we have no gathered all the problems into one house, put them together. And now are going to create rooms in this house, organize the problems together. That's what this class will, to a large extent, be about.

So when we organize the problems we can also talk about how to solve them, because we don't want to solve the problems individually. There are too many of them. But talking about how to solve a certain kind of group of problems, that makes a lot more sense.

IN210 – lecture 1



A sample result

- We want to show that there are more problems than solutions!
- Formalisation:
 problems \rightsquigarrow sets of strings
 solutions \rightsquigarrow strings
- Proof methodology:
 - proof by contradiction
 - diagonalization
 - (non-constructive) existence proof by counting
- Assume all problems have solutions.
- Then all problems fit into matrix.

Autumn 1999

11 of 12

I want to prove our first result, and this result is something that you know from experience, intuitively, but never quite knew how to go about proving it. The result is that there are more problems in this world than there are solutions. Or in other words that certain problems just don't have solutions. And of course you know this, but without going into the formal world, there is no way to prove it. So you just shake your hands and feel maybe even inadequate when you are dealing with a difficult problem, but here we are going to be able to prove that some problems don't have solutions. Later on when we aren't able to solve a problem, we will feel much better about ourselves, and that would be good!

By the way, if you study the proof well enough then you will notice that in fact most problems don't have solutions. That just follows from the proof.

Our methodology will introduce, or review, some very basic techniques such as the proof by contradiction and the proof by diagonalization – which is the proof technique that both Gödel and Turing used to prove their fundamental results. Another basic notion is the non-constructive existence proof by a counting argument. I don't insist on you knowing all these words, but it is good to mention them because some of you may be interested in actually understanding what they mean.

The overall structure of my argument will be a proof by contradiction. Towards the contradiction we assume that all problems do have solutions. If all problems do have solutions then we can write all problems down and organize them into a huge kind of matrix. This is an infinite matrix, so we must be careful about saying what we can or cannot do. But in principle this can be done.

'In principle' meaning in the mathematical sense, for we cannot really write all possible strings, but in principle they can be written. And this is how:

IN210 – lecture 1



	ϵ	0	1	00	01	10	11	000	001	\dots
ϵ										
0										
1										
00	0	1	0	$\emptyset 1$	1	1	0	1	0	
01										
10	1	0	1	0	0	$\lambda 0$	1	1	1	
11	0	0	1	1	0	0	$\lambda 0$	1	0	
000										
001	1	1	1	0	1	1	0	1	$\emptyset 1$	
\vdots										

- Solutions = row labels = finite strings of characters.
- Each yellow row represents a problem (formal language).
- Flip the diagonal entries (replace 0's by 1' and 1's by 0's).
- The diagonal language $L_D = \{00, 001, \dots\}$ is not (solved by any algorithm) in matrix.
- Contradiction!

Autumn 1999

12 of 12

The matrix has strings as column labels and strings as row labels, and these strings they are ordered lexicographically. We know that we can write, in principle, all strings in a sequence because they are ordered. So here is the empty string, 0, 1, 00, and so on.

If all problems have solutions, then all problems can fit into this huge matrix. But first of all, what are solutions? I haven't been talking the solutions yet. We have said that solutions are algorithms or Turing machines, but we don't even have to formalize the notion of the solution to that degree yet, in order to get this result. All we need is a very basic fact about the solutions, and that is that a solution, a solution being a procedure, a sequence of steps, a recipe, a solution is something that we must be able to write down using some kind of text. In other words, a solution is a finite string of characters, a finite string. A solution is something that you can write on one page of paper, or maybe on 100 pages, maybe 1000 pages, but it is a finite string.

So every solution, every algorithm must be one of those strings here, because all strings are here. They are the row labels of this huge matrix. Most of the strings will probably not be algorithms, they will just be nonsense, but some strings will be algorithms encoded in a certain way. And suppose that this row is an algorithm, and this row and this row (shaded yellow). They are probably too short, you know 00 is not a great algorithm, but suppose that these things are algorithms.

And then each algorithm solves a problem. So we encode this problem as a formal language. A formal language is a set of strings, so since we have all the

strings here as column labels, we put a 1 in the row's entry if the corresponding string belongs to the language, 0 otherwise. So by doing so in each entry of the row that corresponds to a solution, we can represent the language that corresponds to the problem that the algorithm is solving.

Take algorithm '00' as an example: the empty string is not in the language, 0 is in the language, 1 is not in the language, 00 is not in the language, and so on. We do the same for each solution. So if all problems have solutions, then all problems are represented in this matrix.

Now I'm going to show you that there is in fact one problem that is not in the matrix. I'm going to construct it and this is how: I look at diagonal of this huge matrix. And this diagonal is again a sequence of 0's and 1's. We don't worry about what is in these entries where we don't have algorithms, because that's irrelevant. It could be either 1 or 0.

So, we look at the diagonal, and then we flip the entries of the diagonal: If we see a 0, we turn it into a 1. If we see a 1, we turn it into a 0. So in row '00' this 0 becomes 1, in row '10' this 1 becomes 0, etc. And then we look at the corresponding language that is defined by this sequence that I have just created. This language includes the string 00, the string 001, and some other strings too. So I put these all together and get this language called L_D , D for diagonal, the diagonal language.

And I claim that the problem that this language is representing is not solved by any algorithm. In other words that this language is different from all the rows in this big matrix. Why is this true? The diagonal language differs from every one of these row languages in at least one position, namely in the diagonal position. That is how the diagonal language was constructed.

So this act of flipping the diagonal magically changes the diagonal language so that it differs from each of the row languages in at least one place. I have constructed a language that is not in the matrix. So I have reached a contradiction, because I have assumed from the beginning that all problems have solutions, which means that all problems can be represented as rows in this matrix. Since I have reached a contradiction under this assumption, that means that my assumption was wrong, in other words that not all problems have solutions.

That's about it. Next time I will be talking about how to formalize the notion of an algorithm, and we will arrive at notion of the Turing machine. And then we will see actually how Turing used his machine and all these ideas to prove that certain concrete problems don't have solutions, to develop the whole methodology for proving that certain problems don't have solutions. We will see that this same argument that I was using here, pretty much is what Turing was using, but in a different way, to prove his result. So then it will all come together.

3.2 Lecture 2

IN210 – lecture 2



Review

How to **solve** the information-processing **problems efficiently**.

\rightsquigarrow : abstraction, formalisation

Problems \rightsquigarrow I/O pairs, \rightsquigarrow formal
functions, languages
“interesting
problems”

solutions \rightsquigarrow algorithms \rightsquigarrow Turing
machines

efficiency \rightsquigarrow resources, \rightsquigarrow complexity
upper/lower classes
bounds



high-level information



low-level information

Autumn 1999

1 of 11

Algo:
9 (p.223-238)

Last time we were talking about how to solve information-processing problems efficiently. That is our subject. That is what we are going to talk about in this class. This is a theory class, which means that we are using abstraction and formalization to study reality. And this abstraction or formalization we represented by a little curly arrow.

We applied abstraction and formalization to problems that exist in this world. First we saw that problems can be represented abstractly as input/output-pairs or functions or interesting problems, which happens basically when we replace objects that exist in the world – like people and salaries – with mathematical objects such as graphs and numbers. And then a further step in formalization took us to formal languages which allow us to represent all the world’s problems as languages, as sets. And the set of all languages we represented by an oval form, an egg. And a very convenient outcome of this development has been that we are now able to put all the problems under one roof, treat them uniformly, and create a kind of a map of problems – divide the problems into classes. This is what is going to happen as a final result of this class.

We have said that when we apply abstraction and formalization to solutions we arrive at the notion of algorithms. Now, an algorithm is still an informal notion. Basically you recognize an algorithm when you see one. An algorithm is a kind of a procedure, it’s a cook book recipe. It’s a recipe that solves all instances of a problem.

An algorithm as a notion cannot be defined in the mathematical sense. Further abstraction is needed and it leads to Turing machines. The Turing machine is a formal concept, it is something that is defined. There is a big difference between something that is defined and something that is informal as an algorithm, because when something is defined then you can use that something in mathematical proofs.

Efficiency, our major concern, leads us to notions such as resources, upper/lower bounds, and further abstraction gives us complexity classes. We can divide problems into classes of problems based on how expensive their solutions are, in terms of certain resources that are used in computation such computation time and computation space.

Based on efficiency we divide the formal languages, or problems, into classes and then by placing a certain language or a problem into a certain class, we make a statement about the complexity or difficulty of the problem. Then we know what sorts of algorithmic approaches, what sorts of solutions, are appropriate or efficient for that problem or for that class. In that way we pinpoint efficiency. We make statements about efficiency. And as I said that is going to be the major concern of this class.

In the upper-right corner of each foil we put a pyramid symbol to show where we are in the information pyramid. When the upper part of the triangle is shaded, that means we are dealing with the big picture, the basic insights, the so-called high-level information. A triangle with the lower part shaded indicates more formal, technical, low-level information.

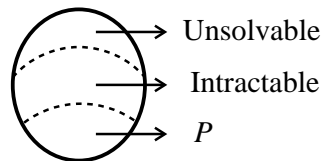
IN210 – lecture 2



Theorem 1 *There are more problems than solutions.*

Corollary 1 *There are (many) problems that cannot be solved by algorithms.*

The “egg of all problems” which we want to divide into (complexity) classes:



Today

- **Turing machines** as an algorithm model
- **Turing's theorem** which gives us a (provably) unsolvable problem.

We are going to study all sorts of classes and all sorts of algorithmic approaches that are appropriate for those classes, but to begin with, as a kind of a most basic picture, we are going to divide the problems into two big classes: The unsolvable problems that cannot be solved by algorithms, and the others that can be solved by algorithms. The solvable problems we are going to divide further into problems that are properly solvable with good algorithms (a class called \mathcal{P} or polynomial-time solvable) and others that are not solvable in polynomial time which we call intractable (\mathcal{NP} -complete, \mathcal{NP} -hard and so on).

Our major concern is going to be dividing problems into unsolvable, intractable and properly solvable, studying techniques that allows us to make this division, and gaining insights into what sorts of problems live in each "country". What kind of problems are those that are unsolvable, what sorts of problems are intractable, and what sorts of problems are solvable in polynomial time?

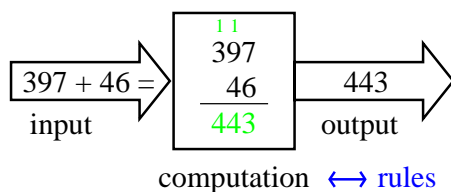
At the end of last class we have seen a theorem which says that there are more problems than solutions. And a consequence of this theorem is that there are problems that cannot be solved by algorithms. So we know that such problems exist, but we don't know how to find one yet. In fact we know a little more: We know that many, most problems in fact, are unsolvable. How do we know that? That follows from the proof of theorem 1 that we have seen last time. There are countable many algorithms and uncountable many problems. There are as many algorithms as there are natural numbers. There are as many problems as there are real numbers. And we know that between every pair of natural numbers, there are infinitely many real numbers. The relationship between algo-

rithms and problems is the same.

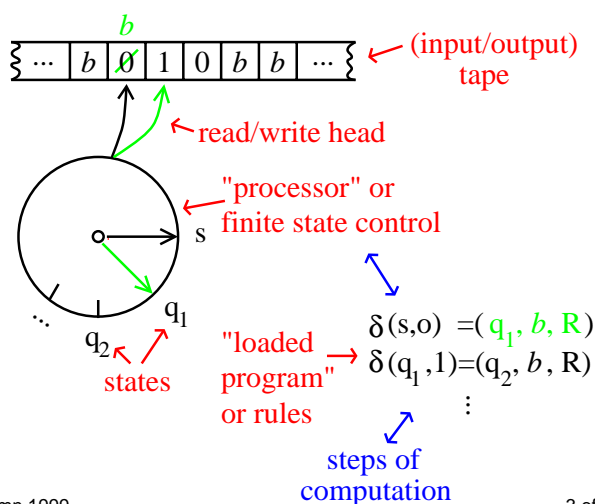
Today we go further: We define and study the Turing machine as a formalization or model of an algorithm, and then we use the Turing machine to study unsolvability. We will try to go as far as proving the Turing theorem which gives us our first provably unsolvable problem. Turing theorem will be the foundation stone for our technique for proving unsolvability. Once we have one unsolvable problem, by using a technique called reduction, we will be able to prove for many other problems that they are also unsolvable.

IN210 – lecture 2

Algorithm



Turing machine – intuitive description



Autumn 1999

3 of 11

If we reduce the notion of an algorithm a little bit from all sorts of things like cookbook recipes, and study only the mathematical algorithms, which is what Turing did, then we can see that algorithms have certain common features. There is something that comes as input, as we would call it in computer science, and that input is usually some kind of numbers, or some text generally.

On this foil we are studying a simple algorithm for adding two numbers, and what comes in is the two numbers and a little plus which tells us what to do with these numbers. We learn very early at school how to make this summation: We write these two numbers one below another and then we go through a certain procedure. Like here we would be adding 6 and 7 which gives us 13. There is a carry 1 which we add to the next column. We have 1 plus 9 which is 10 plus 4, that makes 14. Again the carry is 1. 1 and 3 is 4. So we know how to perform this computation, and what comes out is a result or output: 443.

So there are lots of algorithms in this world that mathematicians have created for us, and all have this basic structure. So when Alan Turing was constructing his machine – computers did not exist yet – he was really modeling a mathematician. So a Turing machine is a mathematician executing an algorithm, or a model of it.

I will first describe the Turing machine intuitively, as a kind of a machine. And then we will see how this intuitive description leads to a completely formal concept of a TM. So a TM consists of a tape, which we could call an input/output tape or simply tape. And this tape is infinite on both ends. Sometimes it is given as infinite on only one end. Every book has a different TM, a different definition,

and in order to be consistent with this tradition I have also invented my own TM, so this machine will probably defer from all machines in all books.

The tape is infinite on both ends and it has squares. Each square can house one character, one symbol. In the example on the foil we see that most of the tape is blank. And you can think that all the non-blank part here is this '010'. That is 3 characters and these 3 characters represent input. The reason why know that this is input, is because at the beginning of time, before the machine has started to do anything, before it has started computing, its read/write head – which is the arrow here – points to the first character of the input.

In addition to the tape and the read/write head a Turing machine has a "processor" – a finite-state control – which tells us about the state in which the machine is. There are a finite number of these states. If you think of the machine of being a digital computer, then you can think of the tape as being the computer's memory. And since memory is cheap you can always add more memory, so this is why this memory is infinite. The read/write head is something that can read the memory and this circular disk is the CPU unit – the central processing unit. You know that any digital device always can have just a finite number of states.

So it makes sense to represent a computer in this way. If you think of this as being a mathematician then this tape is really the paper that the mathematician is using, and paper is also cheap. The read/write head must be his own hands or eyes which he is using to read and write on the paper. And the circular disk is the mathematician's own memory in the head, which tends to be limited – that's why we use paper, because our memories are limited.

The heart of the algorithm and the heart of the Turing machine is a bunch of rules, a finite number of them. The rules tell how the TM does its computation. The rules are defined by this function δ which for every pair of state and tape symbol, tells us three things: 1) the next state 2) the symbol which is going to replace the currently scanned symbol, and 3) the movement of the read/write head – left or right. These rules define the steps of computation. An application of a rule is a step.

So what does a Turing machine do in one step of computation? Here I am showing in green what changes happen in one step of computation. According to this rule, this TM is going to change its state from s to q_1 , it is going to replace the currently scanned symbol – which is 0 – by a blank. It is going to erase, effectively, this 0. Finally it is going to move its read/write head one square to the right. So those are the three actions that are specified by this rule.

There are two special states among these states. One is the state s – the start state – which is the state in which the TM begins its computation. Another special state is the state h – the halting state or the halt state – and this is the state where the machine stops. So when the machine reaches h then it just stops executing – it halts.

If I give you this kind of description of a TM, you can easily perform the computation that the TM is doing simply by applying one rule at the time, one rule in each step. So each application of a rule is going to move the machine one step further – the state is going to change, the position of the head is going to change and the content of the tape is going to change. And this goes on until the machine has reached the state h at which point the computation stops, and then you are free to do something else.

IN210 – lecture 2



Turing machine – formal description

A **Turing machine (TM)** is $M = (\Sigma, \Gamma, Q, \delta)$ where

Σ , the **input alphabet** is a finite set of input symbols

Γ , the **tape alphabet** is a finite set of tape symbols which includes Σ , a special **blank symbol** $b \in \Gamma \setminus \Sigma$, and possibly other symbols

Q is a finite set of **states** which includes a **start state** s and a **halt state** h

δ , the **transition function** is

$$\delta : (Q \setminus \{h\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

NB! “Almost” every textbook has its own unique definition of a Turing machine, but they are all equivalent in a certain sense.

We are descending now to the bottom level. We are looking at something which is completely formal. And my intention here in presenting you all these details is not to throw all sorts of details on you, but mainly to show you how this kind of formalization can be done, and what its meaning is. So please don't get too much involved with details, but try to understand the big basic message, which is the spirit of formalization, what the formalization really means, what it looks like.

What we want to do here is to represent the Turing machine as a mathematical object, something which is completely formal. And we need something which is completely formal in order to be able to do formal proofs. At the point where we will be able to do formal proofs, we will be able to use the mathematical techniques and that is the beginning of theory. At that point we move from this real world where machines and people and algorithms exist, to the abstract or formal world where we can prove theorems – and that is a very big deal. We prove theorems, gain insights, and then we go back to the real world and interpret those results and organize the real world.

So how do we formalize the notion of an algorithm? How do we represent the Turing machine formally? Formally a TM is a quadruple. It is a sequence of four things called sigma (Σ), gamma (Γ), Q , and delta (δ). Sigma is the input alphabet and as every alphabet it is a finite set of symbols, called input symbols. Gamma is a superset of sigma. It contains all of sigma plus some more symbols. Gamma is called the tape alphabet and in addition to the input symbols of sigma, it contains a special blank symbol and possibly others. Q is a finite set of

states which includes a start state s and a halt state h . Delta, the transition function, is a function from all the non-halting states and tape symbols to the set of all states, tape symbols and two special symbols L and R representing left and right. L and R are intuitively the directions of the movement of the read/write head.

IN210 – lecture 2

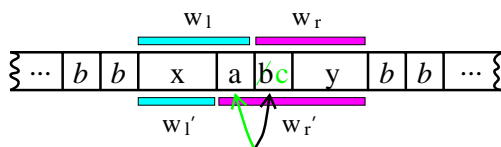


Computation – formal definition

A **configuration** of a Turing machine M is a triple $C = (q, w_l, w_r)$ where $q \in Q$ is a state and w_l and w_r are strings over the tape alphabet.

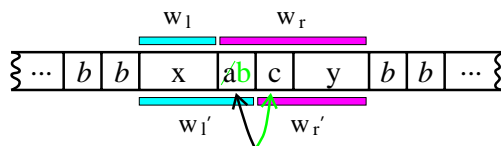
We say that a configuration (q, w_l, w_r) **yields in one step** configuration (q', w'_l, w'_r) and write $(q, w_l, w_r) \xrightarrow{M} (q', w'_l, w'_r)$ if (and only if) for some $a, b, c \in \Gamma$ and $x, y \in \Gamma^*$ either

$$\begin{array}{lll} w_l = xa & w_r = by & \text{and} \\ w'_l = x & w'_r = acy & \delta(q, b) = (q', c, L) \end{array}$$



or

$$\begin{array}{lll} w_l = x & w_r = acy & \text{and} \\ w'_l = xb & w'_r = cy & \delta(q, a) = (q', b, R) \end{array}$$



Autumn 1999

5 of 11

So this is a Turing machine formally. A TM is just these four things. Now we want to say what a TM is doing. We want to define the notion of computation formally. We do that in a few steps, the first being a definition of a configuration. Intuitively a configuration is a snapshot of the machine. It is like a photograph of the machine which tells us everything we want to know about the machine at a certain point in time. So that if we know a configuration of the TM, we don't need to know its past. We can predict, given the definition of the transition function of the TM and a configuration, its future behavior completely. So a configuration and the transition function, tells us all we want to know about the TM.

What do we want to know? It is three things: First, we want to know the content of the tape, all that is written on the tape. Second, we want to know the position of the read/write head. Third, we want to know the state of the machine. These three things together with the definition of the transition function delta, the rules, tell us everything we want to know about a TM. And if we know them we can apply the rules and simulate the behavior of the machine.

We represent this intuitive idea of a configuration formally in the following way: We say that a configuration of a TM M is a triple (q, w_l, w_r) where q is a state and w_l and w_r are strings over the tape alphabet. The intuitive meaning of this is that the machine M is in state q , w_l is the portion of the written tape to the left of the read/write head, and w_r is the written portion of the tape to the right of the read/write head. By convention we assume that the read/write head is scanning the first symbol of w_r . So these things represent the whole configuration: The

position of the read/write head, the state, and what is written on the tape, $w_l w_r$, the concatenation of the two strings.

Now we continue with formalism by defining what it means that the configuration (q, w_l, w_r) 'yields in one step' configuration (q', w'_l, w'_r) . Intuitively it will mean that configuration (q', w'_l, w'_r) follows from configuration (q, w_l, w_r) by the application of one of the rules, the delta-function. There are two situations really: The situation where the head moves to the left and the situation where the head moves to the right. The two figures on the foil represent these two situations, and this bunch of equations is just a formal way of saying it.

IN210 – lecture 2

**Note:**

w_l is the written portion of the tape to the left of the read/write head.

w_r is the written portion of the tape to the right of the read/write head, including the square the head is currently scanning.

We say that a configuration $C = (q, w_l, q_r)$ **yields** configuration $C' = (q', w'_l, q'_r)$ and write $C \stackrel{*}{\vdash}_M C'$ if (and only if) there is a sequence of configurations $C = C_1, C_2, \dots, C_n = C'$ of M such that

$$C_i \stackrel{\vdash}{\vdash}_M C_{i+1} \quad \text{for } i = 1, \dots, n - 1$$

We say that Turing machine M **computes function** f if (and only if) for all $w_1, w_2 \in \Sigma^*$

$$(s, \epsilon, w_1) \stackrel{*}{\vdash}_M (h, \epsilon, w_2) \Leftrightarrow f(w_1) = w_2$$

Autumn 1999

6 of 11

Now we are defining another relation which is the 'yields' relation. It's a relation between configurations. We say that configuration C yields configuration C' if there is a sequence of configurations C_1 to C_n of TM M such that they all yield another in one step: C_1 yields in one step C_2 , C_2 yields in one step C_3 , and so on all the way up to C_n .

We can now define the notion of computation of a function. We say that a TM M computes function f if for all w_1 and w_2 in sigma-star, configuration (s, ϵ, w_1) yields configuration (h, ϵ, w_2) if and only if $f(w_1)$ is w_2 . What is configuration (s, ϵ, w_1) ? It is a starting configuration where w_1 is written on the tape and the read/write head is scanning the first symbol of w_1 . Epsilon is just nothing, it's empty, so the tape is empty to the left of the input string.

So if this configuration yields a halting configuration where there is again nothing written to the left of the read/write head and the read/write head is scanning the first symbol of w_2 , then we say that effectively the Turing machine has turned w_1 into w_2 . It has computed the function f .

(This is a blank page)

IN210 – lecture 2



We say that Turing machine M **decides language L** if (and only if) M computes the function

$$f : \Sigma^* \rightarrow \{Y, N\} \text{ and for each } x \in L : f(x) = Y \\ \text{for each } x \notin L : f(x) = N$$

Language L is **(Turing) decidable** if (and only if) there is a Turing machine which decides it.

We say that Turing machine M **accepts language L** if M halts if and only if its input is an string in L .

Language L is **(Turing) acceptable** if (and only if) there is a Turing machine which accepts it.

The next very important notion is a Turing machine as a decision procedure. We say that a TM M decides language L if M computes the function f from sigma-star to $\{Y, N\}$ – where 'Y' and 'N' are two special symbols in the tape alphabet – such that for each x in L , $f(x)$ is Y . Meaning that if the input is in L then the machine will say 'Yes', and otherwise the machine will say 'No'. This function is a so-called total function because for every possible string it is going to say either 'Yes' or 'No' depending on whether the string is in the language. If M computes this total function then we say that Turing machine M decides language L . And we are going to say that the language L is decidable if and only if there is a TM which decides it.

When you will be reading these definitions at home you will see that they are just completely natural. They are just really bread and butter, but they allow us to formalize the notion of solution and the notion of computation in such a way that we can now just apply these rules completely blindly and formally in a given situation, and we can apply these formal objects in mathematical proofs.

We are going to use one more notion. We say that a TM M accepts language L if M halts if and only if its input is a string in L , and we say that a language L is Turing acceptable, or simply acceptable, if and only if there is TM which accepts it. What does this mean? Notice that a machine can actually *not halt*. A machine can just compute forever. It can be in an infinite loop, maybe just going off to the blank part of the tape and then moving alternately left and right forever. So the machine can end in an infinite loop and never finish. If that happens we say that the machine doesn't halt.

This level of formalism is not really going to be characteristic for this class or for theory of computation as such. We are not going to deal with such things, typically. There is a reason why I am doing this today, why I am going to this level of detail and this level of formalism. The reason has to do with our pyramid of knowledge.

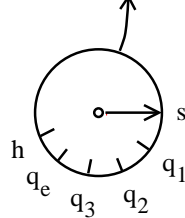
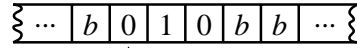
Most of the time we will be talking about TMs and all sorts of other things on a very high level. I will be informally describing a procedure, a proof, and so on. I will be spending time on top of the pyramid. But when I do that you will have to know that in fact all those informal notions can be completely formalized, completely in the sense that they become mathematical things, mathematical objects.

That is what is meant by building the pyramid. When I describe a TM informally then you know that pretty much automatically you are able to make a completely formal description of that TM. Maybe it would take 5 months and 3000 pages of text to do that, but never the less you are able to do it. You are not going to do it because it is terribly boring, it's completely stupid and in fact wouldn't add anything to our understanding. But it is important to know that it is possible to do what we are doing informally in a completely formal way. Because that gives us the foundation in theory. That gives us a solid ground to rest on. So we are now building a foundation, we are building the lower level of the pyramid, and that's what these definitions are about.

IN210 – lecture 2

**Example**

A Turing machine M which decides
 $L = \{010\}$.



$$M = (\Sigma, \Gamma, Q, \delta) \quad \Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, b, Y, N\} \quad Q = \{s, h, q_1, q_2, q_3, q_e\}$$

δ :

	0	1	b
s	(q_1, b, R)	(q_e, b, R)	$(h, N, -)$
q ₁	(q_e, b, R)	(q_2, b, R)	$(h, N, -)$
q ₂	(q_3, b, R)	(q_e, b, R)	$(h, N, -)$
q ₃	(q_e, b, R)	(q_e, b, R)	$(h, Y, -)$
q _e	(q_e, b, R)	(q_e, b, R)	$(h, N, -)$

('-' means "don't move the read/write head")

Autumn 1999

8 of 11

I will now show you an example of a TM which decides language L consisting of only one string, namely '010'. You will see further examples of TMs and computations and things in the groups. Now I just want to illustrate very briefly, in a few steps, how one goes about creating this kind of machine.

So think that there is a tape where '010' is written and the rest is blank. You know that by convention, if this is the input then the TM will be scanning the first symbol and its finite state control will be in state s . In order to define the TM we have to define these 4 things: Sigma, Gamma, Q and delta – with delta being the most important part, the heart of the matter, the rules.

So we will say that Sigma is '0' and '1', because that's how the input is written. We will say that Gamma is '0', '1', 'b' (blank), 'Y' (Yes) and 'N' (No). Possibly for a more complex computation we may need some more symbols here, but this will be good enough for this simple, simple machine. Q is going to have always state s , state h , and then possibly some more states. We will figure that out as we go along. We say we need q_1 and then possibly some more q 's.

Delta will be defined by a table where for each pair of symbol and state we are going to say what the machine does. If the machine is in state s scanning a '0' initially, then we want to say: "So far so good, everything is fine." So this state q_1 will mean "I have seen a '0' as the first symbol. I erased it and moved the read/write head to the right." So if we are in s scanning a '0', then we change state to q_1 , replace the '0' by a 'b' and moves the head to the right.

What should come next is a '1', so if our machine is in state q_1 and it is scanning a '1' then everything is fine. Then we move to state q_2 . So q_2 means that I

have seen '0' and '1' as the two first symbols. That is the meaning of state q_2 .

In the example we are now in q_2 and scanning a '0'. q_2 and 0 will mean: "Everything is fine, the input is just right so far. Go to the right and see if there is a blank there." So state q_3 means that the first three symbols were OK. If we are in q_3 and are scanning a blank, then we want to write 'Y' and halt because the input string is in language L . But in our definition of a TM we are not really allowed to stand still, we have to move either to the right or to the left.

So what can we do? We can augment the definition of a TM with the option 'strek', which means don't move anywhere. That gives us a shorter program (less number of states). If we want the definition of the TM to be as simple as possible, then we can move to the right and then come back and halt. That's a possibility here, but it is a little messier.

And now we have to define the rest. What happens with all the other possibilities? If the machine is in state s and it's scanning a '1', what should it do? It should basically erase the rest of the input and come to the end of it and halt by writing a 'N'. So let's have another state q_e ('e' for erase or error) which means: "What I have seen so far did not satisfy me. I reject, I don't want to accept this input. It's not in the language." So as soon as the machine is in q_e it is just going to be replacing what it sees by blanks and move to the right.

State q_e with anything that is non-blank keeps us still in q_e . We erase the symbol and move to the right. If we are in q_e and we see a blank, we have come to the end, erased the whole thing. Then we write a 'N' and stay where we are ('-').

I am such a tidy person, so I try to clean up the tape and have just one thing written on the tape in the end ('Y' or 'N'). But of course we could write a special symbol 'N' and halt immediately, and not worry about the rest of the tape. That's completely acceptable, but then we have to redefine the definition of a TM computing deciding a language!

As I said, almost every textbook has its own unique definition of a Turing machine, but as will be obvious in due time, they are all equivalent in a certain sense. So these are just irrelevant details.

IN210 – lecture 2



Church's thesis

'Turing machine' \cong 'algorithm'

Turing machines can compute every function that can be computed by some algorithm or program or computer.

'Expressive power' of PL's

Turing complete programming languages.

'Universality' of computer models

Neural networks are Turing complete (Mc Cullok, Pitts).

Uncomputability

If a Turing machine cannot compute f , no computer can!

Autumn 1999

9 of 11

So, we know how to be formal if we really want to, and it will be rare that we do want to be this formal. Now we step back to the top of the pyramid.

This formalism that I have just described gives us one side of a bridge. We are building a bridge between the reality and the formal mathematical world. So we have seen a Turing machine, its formal definition. That's one end of the bridge. The other end is how this attaches to reality. Why is the TM really a good representation of the notion of an algorithm? That 'why' is something that we cannot prove formally, because that side of the bridge exists in the real informal world. We can only argue informally. And that has been done by a guy named Church who lived in Turing's time.

By the word 'thesis' we mean something that is not formally proven, but that is argued well enough so that people believe in it. Church's thesis – or Church-Turing's thesis – says that the formal notion of a TM is roughly equivalent to the informal notion of an algorithm in the following sense: If something can be computed by an algorithm, then that something can be computed by a TM, and vice versa.

Now how can we argue such a thing, how can we argue this kind of thesis? By showing lots of examples, basically. So it is possible to construct TMs for all sorts of algorithms that one normally uses, such as adding and subtracting numbers. And then you compose these TMs, make more complex algorithms. You can make TMs that simulate basic statements in a programming language such as Pascal. It has been done. There is a book where most of the book is actually that: descriptions of TMs which simulate basic Pascal statements. And then you know that whatever you can write in Pascal programming language, you can do it also with TMs. Which shows that TMs are as powerful as general purpose programming languages.

There is something quite wonderful that happens as soon as you have this kind of very simple, yet completely powerful notion of computation, such as the TM. You can talk about the expressive power of programming languages, and prove formally that a certain programming language is what we call *Turing complete*. This now becomes a formal notion. To prove formally that I am able to do in my programming language as much as a TM can do, I basically just need to prove that I can write a TM in the programming language. Because if I can do that, then I know that my programming language can encode or represent any sort of computation that exists in this world. It is as powerful as any other programming language. If I cannot do that then something is missing. Then my programming language cannot express all algorithms.

This tells us that in fact all programming languages are in a way equivalent. As soon as we are able to simulate the TM in the language, we are able to write all possible algorithms in that language. It's a wonderfully powerful statement. You wonder about a language such as Prolog: Can we really write everything in Prolog that we can in Java? Well, you don't have to simulate all of Java with Prolog to prove that. It is enough to just simulate the TM, to write a TM in Prolog. The TM is wonderfully simple. It takes 5 lines to write a TM in Prolog. Then you have proven that the Prolog language is Turing complete, that it is as expressive as any other programming language. Expressive meaning that it can express all possible algorithms or computations.

Another issue is the universality of computer models. If a computer model, or a real computer can simulate the TM, if it can do whatever a TM can do, that means that it can do whatever any other computational device can do. It is universal. There have been interesting applications of this idea.

For example the two gentlemen Mc Cullok and Pitts, who were actually researchers in neurological/cognitive science, produced a model which has later become a very popular model of computation, called the neural nets. They are used in algorithm design for solving difficult problems in artificial intelligence. A neuron is something that exists in a human being and of which our brain and our nerves are made. Mc Cullok and Pitts were interested in proving that their model of a neuron is in a certain sense complete. And they did so by proving that their neurons, when put together, can simulate a TM. What this means is that their model of a neuron is in a certain way complete and meaningful. Because we know that humans can execute algorithms, that we can certainly do whatever a computer can do. And if the model of our brain, of our neuron, is going to be complete then it has to be able to do at least as much as any algorithm can do. That has been proven by simulating a Turing machine using neurons.

The final issue, and that is the issue we will be dealing with here in this class, is uncomputability. We want to prove that a certain problem can not be solved by any algorithm – not by a Simula program, not by a Pascal program, not by a Prolog program, not by the Mac computer, nor the PC, nor any computer that exists in this world. It is a very powerful statement. This powerful statement, to be proven, requires just a very simple thing: We need to prove only that a Turing machine cannot solve that problem. So we don't have to go through all the computers in this world, through all programming languages, all sorts of things. It is sufficient to prove that a TM cannot do it, because of the Church's thesis which says that a TM can do whatever an algorithm or program or computer can do.

So the beauty of this model is that because it is a very simple model, it is also simple to do proofs. It is much simpler to prove that a TM cannot do something, then to prove that all the computers in the world cannot do it. Still that statement is sufficient. We prove that a TM cannot do something. By Church's thesis we have proven that nothing – no computer, no programming language – can do it. It is a very, very fundamental thing.

IN210 – lecture 2

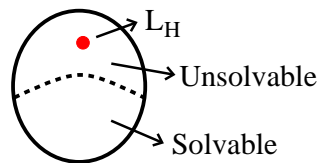


Uncomputability

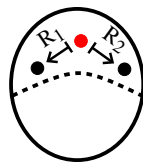
What algorithmic can and cannot do.

Strategy

1. Show that HALTING (the Halting problem) is unsolvable



2. Use **reductions** \xrightarrow{R} to show that other problems are unsolvable



Autumn 1999

10 of 11

We move on to our first big issue about computation, which is uncomputability. It is a very big issue, very important. We want to be able to prove that certain problems can not be solved by algorithms. The strategy will be in two steps: Step 1: We will come up with a first unsolvable problem. That will be the Halting problem. The unsolvability of the Halting problem is the Turing theorem, which Turing proved in 1936. Step 2: Once we have the first unsolvable problem, we can prove that all sorts of other problems are unsolvable by using a technique called reduction. We represent reductions by this kind of arrow (\xrightarrow{R}).

Language L_H represents the Halting problem. We will see what that it is in a moment. When we reduce L_H to a language L_1 we have shown that L_1 is as difficult as L_H in the following sense: If L_1 can be solved then L_H can also be solved. The Halting problem has in a certain way been converted (or reduced) to the problem represented by language L_1 . That's the meaning of the reduction. Now, since L_H is provably unsolvable, then if there is a reduction from L_H to L_1 , that means that L_1 is also unsolvable. Because if we on the contrary assume that L_1 is solvable, then we are able to solve the unsolvable problem L_H by using the reduction to L_1 , and that is obviously a contradiction.

By using reductions from the first unsolvable problem, or later from any unsolvable problem, we will be able to prove all sorts of new problems unsolvable. These two steps will give us the technique for proving problems unsolvable.

(This is a blank page)

IN210 – lecture 2

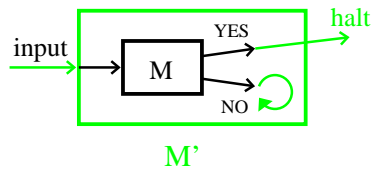
Step 1: HALTING is unsolvable

Def. 1 (HALTING)

$$L_H = \{(M, x) \mid M \text{ halts on input } x\}$$

Lemma 1 Every Turing decidable language is Turing acceptable.

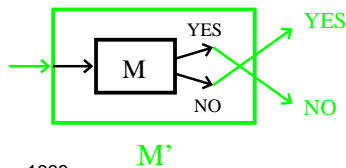
Proof (by **reduction**): Given a Turing machine M that decides L we can construct a Turing machine M' that accepts L as follows:



Lemma 2 The complement of every decidable language is decidable.

(The complement of language L is $L^C \triangleq \Sigma^* \setminus L$.)

Proof: Given a Turing machine M that decides L ...



Autumn 1999

11 of 11

Our goal now is to show that the Halting problem is unsolvable. The Halting problem has to do with the question: Does TM M halt for a given input x ? It is formalized as the formal language L_H consisting of all (concatenation of) pairs of strings M and x such that TM M halts given x as input.

String M is a code for TM M . Sometimes people invent special notation to distinguish between strings and objects. I am not doing that, so I am using M both for the machines and for machine codes, hoping that the distinction will be clear from the context.

So L_H consists of pairs, TM codes and input codes, such that the TM halts on that input. Now we need to build a little bit of theory before we can prove that the Halting problem is unsolvable. We are going to prove a few lemmas.

The first lemma is going to be that every Turing decidable language is Turing acceptable. If we can decide a language we can also accept it. If there is an algorithm or TM which decides the language, then there is an algorithm which accepts the language. The proof is very easy, but we will explain it in some detail because it is our first meeting with a reduction. I am going to represent these reductions with ideograms, with little pictures, which will be very easy to interpret. So that we don't have to use many words and a lot of ink, but just make simple pictures, look at them and understand what is going on immediately.

I am making a reduction by using M as a kind of a subroutine or subprocedure or building block for constructing another machine called M' which solves another problem. So when I use machine M to create machine M' , I am saying the following: If there is a machine M that solves problem 1, then I can use it

and create a machine M' that solves problem 2 and by doing so, I have reduced problem 2 to problem 1. If problem 1 can be solved, then so can problem 2. That is what I have proved by this picture.

Now here is how it works in the concrete case: Suppose I have the machine M which decides language L . So given input, if the input is in L this machine is going to say 'Yes', if the input is not in L the machine is going to say 'No'. So my new machine is going to do exactly the same as the old machine, except that when the old machine will write 'Yes' and halt, the new machine will just simply halt. In fact this is no change at all. The machine halts here. The change happens when my old machine writes a 'No' and halts. At that point my new machine is going to enter into an infinite loop and not halt.

So it is clear that if machine M decides a language L , then machine M' is going to accept language L . It is going to halt if and only if the original machine says 'Yes', which happens if and only if the input is in L . It is going to enter an infinite loop if and only if the original input is not in L . So by this little reduction I have proven this fact that every Turing decidable language is Turing acceptable.

Lemma 2 says that the complement of every decidable language is also decidable. What is the complement of a language? Remember that a language is a set. So the complement of a language is by definition Sigma-star, all the strings over the input alphabet, minus L . So it is whatever is not in L . But intuitively, when we think of languages as problems, then we think of a complement of a language as the reverse problem. A problem is typically defined by certain property: Decide whether this input has a certain property. If that property defines the language L , then the complement of L would be the inverse property: Decide if the input doesn't have the property. So property and non-property they correspond to the language and the complement of the language, intuitively.

For now the complement of a language is just the set complement, simply. To prove that the complement of every decidable language is decidable we use the following reduction: Given a M which decides L we construct M' that decides L^C , simply by reversing the 'Yes' and 'No', by reversing the answers given by the machine M .

I hope you are not getting too bored. We have to be a little bit formal in this class. But the formal machinery we build now will later allow us to go faster, to be more on the high level. We will then deal with the intuitive, interesting notions, but still now that we can bring them down to the ground, to something which is completely formal.

We continue next time with a couple of more lemmas, the proof of the Turing theorem, and then we will see the basic technique for proving that other problems are unsolvable.

3.3 Lecture 3

IN210 – lecture 3

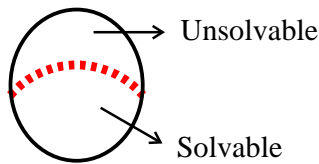


Review

Objective

techniques — how to prove that a problem is unsolvable

insights — what sort of problems are unsolvable



unsolvable (by algorithms) problems \rightsquigarrow undecidable languages

solvable problems \rightsquigarrow decidable languages

Autumn 1999

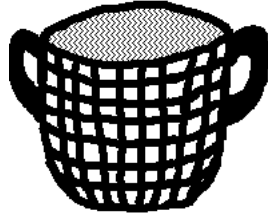
1 of 13

Our goal is to be able to distinguish between the unsolvable problems which cannot be solved by algorithms, and those that are properly solvable. We have formalized this notions of unsolvability and solution by algorithm by saying that we model the unsolvable problems by undecidable languages – we have defined what this is – and solvable problems by decidable languages.

Our objective now is to be able to make a cut through the middle of this universe of problems or languages in order to distinguish between the unsolvable ugly dark guys and those that are properly solvable, the nice guys.

We want two things. We want techniques which will allow us to prove that a certain problem is unsolvable. We also want insights. We want to be able to just intuitively get a feeling for what sort of problems are those that are unsolvable. So that when you come across one such problem in your life, you can immediately recognize it and say: “Look, this problem is probably unsolvable”. So you don’t bother with it. You don’t waste your life trying to solve an unsolvable problem.

IN210 – lecture 3

**Meaning**

- All algorithms in the world live in the basket
- Infinitely many of them — most of them are unknown to us
- Meaning of unsolvability: No algorithm in the basket solves the problem (decides L)
- Meaning of solvability: There is an algorithm in the basket that solves the problem (but we don't necessarily know what the algorithm looks like)

Now a little bit about the meaning of this, what we are doing. We introduce another ideogram here, which is this big basket. It is where the algorithms live. So all the world's algorithms live in this basket. Every algorithm is there. The meaning of unsolvability is that for a given problem there is no algorithm in the big basket. So none of the world's algorithms solves the problem. That's the meaning. One subtle point, which we will return to later, is that in order for a problem to be considered solvable we don't have to be able to find the algorithm which solves it, or to decide which algorithm is the correct one. It is enough that the algorithm is in the basket!

IN210 – lecture 3



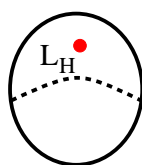
Techniques

To prove that

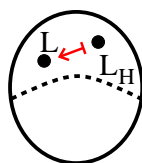
- **L is solvable:** Show an algorithm
- **L is unsolvable:** Difficulty: Cannot check all the algorithms in the basket. Cannot even see most of them, because they have not yet been constructed . . .

Strategy

1. Show L_H (HALTING problem) undecidable using diagonalisation .



2. Show another language L undecidable by **reduction**: If L can be solved, so can L_H .



Autumn 1999

3 of 13

About the techniques we are using: If you want to prove that something is solvable, usually you will show an algorithm for the problem. The algorithm solves the problem, and showing the algorithm solves the problem of proving solvability!

Unsolvability is a little bit more of a tricky matter, because there is a difficulty. And the difficulty has to do with the huge basket full of algorithms. The problem is that not only are there infinitely many algorithms in the basket, but actually most of them have not even been constructed. We don't know what they are. So you cannot possibly go through this whole basket and check one algorithm after another and say in the end that none of them solves your problem. So what do you do? How can you prove that a problem is unsolvable?

And the solution is this little strategy here: We find one unsolvable guy, one unsolvable problem, the first one. And that will be the Halting problem. We prove that the Halting problem is unsolvable by using this powerful technique that we have seen already, diagonalization.

So diagonalization will give us the first unsolvable problem, the Halting problem. And once we have that, then we can use it in a technique called reduction to prove that all sorts of other problems are unsolvable by reducing our unsolvable problem – usually the first one, but you can reduce any other unsolvable problem – to another problem. And the meaning of this is the following: When I reduce an unsolvable problem to another problem L , then it turns out that if I can solve L – if I can decide L – then I can also decide the original unsolvable problem. But since the original problem is unsolvable, this amounts to an argu-

ment that the new problem L is also unsolvable. So these two steps combined give us a technique for proving unsolvability of all sorts of problems.

IN210 – lecture 3



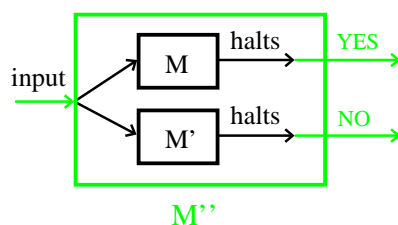
Lemma 1 L decidable $\Rightarrow L$ acceptable

Lemma 2 L decidable $\Rightarrow L^c$ decidable

We proved lemma 1 and 2 last week. We continue today with some more lemmas and the main theorem.

Lemma 3 If both L and its complement L^c are acceptable then L is decidable.

Proof: Given Turing machines M and M' which accept L and L^c respectively we can construct a machine M'' which decides L as follows:



Autumn 1999

4 of 13

In order to prove our first problem unsolvable, we need to build up a little bit of theory. We started doing that last time, and we proved two lemmas. Lemma 1 says that if a language is decidable, then it is also acceptable.

Decidability and acceptability are two notions of a Turing machine solving a problem – deciding whether a word is in the language or whether something has a property or not. You can think of the formal languages as being properties. So each language is a set of strings that has a certain property, for example the strings that are sorted and things like that. We will see a lot of them later.

There are two ways in which an algorithm can go about deciding whether an input has a certain property. They give rise to decidability (saying 'Yes' for all strings in the language, saying 'No' for all strings not in the language) and acceptability (halting for all strings in the language, non-halting for all strings not in the language). Lemma 1 says that decidability is a stronger notion, strictly stronger in the sense that everything that is decidable is also acceptable. We will see that the reverse is not true, there are acceptable languages that are not decidable.

Lemma 2 says that if L is decidable then its complement is also decidable. Remember that the complement is intuitively the reverse property. 'Sorted' is say a property, 'unsorted' (non-sorted) gives rise to the complement of the language. That's how you think about the language and its complement, intuitively. Formally L^c is just a set complement of L . A formal language L is a set of strings. L^c is the set complement, namely all the strings that are not in L .

We continue today with lemma 3 which says that if both a language and its

complement are acceptable, then the language is decidable. How can we prove this? We use another one of those boxes. This box is an ideogram and it shows us how to construct an algorithm M'' , given two algorithms M and M' .

So assuming that M and M' exist which respectively accept L and L_C , we can construct a machine M'' which decides L , in the following way: We run the two machines M and M' in parallel. We give them the same input. The input, being a string, must always belong to either to L or L_C . Since M and M' by assumption accept L and L_C , one of those machines will eventually halt. If M halts we say 'Yes', if M' halts we say 'No'. That gives us M'' .

Now the question is how do we run two machines in parallel by using only one machine, M'' ? And the answer is: Its fairly easy. You basically run one machine a little bit and then store its state and all you need to know about it, its configuration. Then you run the other machine a little bit, and so on. And you can allocate a part of the tape for one machine and a part of the tape for the other machine. If you run out of tape then you have to do some copying and erasing and moving the data from one place of the tape to another, and so on.

So these are just messy details which you can easily reconstruct yourselves. They can be done. So we are not going to deal with such details a lot in this class, especially not in the lectures, but I do recommend that you spend some time with them and really check that you could in principle, given enough time, create such a machine – that you could give all the details.

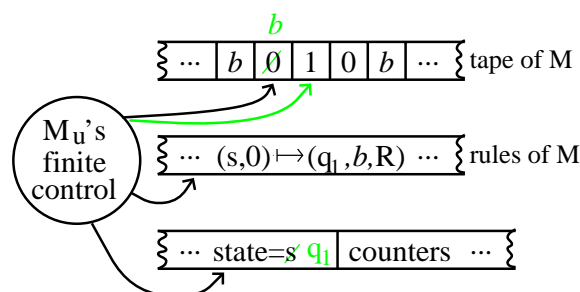
IN210 – lecture 3



Lemma 4 $L_H = \{(M, x) \mid M \text{ halts on input } x\}$ is Turing acceptable

Proof: The **universal Turing machine** M_u accepts L_H .

- M_u works like an ordinary computer: It takes a code (program) M and a string x as input and simulates (runs) M on input x .
- M_u exists by Church's thesis.
- To **prove** existence of M_u we must construct it. Here is a 3-tape M_u :



Autumn 1999

5 of 13

Lemma 4 gives us the actual Halting language or Halting problem, L_H . L_H consists of pairs of M and x where M is a Turing machine code and x is an input code. Notice that both M and x are strings, therefore the set of all pairs of M and x (actually the string concatenations of M and x) is a formal language. So the language consists of those pairs which are such that the machine M halts given input x , in other words, that M accepts x .

So lemma 4 says that this language is Turing acceptable. There is a Turing machine which accepts L_H . We call that Turing machine the universal Turing machine, M_u , and since this Turing machine is going to play an important role in our reductions, and in fact in theory in general, I will describe it in some detail. Notice that M_u is in principle what we call a general purpose computer or a stored program computer. Because what it does is exactly what your typical computer does. It reads both the program and the input, and then runs the program on the input. So we want to see that such a Turing machine indeed can be constructed.

And you would expect that it must be so because of the Church-Turing thesis which says basically that whatever you can do on an ordinary computer, you can do it also with a Turing machine, and vice versa. And since we believe in this thesis by and large, we are ready to believe that M_u exists, and for being convinced about its existence that might be enough. But that is not enough for a formal proof because we want to prove that M_u exists and then use it further for all sorts of other proofs so that we have a consistent and well defined theory.

So to prove the existence of M_u we need to construct it. I will use a construc-

tion which is standard and which uses 3 tapes. It is fairly easy to see that any Turing machine which uses some number of tapes – k tapes – can be simulated by a Turing machines that uses only one tape, with some storage management, garbage collection, moving data left and right when the machine runs out of tape, etc.

So now we have 3 tapes and we use them as follows: One tape is used for simulating the tape of M . Remember what we are doing: We are simulating or running, machine M on input x . When the simulation starts this tape of M will contain the input x , '010' in our example.

The second tape will store the program of M , the description of M , and the most important part of it is the transition function delta, the rules. So you can imagine that this tape here has some encoding of the transition table of M . You cannot just store all sorts of things like q_1 and so on. But these are encoded with 0's and 1's – imagine your favorite encoding. So there is some work actually for the poor, little Turing machine – which is rather kind of a silly, not very clever device – to figure out this code, but imagine that this can be done. And again you can understand actually how the details work out if you work with them a little bit.

M_u will keep track of the state of M and some other data on its third tape. So the third tape is kind of a work tape for M_u where M_u keeps its local data, its variables and things. One of them is the state of M . So initially M , which is being simulated, is in state s and M is looking at the first character of its input, a '0'. And then M_u will use its second tape. It will look left and right – go through the transition function of M – until it finds the entry of the transition function which corresponds to the current situation, which is a '0' being scanned and s being the state.

So it finds out that M in that situation needs to change the state to q_1 , erase the character that is being scanned and move its read/write head to the right. So M_u will do exactly that action: Replace the 0 on the first tape by a blank, move the head of the first tape to the right, and change the state stored on the third tape – replacing the s by q_1 . And q_1 is again a code of something. We don't know in advance how many states M will have. So we need to use some counters here, some ways of representing these q 's. I say here s and then q_1 , but what you really need is to be able to count up to n states. So when we represent q_1 by 0's and 1's it will be a string and in order to figure out exactly which q this is, we will have to use some counting. Details, details, details.

Aside from these details the whole thing is fairly simple and easy. Now the story continues. A new step of M is simulated by M_u : The third tape says that M is in state q_1 , and the read/write head of the first tape is scanning a '1'. Again M_u will search through the second tape until it finds the corresponding rule, apply it, etc. And then of course when M halts, M_u will also halt. If M says accept then M_u knows that M has accepted, or if M says 'Yes' M_u knows that M has said yes. So the simulation is rather simple and it is easy to see how M_u can do exactly what M is doing.

In particular, if M accepts its input x then M_u simply halts and accepts its input (the concatenation of the strings M and x). So effectively M_u accepts L_H , the Halting language, or solves the Halting problem in this accepting, weak sense.

IN210 – lecture 3



Lemma 5 $L_d = \{M \mid M \text{ does not halt given its own code as input}\}$ is not Turing acceptable.

Proof: Suppose there exists M_d which accepts L_d . Does M_d accept its own code?

- If yes — it accepts a string which is not in L_d .
- If no — it fails to accept a string which is in L_d .

So in either case it fails, hence a **contradiction**.

Note: L_d is the diagonal language — a special version of L_H^c .

Autumn 1999

6 of 13

Now the final lemma, lemma 5, which will actually give us the first unsolvable problem. It says that L_d is not Turing acceptable. L_d consists of machine codes which have the property that they don't accept their own code as input. In other words: L_d consist of all Turing machine codes M which describe machines that do not halt given their own code as input. What a strange problem, what a strange language. We will see in a moment where this is coming from.

So lemma 5 says that L_d is not Turing acceptable. That is what we are going to prove. And the proof will be a diagonalization argument, but in the beginning it will not be completely clear why this is diagonalization. Afterwards I will show you a second argument which is the diagonalization that we are used to, and then everything will fall into place. But the diagonalization proof in disguise is sleek and elegant and very short, so let us look at it first.

Towards a contradiction we suppose that there is a machine M_d which accepts L_d . What happens if M_d is given its own code as input? Does M_d accept its own code? There are two possibilities: 'Yes' and 'No'.

If 'Yes', if M_d accepts its own code, then M_d accepts a string which is not in L_d . Because L_d is defined as all machine codes which do not accept their own code. So if M_d accepts its own code, then it accepts a string which is not in L_d . In this case M_d is not a proper machine for L_d .

If 'No', if M_d doesn't accept its own code, then it fails to accept a string which is in L_d , namely its own code. Because if M_d does not accept its own code then the code of M_d belongs to L_d , and therefore M_d should have accepted it.

So M_d in either case fails to accept L_d , which completes the contradiction.

So the assumption that there exist such a machine M_d , must be wrong.

As these sleek concise arguments go, you need to spend some time on it, look at it and ponder it and see actually how beautiful and elegant it is, and that it really does its job. The question is how in the world did people come up with this strange looking L_d , and this strange looking argument?

IN210 – lecture 3



Alternative proof of lemma 5:

	ϵ	0	1	00	01	10	11	000	...
ϵ									
0									
1									
00	1	0	0	1 0	0	1	0	0	
01	0	1	0	1	0 1	1	1	0	
10									
11									
000									
\vdots									

- We have strings as column labels
- We have Turing machine (codes) as row labels
- The 1's in each row define the set of strings each TM accepts.
- After flipping the diagonal elements, the 1's on the diagonal represents those machines which don't accept their own code as input
- No Turing machine can possibly accept that diagonal language!

Autumn 1999

7 of 13

The explanation and complete clarity comes when you look at the same problem in terms of our huge matrix where we represent all algorithms (all Turing machines) and the languages which they accept (the strings for which they halts).

So we once again look at this big matrix. We have all the strings as column labels and all strings as column labels, and some strings among the row labels are going to be Turing machine codes. So suppose that these yellow-shaded rows are Turing machine codes. And whenever we have a Turing machine code as a row label, we use the remainder of that row to represent the language that the Turing machine accepts by putting a '1' for those strings which are accepted by the Turing machine, and a '0' for those strings that are not.

So the empty string is accepted by the machine '00', '0' is not accepted, '1' is not accepted, '00' is accepted, '10' is not, and so on. The same for the other rows – we just put 0's and 1's and so on. And then as in the proof in the first lecture where we were talking about having more problems than solutions, we look at the diagonal and we flip the diagonal – replace 1's by 0's and 0's by 1's.

So what is the diagonal here? The diagonal tells us what each Turing machine will do given its own code as input. And a '1' here means that the Turing machine accepts its input. Now when we flip those 1's into 0's, then we have 1's in those places where the Turing machine does not accept its own input, in other words L_d – the diagonal language. So in the technical jargon, this language is called the diagonal language of the Halting problem. And because every valid TM gives the wrong answer given its own code as input – remember that the

diagonal language was constructed by flipping all the diagonal elements – there cannot be any TM which accepts L_d .

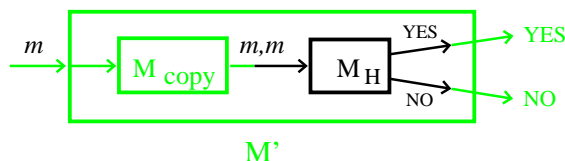
So this is a completely valid proof for lemma 5. And this alternative proof gives us a clear diagonalization and a clear origin of L_d . So now we have the whole thing.

IN210 – lecture 3



Theorem 1 L_H is undecidable.

Proof: Suppose Turing machine M_H decides L_H . Then the following machine M' decides L_d^c :



- By Lemma 2 L_d is also decidable.
- Then Lemma 1 gives that L_d is acceptable.
- We have arrived at a contradiction since by Lemma 5 we know that L_d is not acceptable.

Autumn 1999

8 of 13

We are now ready to prove our first undecidability result, the big one, the important one, which is that the Halting problem is undecidable! So we have our first undecidability problem finally. And why is the Halting problem undecidable? Because if we can decide the Halting problem, we can also decide L_d .

First of all notice that L_d^c – the language consisting of Turing machine codes which do accept their own code as input – is just a special case of the Halting problem, where instead of giving as input a Turing machine M and some x , this x is just another copy of M , the Turing machine code.

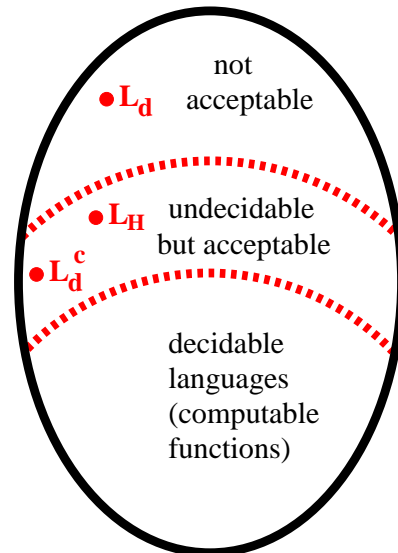
The box on the foil shows the reduction. If M_H decides L_H , then we have a decision procedure M' for the complement of the diagonal language, L_d^c : We take the input M for M' and basically copy it two times, feed it to the machine M_H which supposedly solves the Halting problem, and say 'Yes' if M_H says 'Yes' and 'No' if M_H says 'No'.

So if M_H exists, so does M' , and then we have that L_d^c is decidable because M' decides it. But this cannot possibly be true because if L_d^c is decidable, then by lemma 2 L_d is also decidable, and by lemma 1 L_d is then acceptable, but as we have seen (lemma 5) L_d is not acceptable. Hence we have a contradiction, and M_H cannot exist. In other words, the Halting language is undecidable, which is what we wanted to prove.

IN210 – lecture 3



Conclusion



All languages \supset Turing acceptable languages
 \supset Turing decidable languages

Now we have accomplished quite a bit. We have divided our universe of problems into not only two classes, not only two sub-worlds, but actually three. Just kind of surprising. Among all the unsolvable problems strangely there are problems which are more unsolvable than others. There are problems which are not acceptable, such as L_d , problems which are acceptable but undecidable, such as L_d^c and the Halting language L_H . And then of course all sorts of problems that are both decidable and acceptable.

So we have divided all the languages into sets which contain one another. The big set of all languages contains the Turing acceptable languages as a proper subset, and that set contains the decidable languages as a proper subset. This we have proven by these few lemmas and the theorem.

IN210 – lecture 3

Meaning

An example with $\Pi = 3.14159265359\dots$:

$$L_1 = \{X \mid X \text{ is a substring of the decimal expansion of } \Pi\}$$

$$L_2 = \{K \mid \text{There are } K \text{ consecutive zeros in the decimal expansion of } \Pi\}$$

Classify L_1, L_2 as

- not acceptable
- acceptable but not decidable
- decidable

Note: Only problems which take an infinite number of different inputs can possibly be unsolvable.

Autumn 1999

10 of 13

Now I will talk a bit about the meaning of unsolvability. I will show you an example which has to do with number pi (π), which is what is called a transcendental number. π is an infinite number, it has infinite many digits. If you look at the number $1/3$, that is also an infinite number. But there is a huge difference between the two numbers. This $1/3$ number although it is infinite you can write it in a certain way. All the digits are known because this digit 3 just repeats itself. So the number $1/3$ is kind of periodic.

It is known that number π is not periodic. There is no such repetition. Basically every digit that comes, it is kind of a creative thing. It comes out of the blue, and it cannot be figured out trivially from the previous calculated digits.

So π is not repetitive. Nevertheless π can be approximated to an arbitrary degree. Which means that we do have algorithms which give us as many digits of π as we would like to have.

We look at two languages. In L_1 we ask, given a decimal string x , whether x is a substring of the decimal expansion of π . If x is equal to, say, 415 then x is in L_1 because $\pi = 3.14159\dots$. In L_2 , given an integer k , we ask whether there are k consecutive 0's in the decimal expansion of π .

So these two languages they are quite similar, right? And now I ask you: Can you place L_1 and L_2 onto the map on the previous foil? For each of L_1 and L_2 , is it not acceptable, or is it acceptable but not decidable, or is it decidable?

The obvious way to try to decide L_1 is to make an algorithm that generates the digits of π , and then run the algorithm and check if the substring x appears.

If the machine stops and answer 'Yes' then everything is fine. But what if the algorithm just runs and runs, searching for the substring x ? Suppose you wait 70-80 years, you are getting very old, your beard is very long. Everyone thinks you are crazy, but you are still standing in front of the computer. And the

computer just doesn't give you the answer. What then?

We are building intuition now. This is the difference between accepting and deciding. Deciding means definitively saying 'Yes' or 'No'. So if you are sitting in front of a computer that decides a language then you know you may have to wait for one million years, but ultimately the answer will come. Now, if your machine is just accepting the language, you don't even know that after one million years the answer will come, it may never come. Because the way the machine says 'No' is: It remains silent. So only positive answers come.

Language L_1 can be accepted in exactly the way we were describing. So you generate the digits of π and go consecutively through them and just keep checking if x is a substring of π , and when you find it you say: 'Stop, I have found it!' That's the end. But if you don't find it, then the algorithm just runs forever. It doesn't accept the strings that are not in the language. It only accepts the strings that are in the language. So L_1 is acceptable, but accepting is a very weak notion of solving a problem.

Whether L_1 is actually decidable or not, we simply don't know. So I cannot tell you the answer. But we know for sure that L_1 is acceptable, and we in fact do suspect that it is not decidable. But we don't know enough to prove it. You will need to know the nature of this number π better in order to actually be able to prove that L_1 is undecidable. We cannot do that presently.

However we can prove that L_2 is decidable, although we cannot really show an algorithm. So this is an interesting case where we have a non-constructive proof of decidability. And let me tell you how it goes:

Either there is an number n such that there are not more than n consecutive 0's in the decimal expansion of π , or there is no such number. So, either there is a maximum number of consecutive 0's in the expansion of π , or there is not – meaning that there are arbitrary many consecutive 0's in the expansion of π . If there is such a number n , then the algorithm is very simple: You compare your k with n and if k is smaller then you say: 'Yes, there are k consecutive 0's in the decimal expansion of π '. If k is bigger then you say 'No'.

If there is no maximum number of consecutive 0's in π – if there can be arbitrary many – then you always say 'Yes'. So in either case the algorithm exists. But which algorithm is the right one – what is the maximum number of consecutive 0's in the decimal expansion of π and is there such a maximum – we don't know. So we don't know the algorithm, but we know that an algorithm for deciding L_2 exists in the big basket of algorithms.

This is quite subtle and an indication that our notion of unsolvability is maybe to weak, in the sense that some languages that are unsolvable in practice (we might even be able to prove formally that we cannot determine which algorithm in the basket is the right one for deciding L_2) are being classified as solvable (because we have proven that there do exist an algorithm in the basket for deciding L_2).

The example is not a typical situation. We will see more typical situations later on. But it does illustrate some subtle aspects of the meaning of acceptability and decidability, which I wanted to point out to you.

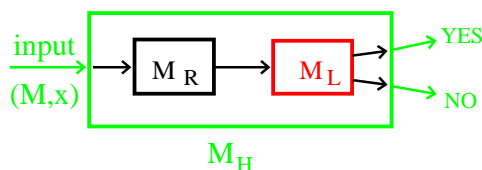
Another fine point is that only languages with infinite number of inputs can possibly be unsolvable. We can decide any language L which take only a finite number of different strings as input, in the following way: We construct a large, but finite array (as large as there are different inputs), and for each possible input x we put a '1' at the corresponding array entry if x is in language L , and a '0' otherwise. The algorithm which decides L will then, given an input x , do a look-up in the array at the index corresponding to input x , and answer 'Yes' if the array entry contains a '1', and 'No' otherwise.

That is why we will only consider languages with infinite inputs in this class.

IN210 – lecture 3



Reductions



Meaning of a reduction

Image: You meet an old friend with a brand new $M_\$$ -machine under his shoulder. Without even looking at the machine you say: “It is fake!”

How the reduction goes

Image (an old riddle): You are standing at a crossroad deep in the forest. One way leads to the hungry crocodiles, the other way to the castle with the huge piles of gold. In front of you stands one of the two twin brothers. One of them always lies, the other always tells the truth. You can ask one question. What do you say?

Autumn 1999

11 of 13

Now, theorem 1 has given us step 1 of our strategy for proving unsolvability. We have our first unsolvable problem, L_H . The second step is: We want reductions. We want to be able to reduce L_H to another language L in such a way that if L is decidable, then so is L_H . And since we know that L_H is undecidable then L can not possibly be decidable either. So that gives us a strategy, or technique, for proving that all sorts of other problems are undecidable also. Which puts us into business.

So, this technique is called reduction. I will represent reductions schematically. But you, when you write your proofs, you will have to spell out the whole thing because we want arguments and not just pictures in the answers of your homework and exam questions. I can afford to be sketchy because this is a lecture and I have done my education, got all my degrees and things, so I can improvise. But you have to be careful and strict.

As improvisation goes this green, outer box here represents M_H , a solution to the Halting problem, which is just about complete. So M_H actually solves the Halting problem provided one thing which is shown here in red – provided M_L . So the reduction is in fact almost a complete algorithm except that it calls a subroutine. It uses a kind of a black box, a slot here – the red box – which we claim doesn’t exist.

And by managing to produce this whole machine – minus this little red box – that solves the Halting problem, we effectively show that the little red box cannot possibly exist. Because if it does exist, then we plug it in here and this whole machine solves the Halting problem. We know that that is impossible.

I invoke two images to illustrate this concept. We will see in a moment an unsolvable problem which I call L_{\S} . The first image goes like this: Imagine that you walk down Karl Johan's street and you meet your old friend and he's carrying something rapped up in newspaper under his shoulder. You ask, after the usual greetings exchange: "What is it that you are carrying?" And he says: "Well, you know, I went to a store and I just bought a little machine that solves the L_{\S} problem. It's a new thing coming out of America, fresh from the shelf – nobody knows about it yet." And you say right away: "No, no, no, no, no. This M_{\S} , it cannot possibly exist." And he says: "Oh, wait a minute. I mean, look: I just bought it, number one. And number two: How do you know that it cannot exist? It's here, under my shoulder, and you didn't even see what I had in the newspaper."

The point is that you don't have to see it. You know that it cannot exist. Why? Because you have this other machine here, M_H , and if your friend can bring M_{\S} and plug it in here, then the whole thing is a complete machine for L_H – for the Halting problem. But since you know that such a machine cannot possibly exist, then you know that his machine that solves L_{\S} cannot exist either. So without looking at it, without even bothering to see what the thing is, you know it cannot exist. And that's the power of the reductions.

Image 2 is about the way these reductions go, how you can think about them: There is this old riddle where you are coming to an intersection which splits into two roads: One leading to some kind of disaster, and another leading to something very good. So this is the road you definitely want to take. And there is a guy standing on the intersection. You know about this guy already. It's one of the two twin brothers. One of the twins always lies, the other one always tells the truth. And you don't know who is who.

The brothers they know the way because they live there. You want to ask the guy the question where to go so that you for sure go this way (to the castle) and not this way (to the crocodile). For some reason or other you are only allowed to ask one question, so a lot depends on your question.

What do you ask? Do you know this? You do? OK, I will tell you what it is. You ask the guy: "What would your brother said if I asked him: What is the way to the castle?" And then if this is the lying brother he would lie about his other brother, so he would say: "This way, this way!" (pointing towards the crocodile) And if it is the truth-saying brother, he knows that his brother would lie, so he says: "My brother would point to this way." (to the crocodile) In either case it is the wrong way you will hear, so you choose to go the other way. You kind of negate the negation and get the right thing.

The same is true for reductions. So towards the contradiction you assume you are given a solution to the undecidable problem. And then your reduction will consist in asking this supposed machine a clever question in such a way that the machine will actually answer what you really want to know – and that's the answer to the Halting problem.

So M_R will in effect transform the question that corresponds to the Halting problem, to the question which this supposed machine wants, so that if this machine M_L can say 'Yes' or 'No' in the right way, then effectively it has answered the Halting problem.

So this was just for explaining this box ideograms. The whole argument is an argument by contradiction. You will see a few of them in group sessions, and I very strongly recommend that you do reconstruct one reduction. You don't have to do a lot of these proofs, but at least one you should really understand completely. And this one I am hoping to do right now. Here it comes.

IN210 – lecture 3

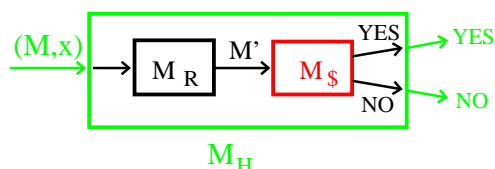


A typical reduction

$$L_{\$} = \{M \mid M \text{ (eventually) writes a \$ when started with a blank tape}\}$$

Claim: $L_{\$}$ is undecidable

Proof:



M' :

Simulate M on input x ;
IF M halts THEN write a \$;

Important points:

- M' must not write a \$ during the simulation of M !
- 'Write a \$' is quite an arbitrarily chosen action!

Autumn 1999

12 of 13

I am showing a typical reduction, and I am going to do it slowly, but you may even want to go into more detail at home than I am doing here.

The word 'typical' is shown in red, because you want to understand what I am doing here as a kind of a template. You can prove all sorts of other problems unsolvable in pretty much the same way. So if you understand this one proof properly, you can understand and construct all sorts of other proofs.

The problem that I am showing undecidable is what corresponds to the language $L_{\$}$, consisting of the codes of all Turing machines M which will eventually – when started on a blank tape – write a dollar sign.

The input is a Turing machine code. It's a program. You can imagine it is written in Java. And the question is: Does this Java program eventually write a dollar sign? And notice that there is no input, so the input is irrelevant. Maybe there is some input written on the input tape, but M will not even look at it. So we can assume that M starts with an empty tape. And the question is simple: Will M eventually write a dollar sign?

I claim that $L_{\$}$ is an undecidable language. Now, the proof of this claim is a reduction, and I am using this whole situation to show you a typical reduction. This reduction will construct a solution to the Halting problem, given a machine $M_{\$}$ which decides $L_{\$}$. The heart of the reduction is this machine M_R which reduces the Halting problem to $L_{\$}$, our new problem.

The whole argument is by contradiction. We say: Assume towards the contradiction that there is a machine $M_{\$}$ which solves the $L_{\$}$ problem. Then we can create – by using that machine – a solution to the Halting problem, in the

following way: ... And then at the end of the argument we say: Since we know that M_H does not exist, the assumption that M_{\S} does exist, must be incorrect because it leads to a contradiction. What is the contradiction? The contradiction is that M_H exist. Because if M_{\S} exists, so does M_H . But we have proven before that M_H cannot exist.

So this whole argument is by contradiction and the heart of the matter is the machine M_R which produces M' given M and x . So we know focus on that.

First I will explain what M' is and what M' does. M' is an instance of the new problem L , produced by M_R . M' is very simple, and in a typical reduction M' will look like this, be some kind of variant of the M' shown here.

Remember that M' is a machine code. M' as a machine will simulate M on input x and then write a dollar sign after M has halted, if it halts. Very simple. And notice that given M and x , the reduction from M and x to this M' is a proper reduction in the sense which I will define shortly, namely that it maps the 'Yes'-instances to 'Yes'-instances and 'No'-instances to 'No'-instances.

Why is it so? The key to understanding the reduction is this little line: "Simulate M on input x ." So M' is effectively running machine M on input x . And then it is going to write a dollar sign. So it is going to write a dollar sign if and only if this simulation eventually ends. Deciding whether the dollar sign will be written amounts to deciding whether M will eventually finish dealing with x – whether it will halt given input x .

There are two comments. Comment one: In its simulation M' must never write a dollar sign. This is easy to fix because if for example M uses the dollar sign, then you can simple modify the M by replacing the dollar sign by a cent sign, by some other sign. So that's an easy fix.

So you know that the dollar sign doesn't have to be used. You can use anything else instead. You can assume that the dollar sign cannot possibly be written on M' tape during the simulation of M on input x . The only way a dollar sign can be written is if M finishes its run on x , in which case the dollar sign is surely written. So the dollar sign is written if and only if M halts on x . So this is a proper reduction.

Comment two: Notice that this writing of a dollar sign is in fact an arbitrary action. We can say not a dollar sign, but 'Hello'. Does M ever say 'Hello'? Does M ever wave its right hand? So, you can replace the line "writes a \$" by "says 'Hello'". And then you have another problem shown undecidable, another reduction.

So using this as a kind of a template to make another reduction, is very simple. But that doesn't mean that any action goes. Because sometimes it is impossible to modify the first part (simulation of M on x) so that the action never happens. A simple example is deciding whether M ever moves its head right or left.

You can say: 1) Simulate M on x . 2) Move your head left or right. It's seems like you have proven this question of whether the machine ever moves its head left or right, undecidable. But you haven't really because you cannot eliminate the movement of the head from the simulation. It is not impossible. This is subtle.

So this is why you cannot use this idea to prove that anything is undecidable. Up to here the story is extremely simple, and I am hoping you can understand it completely when you look at your notes for ten minutes at home. Please do so, because if you understand this you have understood a major part of this class. In fact it is a major part of algorithm theory, and a major part of this whole business of producing algorithms, which has to do with whether there is an algorithm for something or not.

IN210 – lecture 3

 M_R :

Output the M_u code modified as follows: Instead of reading its input M and x , the modified M_u has them stored in its finite control and it **writes them** on its tape. After that the modified M_u proceeds as the ordinary M_u until the simulation is finished. Then it writes a \$.

Reduction as mathematical function

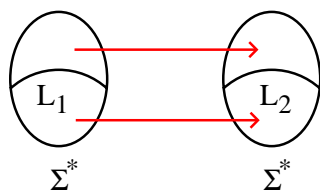
Given a reduction from L_1 to L_2 . Then M_R computes a function

$$f_R : \Sigma^* \rightarrow \Sigma^*$$

which is such that

$$x \in L_1 \Rightarrow f_R(x) \in L_2$$

$$x \notin L_1 \Rightarrow f_R(x) \notin L_2$$



Autumn 1999

13 of 13

The heart of the matter is the machine M_R which produces M' given M and x . This is an important part which defines actually what we mean by a reduction. So a reduction is actually an algorithm which we here call M_R , which computes a certain function, let's call it f_R , that maps strings to strings in the following way: If string x is in L_1 then the string $f(x)$ is in L_2 and if string x is not in L_1 then the string $f(x)$ is not in L_2 . To make the things more clear we use a picture. This time the oval represents the set of all strings, and here is L_1 and here is L_2 . So the essence of the reduction is that it is going to map strings in L_1 into strings in L_2 , and strings that are not in L_1 into strings that are not in L_2 . Or in other words, it is going to map all the 'Yes'-instances of L_1 into 'Yes'-instances of L_2 , and all the 'No'-instances of L_1 into 'No'-instances of L_2 .

In our example the Halting problem is L_1 . When we are reducing the Halting problem to this other problem $L_\$$ (which is L_2 in the definition above), we are computing an instance of $L_\$$, which is M' . So the reduction machine M_R will take an instance of the Halting problem and compute an instance of the new problem $L_\$$ in such a way that if the L_H instance is a 'Yes' instance then the $L_\$$ instance is a 'Yes' instance, and if the L_H instance is a 'No' instance then the $L_\$$ instance is a 'No' instance. That's the meaning of the reduction. So if we can produce such a thing, and that is exactly what we did on the previous foil, then we have reduced the Halting problem to $L_\$$. And if we can say 'Yes' or 'No' to $L_\$$ – if we have the little red box – then we have the whole machine complete and we can solve L_H . Which completes the contradiction.

What comes now is a more subtle, but still simple, and I do want you to

understand this also in full detail, in which case you will just fly through this part of the exam. It's extremely simple once you understand one reduction properly and completely. Then all of them are very clear and you can produce new ones very easily.

We now look at M_R – the reduction – as a machine. How does it function? What is it? What is really this M_R ? This is the heart of the matter because M_R is our reduction. The machine M_R is what we are constructing. M' is just something that is being output by M_R . M' is not a machine. It is just a code. It is a piece of software. It is a string. So the tricky part here is understanding what is hardware and what is text. M' is text. M_R is hardware.

What is M_R as hardware? First, what does M_R do? M_R will take the string (M, x) as input and produce the code for M' , which is also a string, as output. So M_R will turn one string into another. M_R will output M' , and M' will consist mainly of the code of the universal Turing machine M_u , but it will be modified a little bit. And most of what M_R does is actually this simple modification of M_u . When we understand this little modification, then we are done, basically.

Remember that the code of M_u is just a string. M_R stores in its memory – in its state control – a slightly different version of the standard M_u code. We will go into details a little later. So M_R basically has a big program which is almost finished. It just needs some things to be put into certain slots. So it has this modified M_u piece in its state control and then it reads M and x , put them into the empty slots of the modified M_u piece, and output the whole program as M' . M_R is finished.

So this is how M_R functions, and now let's see what the modification are about. How is M_u actually modified? This M_u that M_R is storing in its hardware – in its finite control – is non-standard in the following sense: Instead of reading its input M and x , M_u has both M and x stored already in its finite control. And it begins its operation by writing them – by writing M and x – on its tape. Following that moment this M_u operates precisely like the standard universal Turing machine: It runs M on input x , and does exactly what M would do on input x . And after the simulation, if it ever ends, M_u will write a dollar sign on the tape. That's really the whole story.

M_R has a stored text, the modified M_u . It's a text of a program, which is almost finished. It just needs to be modified a little bit. And in this text, at the very beginning, there are commands which say: Write on the input tape the following text. And this following text is missing, it's like there is an empty slot there.

And all that M_R does is that it takes its own input – which is the pair M and x – and writes it into that particular slot, and outputs the result. The result is this whole piece of code M' . So his whole piece of code M' will begin its operation by writing M and x on its tape. M and x are stored as constant strings in M' . And then following that moment M' will function exactly as M_u . It will look at what is written in its input tape, which is M and x . It will run M on input x and do exactly what M is doing.

So M' will of course after this whole simulation, write a dollar sign. And this will happen of course if and only if M halts on input x . And naturally if M would be using normally a dollar sign, then M_R will go through M and x and just replace every dollar sign with a cent sign. So there is a little subprogram which says: Replace all the dollar signs by cent signs in M and x . So that M doesn't possibly write a dollar sign.

So this is the end of the story. If you look at this reduction very carefully then you know all the reductions of a certain kind. You will see a few of them in the group exercises. You can try creating your own reductions. They all have to do with actions of Turing machines. So what does a Turing machine do? Well,

just about anything. It decides, given a Turing machine as input, whether the Turing machine is going to smile, wave its hand, give you a wink of the eye, etc. It's all undecidable. Because anything goes as an action in the reduction we have went through. Except some very trivial actions which we cannot prevent from happening during the simulation of M on input x . For example whether it will ever move its read/write head or whether it will ever write anything given a blank input tape.

So the properties of programs are by and large undecidable. For example in syntax checking, we would like to know if a program is semantically correct. Say I give you a student assignment: Make program for sorting. And I would like to have a kind of a piece of code that checks your program automatically. Are they correct or not, are they correct sorting programs? Well, there is no such a program that checks whether programs are correct, simply because it is not even possible to check whether the program halts or not. Not even that. Of course, for some programs you can deduce that they will stop or not, but not for a random program.

Whether the program does any sort of action such as solving sorting is undecidable because you can always create a M' which says: Simulate M on x and then run the sorting algorithm. So the sorting algorithm will be run if and only if M halts on x .


So these proofs will be actually very easy. You just need to understand one of them, and that one we have seen today. So please pondering on it a little bit, try to understand all this details. It's a little bit messy to understand because machines produce each other and some machines are code. Some actually are supposed to exist and some don't exist, for example M_s . Understand all of this and you have understood a major part of this course.

We continue next time with other kinds of reductions – reductions which have nothing to do with Turing machines. So that we can talk about other kinds of unsolvable problems also.

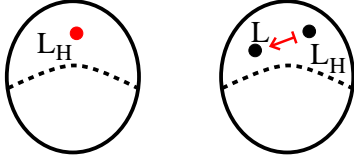
(This is a blank page)

3.4 Lecture 4

IN210 – lecture 4

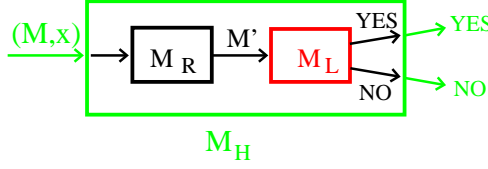


Review



- show L_H unsolvable by **diagonalization**
- show L unsolvable by **reduction**

Reductions



Algo:
8

M' :
Simulate M on input x ;
Do <ACTION>;

Autumn 1999 1 of 13

We begin as always on the top of the pyramid. We look at what we are doing. So right now we are dividing the universe of problems into two very basic classes: Those that can be solved by algorithms and those that cannot. We have been developing a technique which allows us to divide the problems into unsolvable and solvable.

The technique has consisted of first finding the first unsolvable problem, which was the Halting problem, or the Halting language (L_H) when formalized. That we have done by using a fundamental proof technique called diagonalization. And then we have seen how to use another fundamental technique called reduction to show that all sorts of other problems are unsolvable or, formally, that all sorts of other languages are undecidable. We do that by reducing the first undecidable language L_H to the new language L , which we want to show undecidable.

We have seen one such reduction last time, and we have also seen that that reduction can easily be adapted to show all sorts of other problems undecidable. I am representing the reductions by a box diagram, by an ideogram: I am showing in green an algorithm for solving the Halting problem, or a Turing machine that decides L_H , which has all the details provided (in black) except one thing (shown in red): A Turing machine M_L that decides L , the language that is being shown undecidable.

The whole argument is by contradiction: If someone can provide M_L , then

the whole thing is an algorithm that decides L_H – a solution for the Halting problem. Since we know that a solution for the Halting problem cannot exist, then M_L cannot exist either, so language L is also undecidable.

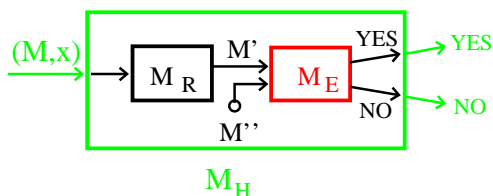
The heart of the reduction is the machine M_R which given M and x – an instance of the Halting problem – creates M' which is an instance of L , the new problem being shown undecidable. M' is a program, a Turing machine code, which will simulate M on input x and then do some action – whatever that action is.

In our reduction last lecture the action was 'write a dollar sign'. And remember that we must secure that this action does not take place in the first line of the M' -program – during the simulation of M on x – otherwise the whole thing won't work. So if we can secure that this action happens only in the second statement of M' , and there only if and only if M halts on x , then we have that M' will do 'action' if and only if M halts on x . Therefore deciding whether M' will do this action is equivalent to deciding whether M halts on x .

IN210 – lecture 4

**Example**

Theorem 1 *Equivalence of programs (Turing machines) is undecidable.*

Proof: **M' :**

Simulate M on input x ;
Accept;

 M'' :

Accept;

- M'' accepts all inputs.
- M and x are constants to M' .
- M' accepts all inputs if and only if M halts on input x .

Autumn 1999

2 of 13

Here is another example. I am just showing how this M' can be modified to show all sorts of problems undecidable. So imagine that I am teaching an algorithm class and I have given an assignment to my students to write sorting algorithms.

I have this idea: I am going to create a correct sorting algorithm myself, and then I will create this wonderful program (machine) M_E which tests equivalence of programs. Which means that M_E is going to say 'Yes' if the two input programs, e.g. two Pascal programs, are equivalent, meaning that they do exactly the same thing on the same input.

If I can have that machine M_E then I can test a student's program by comparing it to my correct sorting program: I give the student's program and my correct sorting program as input to M_E , and if M_E says 'Yes' then I know that the student program is correct, and vice versa.

So it is a nice plan, however it cannot work, because this tester of equivalence of programs, M_E , cannot possibly exist. The proof is here on this foil. I am not giving you the whole proof. I am just giving you the heart of the matter, the reduction.

M_R is as usual going to take M and x as input and produce M' . M' is going to simulate M on x as usual and then accept. It will accept its input whatever the input happens to be. So M' is not even looking at its input. As explained last lecture M' is going to write essentially both M and x on its tape, then run the universal Turing machine – simulate M on x – and then it is going to accept its unseen input. And that will happen if and only if M halts on x . So M' is going

to accept whatever its input is if and only if M halts on x .

I have also this M'' which is a part of my reduction. M'' basically does nothing, it just accepts. So M'' will accept always.

Now remember that M and x are constants in M' . So M' is going to accept always its input if and only if M halts on x . Therefore the two machines M' and M'' are going to be equivalent if and only if M halts on x . So if M_E can test whether two input machines are equivalent, this whole thing shown in green solves the Halting problem – it tests whether M halts on x . And since we know that the Halting problem is unsolvable, then M_E cannot possibly exist. So equivalence of programs (Turing machines) is undecidable.

It is easy to see how we can show that all sorts of questions which have to do with Turing machine properties, or what a Turing machine is doing, are undecidable.

IN210 – lecture 4



Insights

Theorem 2 (Rice; its basic message) *Most “interesting” properties of programs (TMs) are undecidable.*

But what about the unsolvable problems that are not related to programs/TMs?

Today

- A technique (reduction) for proving unsolvability of non-TM problems
- Important insights — physiognomy of unsolvable problems

Question: How to come out of the world of TMs and into the world of general problems?

Answer: View the Turing machine computation as **pattern matching**.

Autumn 1999

3 of 13

At this point your intuition tells you that in fact most questions that are related to properties of Turing machines (programs) are undecidable. I give you a program and ask: ‘What is this program doing?’ You won’t be able to design algorithms for answering such questions. That’s the basic insight.

There is a theorem which captures this insight, which makes it formal. It is called Rice’s Theorem. We are not going to cover it in this class, but its basic message is that most of the interesting properties of programs or Turing machines are undecidable. And it is easy to see how this basic insight follows from just the proof and the argumentation that I have just went through.

The basic question for today is: What about those problems that have nothing to do with Turing machines or with programs and their properties? How do we prove those unsolvable? We will see today a reduction technique which allows us to prove unsolvability of problems which apparently have nothing to do with Turing machine properties.

By going through this proof technique we are going to gain important insight. We gain intuition. We are going to see what typical unsolvable problems look like, so that when you meet one, you are going to recognize it immediately. And that is very important. Because then we won’t waste too much time trying to solve unsolvable problems. We will know how to avoid them, and when we meet them we will know how to prove that their are unsolvable and be done with them, as opposed to trying to solve them.

We are going to see a reduction from L_H to a new class of languages which have nothing to do with Turing machines. The basic question is how to come

out of this world of Turing machines and programs, and go out into the world where general problems live. So how do we transform something which is fundamentally a question regarding Turing machines into more general questions?

The answer is a general technique which we are going to see over and over again. It is another fundamental technique. The technique has to do with viewing the Turing machine computation as essentially *pattern matching*.

IN210 – lecture 4

time (steps)

configuration (tape, state, r/w head)

Turing machine rules (δ) become **templates**:

$\delta(s, 0) = (q_1, b, R)$ is and

but also and and and

for all X, Y, Z and $W \in \{1, 0, b\}$.

We also have for all $X, Y, Z \in \{1, 0, b\}$.

Autumn 1999 4 of 13

We are going to represent the Turing machine computation in a certain way which will allow us to step out of the world of Turing machines into something else. And that something else will essentially for the time being be something like pattern matching.

We are creating another big matrix. It is an infinite matrix where the rows represent configurations of a Turing machine, and where the vertical dimension represents time. Time, or number steps during an execution of the Turing Machine, grows upwards.

How do we represent configurations? Here on this foil I am showing a computation of our favorite Turing machine which will accept only the string '010'. As usual we see the blank squares and the input '010'. There is only difference, and that is that I am encoding the state and the position of the read/write head by using a double character: I am putting together the state and the symbol scanned ($\overset{s}{0}$), and that allows me to both record the state and position of the read/write head. The rest is just straightforward.

So this little idea gives me a complete encoding of a configuration. In the matrix on the foil I have in the bottom row the initial configuration where the input is written on the tape. The machine is then in the initial state and it is scanning the first character of the input. And then the machine is going to replace the first the character of the input with a blank, change its state to q_1 and move its read/write head to the right, scanning the next character. This is shown in the second row of the matrix. Then it will essentially do the same again and again, checking whether the input is '010'. And if the input is '010' the machine is going to halt and say 'Yes'. That's what we have seen. The matrix on the foil is the full computation.

So we can represent the computation by using this matrix. And then we can represent the rules of the Turing machine – the delta function, its program – by a bunch of templates. These templates are of the following kind: If we have the rule $d(s, 0) = (q_1, b, R)$ then we create the templates shown on the foil.

A template is a pattern consisting of four things which I can hold in my hand and then place where it fits. It is a tile. Now imagine that I don't have the second row in the matrix. I only have the first row which is the initial configuration, and which is well defined because I know the input. Then I can do the following:

I have a bunch of templates which represent my Turing machine program, and an unlimited supply of each template. I try to match the templates against every position in the matrix. For example the blue shaded template

	b	
b	s	1

matches the substring $b\overset{s}{0}1$ on the first row. When I find a matching template, then I know what to put in the middle square on the second row. In this case I put a blank.

Then I look at the next three characters which is $\overset{s}{0}10$. Again I try to find the right template among these templates, and I find

	q_1	
s	1	0

So I know that I have to write $\overset{q_1}{1}$ in the square on the second row just above the '1' on the first row. Then I look at the next 3-tuple of squares on first row, and the next one, and the next one, and so on.

If I have a set of templates which give me the rules of the Turing machine, then I am able to produce a new configuration given the preceding configuration. I am able to produce the rows of this matrix one by one, given the templates.

The natural question is: How do I create the templates? It is not so difficult. If the rule of the Turing machine is $d(s, 0) = (q_1, b, R)$, then I have to produce

the two templates

	b	
b	s	1

	q_1	
s	1	0

The first template is saying what happens with this 0 that is being scanned: It is being turned into a blank. And then the second rule tells me that I have to move the read/write head to the right. These two templates together tell me basically all about the rule.

But the two templates are made for this specific situation where I have a '1' to the right of $\overset{s}{0}$ and a blank to the left of $\overset{s}{0}$. But I can have any sort of character on the left side and any sort of character on the right side, and this rule would still apply. So I have to produce also these two templates for all combinations of input characters. Since the alphabet is finite, the number of templates produced is also finite. It is the same for this second template. I have to create all combinations, but again there is a finite number of them.

I also have to produce templates which allow me to copy the characters where the read/write head doesn't reach and the rules don't apply. In that case I am copying up the character in the middle position. For example if I have the 010 then I copy the 1 up. So for all possible combinations of characters I create such a template, but again there is just a finite number of them.

So it is easy to see how I can turn a Turing machine into a bunch of templates, and how I can turn a computation of a Turing machine into this kind of matrix. I have effectively turned the machine into a template matching situation, and now I can ask questions about this template matching situation which will correspond to asking about what the Turing machine is doing – and in particular whether it halts or not.

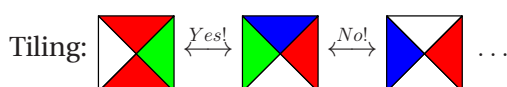
IN210 – lecture 4

Halting configuration

- row with square $\begin{matrix} h \\ \alpha \end{matrix}$
- finitely many rows (matrix is bounded in vertical direction)

An unsolvable Tiling problem

Input: A finite set of tiles with one designated tile (which must be placed by the entrance door). The tiles cannot be rotated or flipped.



Question: Is the set of tiles **complete**? (Given an unlimited supply of each tile, can any room be tiled?)

The Tiling problem (L_T) is unsolvable because we can make the **reduction**

$$L_H^C \mapsto L_T$$

The halting of the Turing machine translates into the situation in the matrix where the halting configuration is produced, namely where we have a row in the matrix which contains h as a state. We can always make this row canonic: We can say that all machines erase the tape before they halt in state h or produce the answer 'Yes', and so on.

So the question whether there is a row containing state h is equivalent to the question whether the Turing machine halts. Given a bunch of templates like this, we can ask whether this kind of row ultimately will be produced by starting from the input row. And that is the Halting question, translated into a pattern matching question. Equivalently we can ask whether there are finitely many rows in the matrix or infinitely many. If there are infinitely many rows in the matrix that is produced by this set of templates, that means that the corresponding Turing machine does not halt. It just runs forever. If there are finitely many rows, that means that the Turing machine halts.

Now you are ready to see a problem which on the surface has nothing to do with Turing machines, but in reality of course it is just the Halting problem in disguise. I am showing you a simple unsolvable pattern-matching problem, a tiling problem in this case.

Imagine that you are given a set of tiles like those on the foil. There is a finite number of different tiles, and each tile has a color pattern. What do we do with tiles? Of course, we tile floors and walls. We put them next to each other. But we put them together in such a way that the adjacent surfaces – the triangles – have matching colors, so that they together make a square.

Imagine also, for some reason, that these tiles cannot be rotated, they cannot be turned. They just have to be as they are. The last constraint is that we have one designated tile which must be placed, say, by the entrance door – there is one tile which has to be there in the first row of tiles.

So we have a finite set of tiles (but an infinite supply of each tile type) which cannot be rotated or flipped, and we also have one designated "entrance" tile. The question that we ask about this tiling system is whether this set of tiles is *complete* – whether it can tile any room, regardless of its shape and size.

Apparently this question has nothing to do with Turing machines, but we will see that this is exactly the Halting problem in fact, disguised a little bit. We see this by showing the reduction from the Halting problem to this tiling question. The reduction will basically follow the same line as the creation of the template matching problem and templates from a given Turing machine.

IN210 – lecture 4

- Tiles for rules of Turing machine M :

 for $\delta(s, 0) = (q_1, b, R)$
- Tiles for tape symbols “far away” from read/write head:

 , etc.
- Designated tile:
- Tiles for input string x :

 , etc.
- We can tile forever (tile any room) if and only if M doesn't halt on input x !

Autumn 1999 6 of 13

To illustrate the reduction I am using exactly the same Turing machine as before. I am only showing the idea – how to construct tiles given a delta function – and you will see that these tiles they are really just the templates we have just seen. For example the rule $d(s, 0) = (q_1, b, R)$ is represented by two tiles. The first tile is saying: "If you are in state s and you are scanning a 0, then move your head to your right and enter state q_1 ." The right side of the tile says " q_1 ". I am using the superscript "" to link this tile with another tile because we need two tiles to represent a rule.

And the other tile is going to say: "If you are coming to the right from this situation which we labeled q_1 then by all means enter q_1 and copy this 1. Everything is fine."

So these two templates, which must be positioned side by side, they represent basically what we have just seen, and again we will have to create all sorts of other tiles to cover all possible situations to which this rule applies. But there is only a finite number of different tiles, so we make them.

And then we have a designated tile which corresponds to this particular square on the tape where the first character of the input is shown. We also use that tile to show that the machine is in state s , the initial state, when the computation starts.

Finally, we need a way to distinguish the first row of tiles from the other rows so that we are able to encode the input in the first row. Here we do this by using tiles with special symbols (I_0 , I_1 , etc) written on the left and right side, so that there is only one tile that matches on the left side and only one tile that matches

on the right side. This little trick, which ensures that there is only one way to tile the first row, allows us to encode the input easily.

This is basically the end of the story. I am giving you a finite set of tiles, say, 25 of them exactly. I give you an unlimited supply of each of the 25 different tiles and a designated "starting tile". Then I tell you: "Look, you are allowed to use your entire lifetime – there are no restriction on how much thinking you can do and how much programming – but your task is to come up with a program which is going to check whether or not this set of tiles is complete, whether we can use them to tile any room."

Can you make such a program which will decide whether any finite set of tiles with one designated "starting tile" is complete? The answer is "No, it is not possible." Because if you could construct such a program, then we could use it to solve the complement of the Halting problem, which we know is impossible.

This looks kind of strange: Given this bunch of tiles – which you can place in your hands and put into your pocket – there is no program that can check whether this finite set of tiles is complete, whether it can tile any room.

IN210 – lecture 4



An unsolvable grammar (language definition) problem

Grammar $G = (T, N, R)$

T, set of terminal symbols

N, set of nonterminal symbols, containing the start symbol **S**

R, set of derivation rules:

$$\mathbf{S} \rightarrow (\mathbf{S}) \quad R_1$$

$$\mathbf{S} \rightarrow ()\mathbf{S} \quad R_2$$

$$\mathbf{S} \rightarrow \epsilon \quad R_3$$

A **derivation** of the string $((()))$:

$$\mathbf{S} \stackrel{R_1}{\underset{G}{\vdash}} (\mathbf{S}) \stackrel{R_2}{\underset{G}{\vdash}} (()\mathbf{S}) \stackrel{R_1}{\underset{G}{\vdash}} (()(\mathbf{S})) \stackrel{R_3}{\underset{G}{\vdash}} ((()))$$

The **language defined by G** is the set of all strings that are derived by G .

In **context-free grammars** the left-hand side of each rule consists of exactly one non-terminal.

I will now show you an unsolvable grammar problem. I will briefly tell you about grammars – what they are and what they serve for. Formally they are ways of defining languages. So a grammar is a way to define a spoken language, like English. But in our little world these grammars serve for defining formal languages, sets of strings, which is kind of the same thing. Because a grammar in the real world will define what words and what sentences are allowed in a certain language – what can be said and what cannot be said.

In this more general and abstract context of formal languages, we define strings and sets of strings. And then we can use the same idea to define the strings in any kind of context. In particular we can define what programs are valid in a certain programming language. Programming languages are commonly defined by grammars, and they are defined by what is called context-free grammars.

Let us first see what these grammars are and how they define languages and strings. Formally a grammar is a triple. It consists of three things: A finite set of terminal symbols T , a finite set of non-terminal symbols N , and a finite set of derivation rules R .

Here on this foil we are writing terminal symbols in small letters and non-terminal symbols in bold capitals. There is one specific designated non-terminal symbol called **S** – the start symbol. And then the derivation rules will tell us how to derive strings. In our example derivation rule 1 is saying: If you see an **S** you can replace it by the string "**(S)**". Derivation rule 2 is saying that you can replace an **S** by the string "**()S**". And finally rule 3 is saying that you can erase an **S** – this

ϵ here means you replace it by nothing.

These 3 rules give us a way of defining all strings which consist of balanced parentheses. On the foil I am showing you a derivation of the particular string "(())" by using the rules of the grammar. Every derivation must start from the start symbol **S**, and then via successive application of the rules – replacing non-terminal symbols by strings consisting of non-terminal and terminal symbols – we derive in the end a string consisting of terminal symbols only.

So in this case we first use rule R_1 , then R_2 , R_1 again and finally R_3 . The whole sequence is a derivation of the string "(())". And if this derivation exists then we say that the string "(())" is derived by the grammar, or that the string belongs to the language defined by the grammar. The language defined by the grammar is of course the set of all strings that can be derived by the grammar.

We talk about several kinds of grammars. Context-free grammars define a limited family of languages. They are such that on the left-hand side of a rule, only non-terminal symbols can appear. So a rule of a context-free grammar says: Whenever you see the non-terminal **S** you can replace it by the string on the right-hand side, and whenever you see an **A** you can replace it by this other string, and so on.

IN210 – lecture 4

**Example**

$$G = (T, N, R)$$

$$T = \{\text{Mark, Ann, I, love, you, very, much}\}$$

$$N = \{\mathbf{S}, \mathbf{A}, \mathbf{B}\}$$

$$R = \begin{array}{l} \mathbf{S} \rightarrow \text{Mark } \mathbf{A} \text{ Ann} \\ \mathbf{A} \rightarrow \text{I love you } \mathbf{B} \text{ much} \\ \mathbf{B} \rightarrow \text{very } \mathbf{B} \\ \mathbf{B} \rightarrow \epsilon \end{array}$$

As an example I give you a little grammar, and you can try to guess what sort of strings the grammar can produce. The grammar consists of terminals, non-terminals and rules. Terminal symbols are 'Mark', 'Ann', 'I', 'love', 'you', 'very' and 'much'. So these are the terminal symbols. **S**, **A** and **B** are the non-terminals. The rules say that **S** can be replaced by 'Mark **A** Ann'. **A** can be replaced by 'Love you **B** much'. And **B** can be replaced by 'very **B**' or it can be erased.

So it is easy to see that this grammar is producing a set of rather boring and dull love letters of the following kind:

"Mark
I love you very very very very very much
Ann"

(This is a blank page)

IN210 – lecture 4



General grammars

- The left-hand side of each rule can have both non-terminals (at least one) and terminals
- A Turing machine M with input x can easily be encoded as derivation rules:

— TM rule $\delta(S, 0) = (q_1, b, R)$:

$$\begin{array}{|c|} \hline s \\ \hline 0 \\ \hline \end{array} 1 \rightarrow b \begin{array}{|c|} \hline q_1 \\ \hline 1 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline s \\ \hline 0 \\ \hline \end{array} 0 \rightarrow b \begin{array}{|c|} \hline q_1 \\ \hline 0 \\ \hline \end{array}$$

etc. (“box symbols” are non-terminals)

— Derivation rule for input string ('010'):

$$\mathbf{S} \rightarrow \mathbf{B} \begin{array}{|c|} \hline s \\ \hline 0 \\ \hline \end{array} 1 0 \mathbf{E}$$

— Rules for inserting and removing blanks

— Rule for removing the last non-terminal:

$$\begin{array}{|c|} \hline h \\ \hline Y \\ \hline \end{array} \rightarrow h$$

- Reduction $L_H \mapsto L_G$:

M halts for input x if and only if the string “h” can be derived by grammar G .

Question: Programming language syntax is defined by context-free grammars, and not by the more powerful general grammars. Why?

Autumn 1999

9 of 13

The point is that there are certain things which can be expressed by these grammars and certain things which cannot. There are limitations to how much you can say using a context-free grammar. A more general grammar would allow us to get around these limitations by allowing us to put the non-terminal symbols on the left-hand side into a context and say: This non-terminal symbol in this particular context of terminal symbols can be replaced by this other string of symbols. So essentially this string can be replaced by this other string.

The basic message that I am conveying here is that a general grammar can be used to encode the computation of a Turing machine in exactly the same way in which we have been using the patterns for the same purpose. And that makes an essential difference between general grammars and context-free grammars.

I am giving a hint about how you can use a general grammar to encode the code of a Turing machine, its program. Again we are using that same Turing machine we have seen before. The first derivation rule is saying: If you happen to see this non-terminal $\begin{array}{|c|} \hline s \\ \hline 0 \\ \hline \end{array}$ followed by a '1', then replace the box symbol by a blank and the '1' by the non-terminal $\begin{array}{|c|} \hline q_1 \\ \hline 1 \\ \hline \end{array}$. This derivation rule, together with its twin brothers which have a '0' or 'b' instead of the '1', corresponds to the Turing machine rule $d(s, 0) = (q_1, b, R)$.

I also make a derivation rule for the input string: $\mathbf{S} \rightarrow \mathbf{B} \begin{array}{|c|} \hline s \\ \hline 0 \\ \hline \end{array} 1 0 \mathbf{E}$.

If I begin with an \mathbf{S} this general grammar will essentially do what my templates were doing before: It will tell me how to produce each row of the matrix

starting from the preceding row.

You can see that this is exactly the same thing as the one we have seen before. Here we are creating strings by applying the rules. Before we were creating rows of the matrix by applying the templates. But the templates and the rules they are similar things, they can represent one another so the two situations are equivalent. They can easily be translated into one another.

The basic message is that deciding whether or not a certain string is derivable by a certain general grammar is unsolvable by algorithms. Why? Because if I can decide whether or not a certain grammar derives a certain string, then I can decide whether this grammar derives the string "h", which says 'halt'. So if the string "h" can be generated by the grammar, that means that the corresponding Turing machine reaches the halting configuration beginning from this particular input, which is the answer of the Halting problem.

It is easy to see that this transformation of a Turing machine – its rules and input – to a general grammar is something that can be done automatically. You can imagine writing a program that takes the rules of the Turing machine and its input and creates these rules of the grammar automatically. That program is the reduction.

So given M and its input x the program can generate the grammar such that the grammar will derive the string "h" that corresponds to the halting configuration of the machine, if and only if the original Turing machine M halts on its input. So deciding whether or not a given grammar derives a given string is equivalent to deciding whether or not a given Turing machine halts on a given input. Both problems are undecidable.

And then we have this practical question: Programming language syntax is usually defined by context-free grammars, and never by general grammars. Why is that? Well, as you will see in some later classes (IN211 and IN310) it is usual to define a programming language by using a context-free grammar, and then this context-free grammar is used to produce a compiler. What a compiler will do as the first thing is: It will check the syntax. Before it tries to translate the program into machine code, it will check whether or not what you have written is really a program or not. And you are familiar with this already because when you try to write Java or C code, then if you make some syntactic errors, then the compiler is going to complain and say: "Look, you forgot a comma there." Or: "I don't know what you are doing, it's just some kind of nonsense." And then you look at the line and you see that you have done some mistake.

If the syntax is correct – if your program is a correct program – then the compiler goes on to actually interpret the program as machine instructions by using semantic rules, by trying to give meaning to the sentences. But by and large, when the syntax is wrong there is no meaning, and the compiler will complain right away.

So the first thing the compiler does is: It checks whether or not your program is properly written, whether it is a valid program. The point is that if your grammar can only be written as a general grammar, then no compiler would be able to check whether or not your program is a valid program, because such a compiler could also be used to solve the Halting problem. So it is completely essential that programming languages can be defined by context-free grammars, because then they can have compilers.

IN210 – lecture 4

**Can machines think?****Example: Theorem proving in a formal system**

$$\begin{aligned} (x + y)^2 &\equiv x^2 + 2xy + y^2 && \text{(Theorem)} \\ (a + b)(c + d) &\equiv a(c + d) + b(c + d) \\ &\vdots && \text{(Rules/axioms)} \\ ab &\equiv ba \end{aligned}$$

Question: Can algorithms prove/verify theorems?

Answer: Not if the rules can encode a Turing machine . . . But algorithms can **accept** theorems.

Algorithms can also **enumerate** theorems.

Example: Theoremhood in first-order logic is undecidable (IN 394)

Question: What about automatic program correctness proving?

Autumn 1999

10 of 13

We continue to use the little technique that we have developed, in order to explore the limits of what can be done by machines. We now turn to the more general question: "Can machines think? What are the limitations to using computers to solve human problems?"

One common problem that requires some intelligence is proving theorems. In mathematics you have some kind of mathematical expression on the left-hand side of the equation and another one at the right-hand side. This is basically a theorem: Are these two expressions identical? Is it always true that $(x + y)^2$ is equal to $x^2 + 2xy + y^2$?

What is usual in mathematics is to use some kind of rules called axioms to transform the left-hand side of the equation into the right-hand side, which amounts to proving that these two are equivalent. So these rules are such that they always turn something into something that is equivalent to it.

For example, we have a rule that says that $(a + b)(c + d)$ is equivalent to $a(c + d) + b(c + d)$ And we have the commutativity rule which says that $a \cdot b$ is equivalent to $b \cdot a$. Things like that, where a, b, c and d are algebraic terms.

So these rules tell us how we can transform strings into other strings, and then again you are beginning to recognize the situation. The axioms are basically some kind of patterns which tell us how to turn one string into another, and thereby derive different strings. And we can again see the similarity between this and our original template matching. We are again being asked whether starting from a certain string and applying certain rules or axioms or templates, whether we can derive a certain other string.

What does this situation remind us of? It most reminds us of our original matrix and templates question and of the Halting problem. These questions are completely similar, in fact they are like twin brothers. If you just use abstraction

a little bit and ignore the fact that here we are still talking about the alphabet and the configuration and things, then you see that we are just giving some kinds of rules which tell us how we can turn one string into another string and that one into another string and that one into another string, and so on.

So by and large here we are dealing with exactly the same situation. Without going into details we can answer this question "Can algorithms prove or verify theorems?" by saying "No" if the rules we are talking about are rich enough – or powerful enough – to encode a Turing machine.

As an example imagine that you are given a formal system, something like arithmetic, and you are asked whether or not a program can prove theorems in that arithmetic. Arithmetic is a pretty rich kind of formal expression, so your intuition tells you that this is probably undecidable. But to prove it you would naturally encode the rules and input of a Turing machine as the axioms of the formal system. And if you can do that, then in a very short time you have a proof that programs or machines cannot prove all theorems in that formal system.

This has been done. For example there is a well-known result which says that theoremhood in the first order logic is undecidable. We speak more about this in a graduate version of this class, IN394, which is taught usually in Spring.

First order logic is one of the very, very basic formalizations of mathematical or scientific thinking. So this result amounts to saying that what is true and what is false in a kind of a scientific or mathematical sense, cannot really be proven by algorithms. This theorem, when properly interpreted, gives a very fundamental limitation on what machines can and cannot do.

We can notice however that machines can enumerate theorems. And algorithms can also accept theorems. Given this as a hypothetical theorem, you can easily come up with an algorithm which is going to say 'Yes' if this is a theorem. How? By trying all possible derivations.

In any kind of situations a number of rules can be applied to the string, and because we are talking about finite axiomatizations, we have only a finite number of rules. So you apply all of these rules, one by one, to this situation and see what are the possible next strings. And then to each one of these next strings you apply all possible rules, and so on. This may take a long, long time to do. But a finite situation always results in finitely many possible next situations, and then each of them in finitely many next situations.

So that if something is a theorem, the proof will be constructed in finitely many steps, but we don't know how many. If something is not a theorem the proof will never be constructed, so this machine will just run forever and ever. In this concrete, practical situation we again see the meaning of the fact that we have to kinds of unsolvability, and that Theoremhood will typically be of about the same difficulty as the Halting problem, namely it is something that can be accepted but not decided.

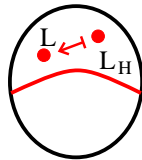
A very popular problem in computer science practice is proving program correctness automatically. It is a step beyond the usual compilation where you are just checking the syntax and translating the program into something. Proving program correctness is a very, very big issue. Why? Because more and more essential things depend on computer programs: banking, defence systems, nuclear power plants, etc. And if we can have a real proof that the program is correct, then a lot of both practical and legal issues are resolved, even kinds of life-and-death issues in some situations.

So we would very much like to have programs proving other programs correct. Obviously there are limits to how much can be done. The insights that we have reached should give us a rather clear idea of where those limits lie. Remember, even deciding whether a general program merely stops or not for a given input, is undecidable.

IN210 – lecture 4



Closing remarks on unsolvability



Two kinds of reductions:

- to Turing machine questions
- to general questions

The reductions give us a **tool** for proving undecidability and **insight** into the nature/physiognomy of unsolvable problems.

Autumn 1999

11 of 13

I finish the story about unsolvability by putting what I have said together into a little diagram, an "egg". What unsolvability is about, it is about cutting the egg in two halves, or dividing the universe of all problems into two subclasses – the unsolvable problems and the solvable problems.

We have seen two kinds of reductions for proving problems unsolvable after we had used a technique called diagonalization to prove our first problem, the Halting problem L_H , unsolvable. The first kind was applicable to Turing machines questions. The second kind to more general questions, usually some kind of pattern matching, theorem proving sort of questions.

These techniques have on the one hand given us tools for proving for concrete problems that they are unsolvable. And at the same time they have given us essential insights so that we can recognize unsolvable problems when we see them. Related to Turing machines or programs the insight we have gained is that just about any properties of Turing machines or programs are undecidable. And a similar insight has been gained about these general questions of pattern matching problems or theorem proving problems. Most such problems will be undecidable. In fact, provided that the systems in which we are coding the questions or axioms are rich enough to encode the Turing machine, the questions will usually be undecidable, and so on.

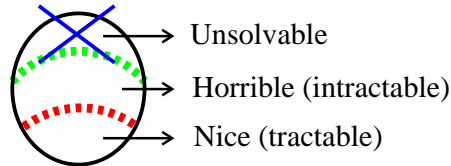
So now we have both an idea of what sort of problems are undecidable and the techniques for proving that. We are about to leave the the unsolvable problems all together and focus only on the solvable problems.

(This is a blank page)

IN210 – lecture 4



Complexity



- Horrible problems are solvable by algorithms that take billions of years to produce a solution.
- Nice problems are solvable by “proper” algorithms.
- We want **techniques** and **insights**

Complexity \longleftrightarrow **resources**: time, space

\updownarrow
complexity classes:
 P(olynomial time), NP-complete,
 Co-NP-complete, Exponential time,
 PSPACE, . . .

Autumn 1999

12 of 13

We are closing the first part of the material on complexity and algorithm theory that we are covering. We are now entering into the second half which is going to be a major part of the class.

We are again on the top of the pyramid, looking at our work: What is it that we have been doing? We have just eliminated this part, the unsolvable problems. We have thrown them out.

We are now looking at the solvable problems, and we are about to work on the second borderline here, shown in red. We want to divide the horrible or hopeless problems from the nice ones. Meaning: Divide the problems that are properly solvable by algorithms from the ones that are solvable by algorithms in principle, but those algorithms take billions of years to compute. So they are kind of useless.

Again what we want is both techniques and insights. We want to be able to recognize the hopeless problems and the nice problems – tell them from each other when we see them in practice. We also want the techniques which will allow us to prove that a certain problem is hopeless or that a certain problem is nice. So both of these are issues dealt with in complexity theory.

Complexity intuitively has to do with the question how complex a certain problem is, and the complexity is measured in terms of some resources that are used in computation – typically time, but also space. So roughly we are asking: How much time does an algorithm need to solve a certain problem? Or how much space does it take?

And depending on how much space or time a certain problem requires from

the best possible algorithm (usually in the worst possible case for the algorithm, i.e. when it has the most annoying input), the problems will be classified into complexity classes. Our goal will be now to subdivide the universe of all solvable problems into classes where problems of similar complexity live.

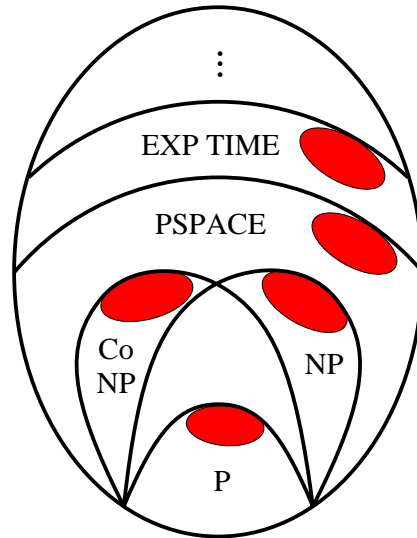
When we can place a problem on this map, we know two things about the problem. One: How difficult the problem is, how complex it is, how much time or space its solution will require. And two: We know something about the nature of that problem. We know its basic physiognomy. And we know what sort of algorithmic approaches would be appropriate for solving that problem.

So studying the complexity of problems will in fact give us a lot. It will give us a foundation for understanding problems, both their nature and the ways of solving them.


IN210 – lecture 4



Goal



Map of classes

 = complete or "hardest" problems in a class

Autumn 1999

13 of 13

This is what our map of solvable problems will ultimately look like. It will consist of classes which contain one another in a certain way, or partially contain one another. Beginning with \mathcal{P} – the problems solvable in polynomial time – we have a larger class called \mathcal{NP} and another one called $\text{Co-}\mathcal{NP}$, then we have polynomial space, exponential time, and so on.

Typically in these classes we will be interested in finding problems which represent the class in a certain way, problems which are in a certain sense hardest for that class. Those problems we call the complete problems. So we will be studying \mathcal{NP} -complete problems, \mathcal{P} -complete problems, $\text{Co-}\mathcal{NP}$ -complete problems, PSPACE -complete problems and so on.

When we say that a problem is complete for the class exponential-time, we intuitively mean that the problem is as hard as any other problem in this class. So that if you can solve that problem efficiently, then you can solve all the problems in that class efficiently. In that sense those problems are the hardest problems in the class. And being the hardest they represent the class in a certain way, because you want to know what sort of hard problems are there in a class. Well, those are the complete problems. So one of our goals will be coming up with complete problems for the class.

And beginning next time we will start studying the complexity theory formally. We are going to formulate certain measures, certain ideas, certain concepts which will allow us to come up with these classes in a formal way. We will also learn formal proof techniques that will allow us ultimately to place the interesting problems that exist in this world – the interesting languages – onto the

map of classes. Then we will ultimately understand how difficult they are in a very precise sense.

For a while we will be studying the complexity classes and the techniques for proving \mathcal{NP} -completeness and so on, and then you will be able to use your G&J textbook because that is the part which G&J cover.

After that we will be studying systematically the techniques which will allow us in fact to deal with these horrible problems in different ways. So we are not going to give up as we did in the case of unsolvability. We are going to see all sorts of techniques which allow us to deal with the hard problems. Because they exist, they are important to solve in practice.

Our basic scheme will be this: By showing that a certain problem is nice we have produced a good solution for the problem. Our work is done. By showing that a certain problem is horrible, we know that we should use the same techniques that we used for the nice problems. We are not going to try to produce a kind of a worst-case, efficient solution. But we are going to use other techniques – and we are going to study those techniques systematically – that will still allow us resolve these horrible problems. When we look at the horrible problems in more detail, will see that depending on which of these classes they belong to, different techniques will apply for turning these horrible problems into something which is not hopeless, something which is quite tame and nice.

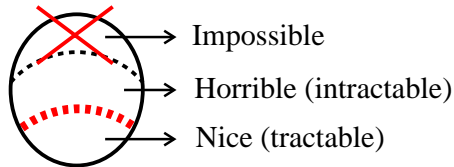
So roughly speaking that will be our agenda for the rest of the semester: Classifying problems first of all and then studying the algorithmic techniques which apply in different situations – gaining insights and developing techniques.

3.5 Lecture 5

IN210 – lecture 5



Complexity



Intractable, best algorithms are infeasible

Tractable, solved by feasible algorithms

Problems Complexity classes

Horrible \rightsquigarrow \mathcal{NP} -complete, \mathcal{NP} -hard,
PSPACE-complete,
EXP-complete, ...

Nice \rightsquigarrow \mathcal{P} (Polynomial time)

Goal of complexity theory

Organize problems into **complexity classes**.

- Put problems of a similar complexity into the same class.
- Complexity reveals what approaches to solution should be taken.

Complexity theory will give us an organized view of both problems and algorithms.

Autumn 1999

1 of 12

G&J:
2.0-2.5

We are looking at all problems in the world and dividing them into classes. We have just finished with one large class – the largest of all in size – which is the class of all problems that cannot be solved by algorithms. Now we move those out of the way, label them as hopeless or impossible, and we deal with the others.

We are at the highest level of abstraction dealing with the big picture. And our big picture is to begin with that we divide those remaining, solvable problems into two large categories: Nice ones that can be properly speaking solved with nice algorithms, and the horrible ones which don't seem to have good algorithms.

The horrible problems can be solved by algorithms, but those algorithms take an unbelievable amount of time to compute the answer – zillions of years. Nobody has that kind of time to wait. However those problems can be treated by other methods, and a significant part of our time will be devoted to actually studying the methods by which those problems can be treated. So that is why I call them horrible instead of hopeless.

We use abstraction and formalization in order to arrive from these intuitive, vague notions of horrible and nice to well-defined things – mathematical, theoretical concepts – which we can use to build up theories and arguments and prove things. We will finally arrive at the formal concept of complexity classes. We will see that this notion of 'horrible' gives rise to concepts such as \mathcal{NP} -

complete, \mathcal{NP} -hard, PSPACE-complete, Exponential time-complete, and so on. These are classes of problems. And then 'nice' will give rise to \mathcal{P} and others such as \mathcal{NC} . As is usual in complexity theory we will also replace this somewhat emotional notion 'horrible' with 'intractable' and the notion 'nice' by 'tractable'.

Why do we organize problems into complexity classes? There are two large benefits: One is that we put problems of a similar complexity together, into the same drawer. And then when we open the drawer and take out a problem which has a certain label – the label of that complexity class – we know a lot about that problem. We know how difficult that problem is to solve, and we know something about the nature or the physiognomy of that problem. Because those drawers they contain similar problems – not only of similar complexity, not only of similar difficulty – but also in a certain sense: similar.

And by virtue of that similarity, when we know the problem's complexity class we also know what sort of algorithmic approaches will be appropriate for the problem, because they are the algorithmic approaches appropriate for that complexity class.

In sum, by organizing the problems into complexity classes we learn both about the problems and about the solutions. We classify the problems or the formal languages. We also organize the solutions or the algorithms or the Turing machines or the algorithmic approaches. So we have both: By organizing the problems into complexity classes we have organized both problems and algorithms, and our whole world is nicely organized.

So one major result of this class will be organizing problems and languages into a map of classes and understanding what sort of creatures live in each area of that map and what sort of algorithmic approaches are appropriate for them.

IN210 – lecture 5



Time complexity and the class \mathcal{P}

We say that Turing machine M **recognizes language L in time $t(n)$** if given any $x \in \Sigma^*$ as input M halts after at most $t(|x|)$ steps scanning 'Y' or 'N' on its tape, scanning 'Y' if and only if $x \in L$.

($|x|$ is the input length – the number of TM tape squares containing the characters of x)

Note: We are measuring **worst-case** behavior of M , i.e. the number of steps used for the most “difficult” input.

We say that **language L has time complexity $t(n)$** and write $L \in \mathbf{TIME}(t(n))$ if there is a Turing machine M which recognizes L in time $\mathcal{O}(t(n))$.

Polynomial time $\mathcal{P} = \bigcup_k \mathbf{TIME}(n^k)$

Note: \mathcal{P} (as well as every other complexity class) is a class (a set) of formal languages.

Autumn 1999

2 of 12

Now we descend a little bit in our level of abstraction and we look at the theory. We want to construct a theory which will allow us to achieve our goal. You will see that this complexity theory is actually very similar to what we have already seen. So if you look completely abstractly – just at the shape of the theory, at its broad features – you will see that we are basically repeating the same story. The definitions will be slightly different, adjusted now to our goal of studying complexity, not undecidability. We will have to include the notion of resources and model the consumption of the resources by the algorithms, and that will give us basically the complexity theory. But the basic ideas will remain more or less the same. So this will be almost like a review of what we have seen before in undecidability.

We say that a Turing machine M recognizes language L in time $t(n)$ if given any string x from Sigma-star as input M halts after at most t of the length of x steps, saying 'Yes' or 'No' – saying 'Yes' if and only if x is in L . So this definition is completely straightforward.

The length of x is the length of the input, the number of characters in the input. So now we know how to measure time on a Turing machine. An important observation is that we are measuring worst-case performance, in the sense that we are measuring the time used by the Turing machine on the most annoying, worst possible input of length x .

Time is given as a function, and this function is associated with a language. So we say that the language L has time complexity $t(n)$ – and write that " L is in $\mathbf{TIME}(t(n))$ " – if there is a Turing machine M which recognizes L in time $\mathcal{O}(t(n))$.

So big-O is a major instrument of abstraction in complexity theory. I expect that you are all familiar with big-O from some other classes that you have taken, possibly IN115. Big-O says that we don't care about constants and such things in complexity theory. We only care about the shape of the curve, its slope. Big-O also means that we don't care about the time used for one particular input, but for the growth of time used as a function of the length of the input – in other words how fast the time usage grows when the input size grows.

This TIME thing is really a set of languages. Any function will give us a set, a complexity class – which is the class of all languages that can be recognized in that time.

We can now define a large class called 'Polynomial time' as the union of all complexity classes whose time function is a polynomial. So \mathcal{P} is simply a class of formal languages, or a set of formal languages, and this is how it is defined: \mathcal{P} is the set of all languages which can be recognized in some polynomial time. More formally we say that \mathcal{P} is the union of the TIME classes for all n^k .

IN210 – lecture 5



“Nice” or “tractable” $\rightsquigarrow \mathcal{P}$

Real time on a PC/Mac/Cray/Hypercube/... \rightsquigarrow Turing machine **time** (number of steps)

Computation Complexity Thesis

All **reasonable** computer models are **polynomial-time equivalent** (i.e. they can simulate each other in polynomial time).

Consequence: \mathcal{P} is **robust** (i.e. machine independent).

Worst-case complexity \rightsquigarrow Real-world difficulty

Feasible solution \rightsquigarrow Polynomial-time algorithm

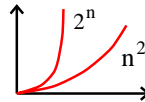
- $t(n) \rightsquigarrow \mathcal{O}(t(n))$

Argument: “for large-enough $n \dots$ ”

- $n^{100} \leq n^{\log n}$. Yes, but only for $n > 2^{100}$.

Argument: Functions like n^{100} or $n^{\log n}$ don't tend to arise in practice.

$n^2 \ll 2^n$ already for small or medium-sized inputs:



Autumn 1999

3 of 12

Now we quickly go back to the high level and ask the obvious question: Why is it that \mathcal{P} models the intuitive notions 'nice' or 'tractable'? Why are the solutions that are polynomial-time solutions good solutions? This is completely essential because after this argument is done, we will be able to basically solve a problem by a polynomial-time algorithm and then smile and say: "OK, this is it. We have accomplished a great feat. We have a good solution."

But the question to begin with is: Why is a polynomial-time solution a good one and non-polynomial time not good?

We have defined the complexity by using the number of steps used by the Turing machine. We measured the time used by the Turing machine in the worst case, for the most difficult input. And then we have put together a bunch of these TIME sets of languages into a big lump, called \mathcal{P} , and we call this \mathcal{P} 'nice'. So we have done three major steps of abstraction. Let's see why we are allowed to do that.

So step one: "Turing machine time – the number steps in a Turing machine – is supposed to represent the real time that we actually spend on a Mac or on a PC or on a Cray computer, which is this amazing pipe-lined fast machine, or on a Hypercube which is a parallel computer." We have all kinds of computers! Each kind completely different even in the way it functions.

How can it be that such a silly little thing like the number of steps on a Turing machine can represent the actual time on these machines? That is a serious question. This question is addressed by something which is called the Computational Complexity Thesis, which says that all "reasonable computer models

are polynomial-time equivalent". And this polynomial-time equivalent means that they can simulate each other in polynomial time. Now, this is something you want to think about, what this means. Another way of expressing the same fact is to say that \mathcal{P} is robust, that is: machine independent.

We need to understand this notion of polynomial time a little bit, and we will understand it better and better when we work with it, but the basic message here is that when we measure polynomial time on a Turing machine – and you imagined that one step on the Turing machine takes one unit of time, whatever that unit of time is – then that means that there is a polynomial-time algorithm for the same problem on anyone of these computers. And vice-versa: If there is a polynomial-time PC algorithm for a certain problem, then there is a polynomial-time algorithm for the Turing machine.

The fact that the real time can be measured in a certain sense by the number of steps in a Turing machine, comes from our notion of what we mean by time, and how accurately we measure things. We don't care about constants, so we don't care about multiplicative factors. We only care about the basic shape of the curves, the basic dependence, how fast something grows. And that is basically what the issue of polynomiality or exponentiality of the curves is about.

Polynomiality gives us manageable growth of complexity with the input size, Exponentiality gives us very fast growth, not manageable. So this is what we roughly want to distinguish: One kind of dependence between size of input and computation time: polynomial, nice. Versus another one: exponential, not so nice.

And it turns out that precisely this that we are trying to capture, is independent of the machine model. And this is preserved when we move from the number of steps on the Turing machine to real-time which we measure by a clock. This is something for you to think about.

Step number two: "The worst-case complexity of a problem models in a good way the difficulty observed by computer engineers who are solving the problem in real life on "real input". In this second part of the class we will stick to worst-case complexity, but as will become evident later, a strong case can be made for using some kind of average-case complexity instead, where we measure the time used on an "average input". The theoretical worst inputs might never arise in real life. This is a rather subtle point. Worst-case complexity has historically been the most important way to measure real-life complexity, possibly because the underlying theory is easier to work with.

Our last abstraction step was: "A polynomial-time algorithm is a feasible solution." Now, why is polynomial time feasible? The argument here is that for large inputs the shape of the curve becomes the decisive factor, whether that shape is polynomial or exponential. We basically don't care about small problem instances because they are manageable one way or the other. But if the instances are large, then it really matters how efficient our algorithm is. And at that point the argument says: Whether the algorithm is polynomial or exponential makes a huge difference.

You might say: "Look, n^{100} is a polynomial and $n^{\log(n)}$ is what we call an exponential function, but this exponential function will be larger than this polynomial only when n is bigger than 2^{100} , which is like the number of molecules in the universe or something like that – an unbelievably huge number. So this really doesn't make sense."

The counter argument that takes care of this problem is that functions like n^{100} and $n^{\log(n)}$ never or very rarely arise in practice. What we have in practice is these kind of creatures: n^2 or 2^n . And then it is easy to see already for small values of n , that n^2 is actually much, much smaller than 2^n . So the polynomials that occur in practice they are nice ones, and exponential functions that occur

in practice they are terrible ones. So that this distinction between polynomiality and exponentiality is actually in practice a sharp one, a meaningful one.

In undecidability theory Church-Turing Thesis said that a Turing machine can do whatever any algorithm, program or machine can do. The Computational Complexity Thesis is a stronger claim. It says that a Turing machine can compute in polynomial time whatever any algorithm, program or machine can compute in polynomial time.

The Computational Complexity Thesis cannot be proven because we are talking about informal things. But a thesis can be substantiated never the less, and that is what modeling is quite a bit about. It is about relating the formal world to the informal world. So how do we substantiate the fact that all reasonable computer models are polynomial-time related? Or that they can simulate each other in polynomial time? We do it by demonstrating sufficiently many cases, and gaining insights so that we see that it is really so.

And that is something that you will have to do on your own. There is a fair amount of insight in your textbooks. I will just help you out by saying that given two computers – say a Turing machine and a PC – then you can show that a PC can simulate a Turing machine in polynomial time, and you can show that a Turing machine can simulate a PC in polynomial time. Where time is being measured by number of steps, or even real time.

By showing that 1 PC-instruction can be done in polynomial time on a Turing machine and since a polynomial of a polynomial is a polynomial, that means that polynomially many PC-instructions can be done in polynomial time on a Turing machine. So then you would go through some basic machine instructions like add, multiply, subtract – whatever a PC can do as instructions – and you would show Turing machine programs that do exactly the same in polynomial time.

It takes a little bit of work to actually do that, but it has been done. People have verified left and right for all kinds of computers that they are actually polynomial-time related. And this is how people came to believe in this thesis. Now, we are not going to go through all that details here in class. I do encourage you to think about this because this is completely central.

So by and large we believe in the Computational Complexity Thesis, but as we will see in the very last lecture, it has recently been proven that a new kind of computer, called the Quantum computer, are capable of disproving the thesis in the sense that there exist problems which can be solved in polynomial time on a Quantum computer, but which provably need super-polynomial time on a Turing machine. This Quantum computer is so new that it hasn't really been built yet, but many scientists believe that it will. We will talk more about Quantum computers in the last lecture.

(This is a blank page)

IN210 – lecture 5

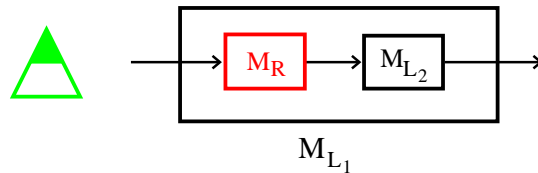


Polynomial-time simulations & reductions

We say that Turing machine M **computes function $f(x)$ in time $t(n)$** if, when given x as input, M halts after $t(|x|) = t(n)$ steps with $f(x)$ as output on its tape.

Function $f(x)$ is **computable in time $t(n)$** if there is a TM that computes $f(x)$ in time $\mathcal{O}(t(n))$.

For constructing the complexity theory we need a suitable notion of an efficient 'reduction':



We say that L_1 is **polynomial-time reducible** to L_2 and write $L_1 \propto L_2$ if there is a polynomial-time computable reduction from L_1 to L_2 .

Now that we know where we are going, we want to build up some machinery which will allow us to divide between those problems that are polynomial and those that are not. The most basic notion that we will need is the notion of a reduction. Because that notion will allow us to compare problems to each other, to organize them together and say: If we can solve this problem efficiently, then we can solve this other problem efficiently also, because here is a reduction.

We will say that Turing machine M computes function $f(x)$ in time $t(n)$ if given x as input, M produces $f(x)$ as output in $t(|x|)$ steps or less. And we will say that function $f(x)$ is computable in time $t(n)$ if there is a Turing machine that computes $f(x)$ in time $\mathcal{O}(t(n))$. Finally we say that language L_1 is polynomial-time reducible to language L_2 , and write $L_1 \propto L_2$, if there is a polynomial-time computable reduction from L_1 to L_2 .

(This is a blank page)

IN210 – lecture 5



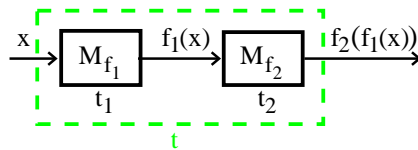
For arguments of the type

L_1 is hard/complex $\Rightarrow L_2$ is hard/complex

we need the following lemma:

Lemma 1 *A composition of polynomial-time computable functions is polynomial-time computable.*

Proof:



- $|f_1(x)| \leq t_1(|x|)$ because a Turing machine can only write one symbol in each step.
- “polynomial^{polynomial} = polynomial” or $(n^k)^l = n^{k \cdot l}$
- $t_2(|f_1(x)|)$ is a polynomial.
- $\text{TIME}(t) = t_1(|x|) + t_2(|f_1(x)|)$ is a polynomial because the sum of two polynomials is a polynomial.

Autumn 1999

5 of 12

These are all very straightforward, natural definitions – you don’t have to worry about them. What you have to worry about as I am going through these things is not the way the spelling goes and so on, but the way the theory is actually constructed. So you want to follow the big picture as I am going through the details. That is very important.

So this is just the most natural way of defining the notion of reduction. And that notion of reduction will give us something which is completely crucial, which is a way of saying: "If this problem is easy, then so is this other problem also." If we happen to know that L_1 is difficult and if we have an efficient reduction M_R from L_1 to L_2 , then we know that L_2 also must be difficult. Why? Because of the following kind of argument: If L_2 turns out to be easy, then L_1 also is easy – which is a contradiction. This is the scheme of things we have seen before in undecidability. Now we are just tuning or modifying the same scheme of things to deal with complexity.

So we need to be able say: If L_2 is easy so is L_1 , given that M_R is an efficient reduction, a polynomial-time reduction. And the lemma on this foil will allow us to say exactly that. Lemma 1 is a key element in our theory which will give us just the right notion of reduction that we need to construct the complexity theory.

The lemma says: "A composition of polynomial-time computable functions is polynomial-time computable." I am not really proving the lemma here, I am just telling you why this is so.

What is a composition of polynomial-time computable functions? You first

apply one function – one transformation – and then you apply another. You can use the following image: First you send the input to one algorithm which produces an output. That output is given as input to the other algorithm, which produces the overall output. When you put those two pieces into one box, then you have another algorithm, which is the composition of those two. I have drawn this image on the foil.

So M_{f_1} , the machine that computes the function f_1 , will be the first part in the box. M_{f_1} is going to produce output $f_1(x)$, given x as input. The first thing we want to observe is that the length of the output, $|f_1(x)|$, cannot possibly be larger than $t_1(|x|)$, where t_1 is the number of steps M_{f_1} takes in computing $f_1(x)$. The reason being that a Turing machine can output at most one character at the time. So the length of the output cannot possibly be longer than the time the Turing machine takes, its number of steps. This is the first piece in the proof of the lemma.

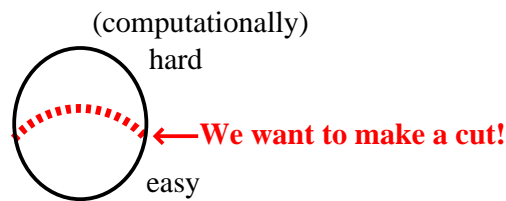
When the machine's time is bounded by a polynomial, then we know that its output is also bounded by a polynomial, in other words that size of output is polynomial in the size of the input. Now we are going to apply another polynomial-time algorithm to this polynomial-sized output, and we want to be able to assert that the overall output is polynomial in the length of the first input x . We are given $f_1(x)$ as input to M_{f_2} , and M_{f_2} will use time $t_2(|f_1(x)|)$ to compute the output. So we want to be able to say that $t_2(|f_1(x)|)$ is a polynomial.

Notice that $|f_1(x)|$ and $t_2(n)$ are both polynomials. That is why we are able to say that $t_2(|f_1(x)|)$ is a polynomial because from high-school mathematics we know that the polynomial of a polynomial is a polynomial! We have the equation $n^{k^l} = nk * l$.

So $t_2(|f_1(x)|)$ is a polynomial, and then we wrap it all up together by saying that the overall time for M_L , which is $t_1(|x|) + t_2(|f_1(x)|)$, is a polynomial because the sum of two polynomials is obviously still a polynomial.

Remember that a reduction means a mapping of 'Yes'-instances to 'Yes'-instances and 'No'-instances to 'No'-instances. Now we know that if we can reduce one problem to another problem in polynomial time, then we have in a way put those two problems together. We are able to say that if the second problem – the one that the first problem is reduced to – is easy, then the first problem is also easy. Because if we can solve the second problem in polynomial time, then we can also solve the first problem in polynomial time. Also, if we can somehow prove that the first problem is hard, then we automatically know that the second problem is also hard.

IN210 – lecture 5



**all solvable
problems**

Strategy

It is the same as before (in uncomputability):

- Prove that a problem L is easy by showing an efficient (polynomial-time) algorithm for L .
- Prove that a problem L is hard by showing an efficient (polynomial-time) reduction ($L_1 \propto L$) from a known hard problem L_1 to L .

Difficulty

Finding the first truly/provably “hard” problem.

Way out

Completeness & Hardness

Autumn 1999

6 of 12

That gives us almost all we need for constructing the complexity theory. We step back for a moment and look again from the highest level: We want to divide the solvable problems into computationally hard and computationally easy problems. (I am deliberately using several kinds of terminology because these appear in practice – they are really used.) And the strategy should be what we had before:

Proving that something is easy is easy – conceptually speaking. You show a good algorithm. So in this case, by showing a polynomial-time algorithm for a problem, we prove that the problem is “easy”.

Showing that a problem is hard, is harder. We need two things: We need a hard problem L_1 to begin with, and an efficient (polynomial-time) reduction from L_1 to our unknown problem L . These two things, by virtue of the lemma that we have seen, will amount to prove that L is also hard. This is so because if L on the contrary is easy, then L_1 would also be easy, but we know that L_1 is hard.

The difficulty here is in finding the first hard problem that will allow us to begin all this theory. Remember we used diagonalization earlier to get a hold of a hard, unsolvable problem, and then everything else was reductions. In complexity theory we are able to prove that certain problems are hard by using diagonalization of a different kind. Unfortunately those certain problems that we can prove hard, they are not the most interesting ones. For many practical problems we haven’t been able to prove that they are hard, even though they *seem* hard. That is the basic difficulty, and the way out of that difficulty is to introduce

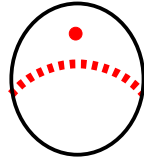
the notions of completeness and hardness.

So this is a kind of a cop-out. We will not really be able to prove that these problems are absolutely not solvable in polynomial time. But we will be able to do – for practical purposes – just about as well.

IN210 – lecture 5



\mathcal{NP} -completeness



How to prove that
a problem is hard?



Completeness

We say that language L is **hard for class C** with respect to polynomial-time reductions[†], or **C -hard**, if every language in C is polynomial-time reducible to L .

We say that language L is **complete for class C** with respect to polynomial-time reductions[†], or **C -complete**, if $L \in C$ and L is C -hard.

[†] Other kinds of reductions may be used



Note:

- If L is C -complete/ C -hard and L is **easy** ($L \in \mathcal{P}$) then every language in C is easy.
- L is C -complete means that L is “hardest in” C or that L “characterizes” C .

Autumn 1999

7 of 12

At this point we leave the nice problems and turn to the not so nice, intractable ones. And the big problem in constructing a theory about those, is that for a number of practical interesting, important intractable problems, we have not been able to prove that they really are complex – that they really cannot be solved in polynomial time. Many of the best theoreticians in the world have been working on it for many years without managing to come up with a valid proof. Most people believe that those problems really are hard, because nobody has come up with a polynomial-time algorithm for any of these intractable problems. But at this level of our proficiency, we are not able to prove it.

So for the time being we give that ambition up, and instead we introduce two slightly less, but practically just as powerful notions, called completeness and hardness. So completeness and hardness will basically be our theoretical notions that correspond to the notion of intractability. It will be a way to model intractability. I am now first going through the technical things here – defining what completeness and hardness are and explaining what they mean intuitively – and then we will see how to prove completeness or hardness.

We will say that a language L is hard for a class C with respect to polynomial-time reductions – or C -hard for short – if every language in C is polynomial-time reducible to L . And then we say that L is complete for class C with respect to polynomial-time reductions, or C -complete, if L is in C and L is C -hard. Please note that other kind of reductions maybe used for certain classes, i.e. for class Exponential time, exponential-time reductions are allowed, since exp. of an exp. is an exp.

I have just defined what being hard for a certain class C means formally, now here comes an essential insight which will tell us why this notion of hardness is exactly the kind of notion we need. The point is that if L is complete or hard for the class C of problems, and L turns out to be easy in the sense that L is computable in polynomial time, then it follows immediately that every language in class C is easy. This must be so because every language in C can then be solved in polynomial time by first reducing it in polynomial time to language L , and then solving the corresponding instance of problem L in polynomial time.

Now suppose that class C has all kinds of problems which people have studied for a very long time, without being able to solve them efficiently, meaning in polynomial time. By showing that L is hard for that class, we show that if L can be solved efficiently, then so can all those problems in C that people have been trying to solve for a long time.

So this is why proving completeness or hardness is a very strong piece of evidence that L indeed is hard, that it cannot be solved in polynomial time. It is not conclusive evidence however, but for practical purposes it is very strong evidence. And that is the best we can do at this point.

So, intuitively, by proving that L is complete for a certain class, we have proven that L is in a sense hardest in the class, hardest meaning: If L can be solved efficiently, then so can all the other languages in the class.

IN210 – lecture 5



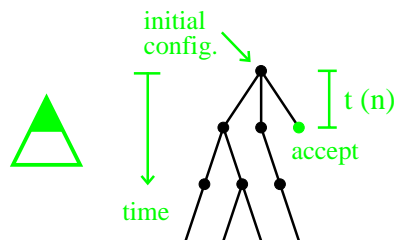
\mathcal{NP} (non-deterministic polynomial time)

A **non-deterministic Turing machine (NTM)** is defined as deterministic TM with the following modifications:

- NTM has a **transition relation** Δ instead of transition function δ

$$\Delta : \{((s, 0), (q_1, b, R)), ((s, 0), (q_2, 1, L)), \dots\}$$
- NTM says 'Yes' (accepts) by halting

Note: A NTM has many possible computations for a given input. That is why it is non-deterministic.



- Mathematician doing a proof \rightsquigarrow NTM
- The original TM was a NTM

Autumn 1999

8 of 12

This notion of hardness or completeness works with all sorts of classes, and we will start by looking at the class \mathcal{NP} , which will be our main class in this course. \mathcal{NP} is also in fact the most interesting class in complexity theory, not because it is such an interesting theoretical object as such, but because it is a drawer where most of the practically interesting and important problems live.

So \mathcal{NP} -complete will be a large and practically important class of problems, or languages, which we are now beginning to study. \mathcal{NP} means non-deterministic polynomial time, or polynomial time on a non-deterministic Turing machine. And the first thing we need is to introduce the non-deterministic Turing machine.

I am not going to go through all the details. I will just point to the differences between a non-deterministic and a deterministic Turing machine, because the two machines are very similar. Formally the basic difference is that a non-deterministic Turing machine – NTM for short – is non-deterministic, meaning that it will have a transition relation instead of a transition function.

Now, what is a function? A function is a kind of thing which given an input gives you a very precise, single output. That is a function. And a relation, it is like brothers and sisters – you can have several brothers. So in this case the transition relation will include several possible steps from a given situation. For example, in our standard situation where the machine is in state s and scanning a '0', this delta transition relation says that we can apply the rule we had before, $d(s, 0) = (q_1, b, R)$, or we can apply another rule: $d(s, 0) = (q_2, 1, L)$. So we can choose to move from this situation into q_2 , replace the zero by one and move

left.

So the rules of this machine they are kind of wishy-washy, and you can of course wonder what kind of a computer is this? I mean, there is not a computer in this world which has this property that from a given state, from a given situation, the machine itself can decide what to do. So this is strange.

I will come back to this in a second. Let me first finish with the differences. Since a NTM obviously can have several different computations on a certain input, then it wouldn't really make much sense for the machine to say explicatively 'Yes' or 'No'. Because some of these can be 'Yes'-computations, some of these can be 'No'-computations, and then this would be a very confusing kind of machine. So instead it is enough that the machine just says 'Yes' by halting. If the machine halts in state h , it says effectively 'Yes'. So this machine will be accepting. It can reject by just computing forever.

It is a very important observation that a NTM has many possible computations for a given input. That is why it is non-deterministic. Why is this a reasonable notion of computation? To answer that question we must go back to Turing's basic situation – when he was defining his machine – and recall that he was really modeling a mathematician solving a problem, proving a theorem. He wasn't modeling an actual computer, because in his time computers did not exist.

When a mathematician is doing a proof, then in any situation several possibilities always applies, several rules can be applied to his line to modify the line and get another line. So a mathematician is fundamentally a non-deterministic machine. And all that matters there is whether the mathematician can begin from the left-hand side of the theorem, of an equation, and by transforming it using the rules arrive at something which looks like the right-hand side, in which case he says 'Bingo!' And that is accepting – the end of the story.

Or a mathematician can spend his entire life modifying this left-hand side, and modifying and modifying, and never really finish – never obtain the right-hand side. In which case the non-deterministic Turing machine, which is the mathematician, has not managed to accept. It simply computes forever. There is no proof – or he could not find a proof – that the left-hand side is the same as the right-hand side.

So the point here – which justifies that NTM says 'Yes' by halting, by accepting – is that the acceptance is like coming up with the proof. If there is a proof, then the fact is correct. A mathematician may or may not be able to find the proof, but if there is such a thing as a proof, then we know that everything is in order.

So this is why a non-deterministic Turing machine is in fact a very reasonable kind of concept, but not as a piece of hardware, just as a concept.

We think of the computation of a NTM as being essentially a tree where the vertices are configurations. Because starting from an initial configuration there could be several possible configurations which result in one step, and then from each of those several possible configurations, and so on. So think of a computation of a NTM as a tree of configurations.

And then we introduce the notion of time as being basically the height of the tree. The time used by a NTM is defined as the length of the shortest path from the root to an accepting configuration – in other words the length of the shortest accepting computation.

IN210 – lecture 5



We say that a non-deterministic Turing machine M **accepts language L** if there exists halting computations of M on input x if and only if $x \in L$.

Note: This implies that NTM M never stops if $x \notin L$ (all paths in the tree of computations have infinite lengths).

We say that a NTM M **accepts language L in (non-deterministic) time $t(n)$** if M accepts L and for every $x \in L$ there is at least one accepting computation of M on x that has $t(|x|)$ or fewer steps.

We say that $L \in \mathbf{NTIME}(t(n))$ if L is accepted by some non-deterministic Turing machine M in time $\mathcal{O}(t(n))$.

$$\mathcal{NP} = \bigcup_k \mathbf{NTIME}(n^k)$$

Note: All problems in \mathcal{NP} are decision problems since a NTM can answer only 'Yes' (there exists a halting computation) or 'No' (all computations "runs" forever).

Autumn 1999

9 of 12

We say that a NTM M accepts language L in non-deterministic time $t(n)$ if for every string x in L , there exist an accepting computation of M on input x that has $t(|x|)$ or fewer steps. This just formalizes this notion of time in a non-deterministic Turing machine.

In the world of the proofs we would say that something has a proof that has $t(|x|)$ steps. A step is like using a rule, using a transformation, or a step could be a line in the proof. So this is how this notion of time really makes sense.

And then the rest is as before. We introduce a complexity class \mathbf{NTIME} , non-deterministic time, and we say that a language is in \mathbf{NTIME} of $t(n)$ if the language is accepted by some non-deterministic Turing machine in time $\mathcal{O}(t(n))$. And then we define \mathcal{NP} – non-deterministic polynomial time – in essentially the same way as we defined the deterministic polynomial time.

It is worth nothing that \mathcal{NP} consist entirely of decision problems since a NTM are only able to answer 'Yes' (halt) or 'No' (compute forever).

IN210 – lecture 5



The meaning of “ L is \mathcal{NP} -complete”

Complexity

Many people have tried to solve \mathcal{NP} -complete problems efficiently without succeeding, so most people believe $\mathcal{NP} \neq \mathcal{P}$, but nobody has **proven** yet that \mathcal{NP} problems need exponential time to be solved.

L is computationally hard ($L \in \mathcal{NP}$):

$$L \in \mathcal{P} \Rightarrow \mathcal{NP} = \mathcal{P}$$

Physiognomy

Checking if $x \in L$ is easy, given a certificate.

We know what the class \mathcal{NP} is formally, and we know what completeness and hardness are formally. As a result we now know in a very precise sense what it means for a language to be \mathcal{NP} -complete or \mathcal{NP} -hard.

Again we go to the top of our pyramid and look at these formal notions. We want to understand why they actually model exactly what we want them to model. Why is it meaningful to prove that a language is \mathcal{NP} -complete or \mathcal{NP} -hard?

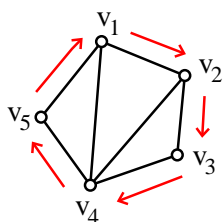
And there are two reasons: The first one is that there are many problems which have been shown to belong to the class \mathcal{NP} , which are computationally difficult. For many, many years people have been trying to solve them, without being able to solve them efficiently. The best solutions we know to those \mathcal{NP} -hard problems runs in exponential time. We will see a number of those problems.

So proving for another problem L that it is \mathcal{NP} -complete or \mathcal{NP} -hard would mean that if it turns out that L can be solved in polynomial time, then all problems in \mathcal{NP} can be solved in polynomial time. \mathcal{NP} -completeness is very strong evidence that a language is hard.

We have however more than that. Proving that a problem is \mathcal{NP} -hard tells us something very important about the physiognomy of that problem, about its nature. We will see that each instance x of an \mathcal{NP} -hard language L has a short certificate, and there will be a deterministic machine which can use that certificate to check or verify in polynomial time that x is a member of L . We will now talk more about that.

IN210 – lecture 5

Example: HAMILTONICITY



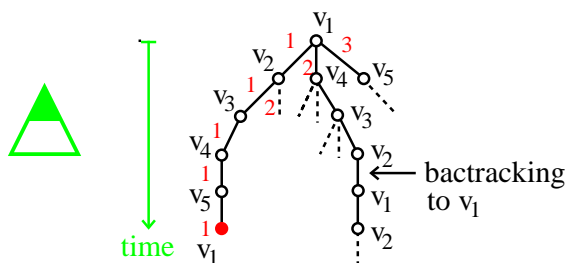
- A deterministic algorithm “must” do exhaustive search:

$v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow v_2 \rightarrow$ **backtrack**

$\swarrow v_2 \rightarrow$

$n!$ possibilities (exponentially many!)

- A non-deterministic algorithm can **guess** the solution/**certificate** and verify it in polynomial time.



Certificate: **(1,1,1,1,1)**

Note: A certificate is like a ticket or an ID.

Autumn 1999

11 of 12

Here is one very common example of a \mathcal{NP} -complete problem, which is the HAMILTONICITY problem. The input instance is a graph and as usual in these problems we are asked to test whether this graph has a certain property. And the property is whether the graph is what is called 'Hamiltonian', or whether the graph has a Hamiltonian cycle. A Hamiltonian cycle is a path in the graph – a sequence of vertices – such that every pair of vertices in that sequence is an edge in the graph. And this sequence of vertices has to be such that it visits each vertex in the graph exactly once. If you are talking about the Hamiltonian cycle, then we want to go back to the original vertex where we started from, but if you are talking about the Hamiltonian path, then we just want all the vertices to be on the path.

Which variant of HAMILTONICITY we are talking about is not really important, because the two are closely related. If one is hard, the other one is hard. If one is easy, the other one is easy. Think about that. Why is it so? Can you prove that if HAMILTONIAN PATH is solvable in polynomial time, so is HAMILTONIAN CYCLE? Try to make that proof.

So given an instance of HAMILTONICITY, as on the foil, what would a deterministic algorithm do to test whether the input is Hamiltonian or not? A deterministic algorithm basically would do an exhaustive search in one way or another. It would test all possibilities. And the only way algorithm design comes into play here is that we do this in a kind of a clever way so that we don't waste too much time on possibilities that are obviously hopeless.

A typical deterministic algorithm would try a path. It would begin from, say,

v_1 and then extend the path by moving to one neighbor of v_1 , for example v_4 . From v_4 it would move to, say, v_3 . And then proceed to v_2 . But after coming to v_2 the algorithm is stuck, meaning that the path cannot be extended anymore. It is not Hamiltonian yet because v_5 is not there. Then the algorithm has to backtrack. It has to undo what it has done, and try other possibilities.

So this gives us a kind of an algorithmic approach which relies on trial and error, but ultimately this approach is about trying all kinds of possibilities. How many possibilities are there? If you are talking about possible Hamiltonian paths, then by and large there are $n!$ possibilities. We can think of n factorial as being roughly n^n . So it is even more than 2^n . It is a phenomenal large number for even a moderate small n . We could of course reduce the number of possibilities by using some clever approaches, but it would still be exponentially many left to try.

So there are exponential many possibilities, and therefore these deterministic exhaustive search algorithms are by their very nature exponential time. So this illustrates a little bit what an exponential-time algorithm is, and how exponential time naturally arises in this context.

It turns out that when we use the non-deterministic Turing machine, suddenly this exponential time turns into polynomial time. This is because a non-deterministic algorithm can in effect guess the solution. This guessing is not really guessing, it is just stating a fact that there exists a short proof of the fact that this graph is hamiltonian.

What is the short proof that this graph is Hamiltonian? It is the Hamiltonian path itself! If I tell you: "Look at the red arrows in the graph. There is a Hamiltonian cycle in this graph." Given this piece of information, the path, any deterministic machine can easily verify efficiently – in polynomial time – that this indeed is the path. And by doing so the machine verify that the graph is indeed Hamiltonian, that it has indeed the property.

And that is the nature, or the physiognomy, of the problems in \mathcal{NP} . They all have this property that given guessing, given non-determinism, verification of a property can be done fast. How does this relate to the non-deterministic tree of computation?

Imagine that we have an algorithm which is scanning v_1 , and the algorithm basically says: Take one of the neighbors and try to extend the path in that way, and then move on from the neighbor, taking its neighbor, and so on. So this is a non-deterministic algorithm. It doesn't tell you which neighbor exactly. But in this example there are 3 possibilities from v_1 , and each of these possibilities give you a different configuration. And from each of them certain other configurations – intuitively speaking – are accessible.

So this tree of configurations gives us all possible searches in this graph. And if there is a positive answer, meaning a short accepting computation, then the answer is in this tree. So if the Turing machine is clever enough to take this path – $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_1$ – then it has found the solution.

These other possibilities here – moving to vertex 4 and then to vertex 3 and so on – these are the backtracking possibilities where the machine has basically gone a certain way, gotten stuck and then backtracked all the way back, and then tried another possibility, and so and so on. These take longer time.

The big point about a NTM is that it can magically guess this short solution, and then check in polynomial time that it is really a solution. We will see how shortly, but first a comment about our definition of the NTM.

You will notice that in your G&J textbook the NTM is a little bit different. The machine really just has a kind of a magical guessing module which allows the machine to first compute a string, and then use a deterministic algorithm on the combination of that magical guessed string and the input.

What we have defined in this lecture is the actual non-deterministic Turing machine, the standard one. The one in your books is non-standard and it is not very good. It is really only suitable for proving \mathcal{NP} -completeness. It is a kind of a simplification done for people who are just interested in technical proofs of \mathcal{NP} -completeness.

We will stick to the definition given today, but you can read your G&J textbook and find out that the definition you have there is for the purpose of proving \mathcal{NP} -completeness roughly equivalent to our. You might want to think about why that is so.

So, since a NTM can in fact always begin its computation by producing a random string on its tape – '01011101' or whatever – and then proceed as a deterministic machine, that gives the non-deterministic Turing machine the ability to guess a string!

If the NTM can guess a sequence of vertices which is the right one if and only if the graph is Hamiltonian, then overall we have an efficient non-deterministic algorithm, in theory. More intuitively we think of this as the members of the HAMILTONICITY language being the kind of folk who have, each of them, an identity card or a ticket in their pocket. It is like having a tram ticket in my pocket. When I come to the train, I show it to the guard. The guard looks at it, sees my picture and says 'Yes, OK, you can go in'.

Each member of the HAMILTONICITY language has a short certificate – short in the sense that it is polynomial in its size – such that given that certificate a deterministic algorithm can check or verify in polynomial time the fact that the graph is hamiltonian. By using the short certificate a deterministic algorithm can efficiently verify whether the instance is a member of the HAMILTONICITY language. The existence of this certificate is the essential physiognomy of languages in \mathcal{NP} . They all have this short certificate.

(This is a blank page)

IN210 – lecture 5



Proving \mathcal{NP} -completeness

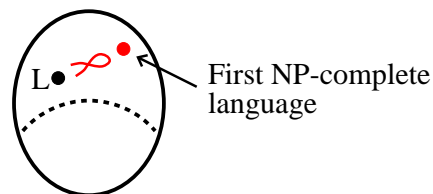
1. $L \in \mathcal{NP}$

Prove that L has a “short certificate of membership”.

Ex.: HAMILTONICITY certificate = Hamiltonian path itself.

2. $L \in \mathcal{NP}$ -hard

Show that a known \mathcal{NP} -complete language (problem) is polynomial-time reducible to L , the language we want to show \mathcal{NP} -hard.



Cook's Theorem (Cook, Levin 1971/73)

SATISFIABILITY is \mathcal{NP} -complete.

Autumn 1999

12 of 12

Now we are moving on to the next big theme which is proving \mathcal{NP} -completeness. Remember that \mathcal{NP} -completeness consists of two facts. The first fact being that L is in \mathcal{NP} . Instead of constructing a messy NTM which accepts L in non-deterministic polynomial time as it is defined by the definition, we take a much easier route: We prove that L has a short certificate of membership, a kind of a ID-card. For example in the case of HAMILTONICITY the certificate is the Hamiltonian path itself, or the Hamiltonian cycle.

The second element of proving completeness is proving that L is hard for \mathcal{NP} . This is even more difficult. We would have to prove that all possible languages in \mathcal{NP} are polynomial-time reducible to L . There are infinitely many of them, so how can we do that? As we have done before in undecidability theory, we do that by proving one language \mathcal{NP} -hard – the first one. And then once we have one language, it is enough to show that this one \mathcal{NP} -complete language is polynomial-time reducible to our L . This is enough because if every language in \mathcal{NP} is polynomial-time reducible to our first \mathcal{NP} -complete language, and this first language is polynomial-time reducible to our new language L , then by virtue of the fact that the composition of two polynomial-time reductions is polynomial time, that means that every language in \mathcal{NP} is polynomial-time reducible to our language L .

So what is needed now is the first \mathcal{NP} -complete language. And the first \mathcal{NP} -complete language is given by an essential piece of theory: Cook's Theorem, which was proven independently by Cook and Levin in 1971/73. The theorem says that SATISFIABILITY is \mathcal{NP} -complete. So SATISFIABILITY will be our first

\mathcal{NP} -complete problem, and an essential foundation stone for us to build the theory.

3.6 Lecture 6

IN210 – lecture 6

NP-completeness

have no feasible solutions

have feasible solutions

solvable problems

$L \in \mathcal{NPC} \Leftrightarrow L \in \mathcal{NP}$ and $L \in \mathcal{NP}$ -hard

Today: Proving \mathcal{NP} -completeness

- $L \in \mathcal{NP}$: show that there is a “short”[†] **certificate** of membership in L (“id card”).
- $L \in \mathcal{NP}$ -hard: show that there is an “efficient”[†] **reduction** from a known \mathcal{NP} -hard problem L_{np} to L .

[†] polynomial (length, time . . .)

Autumn 1999 1 of 15

G&J:
2.6, 3.0-3.1.2

This lecture will be about \mathcal{NP} -completeness. We have thrown away the unsolvable problems and we are now dealing with the solvable ones. We are about to divide them into two classes: those that have feasible solutions and those that don't seem to have feasible solutions.

What is feasible? Feasible means something that is practical, that works in practice. We have seen that some problems, in fact a lot of problems, have algorithms which take practically forever to compute even for small or medium-sized inputs. They are useless for reasonable large instances. So we want to distinguish between problems that have proper, nice, efficient algorithms and those that don't.

So this practical concern, with a little bit of abstraction and formalization, has given us some theoretical notions that we are now studying. Those notions are complexity classes, and in particular class \mathcal{P} and class \mathcal{NP} -complete. Class \mathcal{P} models or captures those problems that have feasible solutions. \mathcal{NP} -complete will be a large, important class of problems that seem not to have feasible solutions. Those are the \mathcal{NP} -complete problems. The importance of the class \mathcal{NP} -complete lies in the fact that many of the practical important intractable problems live there.

Today we will see how to prove \mathcal{NP} -completeness. What does an \mathcal{NP} -complete problem or language look like? Notice that I am mixing the words 'problems' and 'languages'. When I say 'problem' I am in the informal world. When

I say 'language' then we are completely in the formal world on the bottom of the pyramid. But 'problem' and 'language' are really in a sense synonyms. They serve the same purpose. 'Language' is just a formalization of 'problem'.

We say that a language L is in the class \mathcal{NP} -complete, or that L is \mathcal{NP} -complete, if L is in the class \mathcal{NP} and L is \mathcal{NP} -hard. \mathcal{NP} -hard means for our purpose that every problem in \mathcal{NP} is polynomial-time reducible to L . Intuitively that means that L is at least as hard as any problem in \mathcal{NP} . Because if we happen to solve L in polynomial time, then every other problem in \mathcal{NP} can be solved in polynomial time also.

How do we prove that L is \mathcal{NP} -complete? The first part – proving that L is in \mathcal{NP} – amounts to showing that there is a "short" (polynomial in the length of the input) certificate of membership in L , a kind of an ID-card. What is \mathcal{NP} ? It contains all languages that can be solved by a non-deterministic Turing machine in polynomial time.

We have seen that a non-deterministic Turing machine can effectively guess. It can begin by writing a sequence of characters on its tape non-deterministically or randomly, and then use them in a deterministic computation to test something. So in effect a NTM can guess. And if the NTM can guess some piece of information that amounts to a proof that a string x is member of language L , and this proof can be verified in deterministic polynomial time, then we say that L is in class \mathcal{NP} .

You can use the following image: If there is a kind of a ticket which an instance of a language L can keep in its pocket and show on the entrance, and then the guard on the entrance can test in polynomial time whether the ticket is a valid ticket or not, then we say that L is in \mathcal{NP} . It has a short certificate.

\mathcal{NP} -hardness is normally shown by showing an "efficient" reduction from a known \mathcal{NP} -hard problem to L . So think of a known \mathcal{NP} -hard problem L_{np} . By definition every language in \mathcal{NP} is polynomial-time reducible to L_{np} . If we can make a polynomial-time reduction from L_{np} to L , then L is also \mathcal{NP} -hard because every problem in \mathcal{NP} is polynomial-time reducible to L by the virtue of the lemma which we have proven last time, which says that a composition of polynomial-time reductions is a polynomial-time reduction. The figure on the foil illustrates this argument.

IN210 – lecture 6

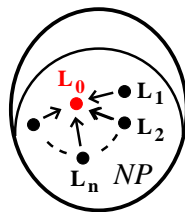
**Skills to learn**

- Transforming problems into each other.

Insight to gain

- Seeing unity in the midst of diversity: A variety of graph-theoretical, numerical, set & other problems are just variants of one another.

But before we can use reductions we need **the first \mathcal{NP} -hard problem**.

**Strategy**

As before:

- 'Cook up' a complete Turing machine problem
- Turn it into / reduce it to a natural/known real-world problem (by using the familiar techniques).

Autumn 1999

2 of 15

Our goal for this lecture and the next one will be to show for a number of problems that they are \mathcal{NP} -complete. Why are we doing that? First of all, we are going to learn certain important technical skills. The most important skill that we are going to learn, is how to transform different common decision problems into each other. So we will be looking at how to transform problems into each other.

And by doing so we will learn a little bit about the nature of those problems and gain some insights. We will build intuition. The most important insight that we are going to acquire, hopefully, is an ability to see unity in the midst of diversity. We will see that in fact a variety of problems – some involving numbers, some involving graphs, some involving sets – are actually variants of one and the same meta-problem or situation. And when we see that, we understand that there are not so many different things in this world. Actually a lot of things that seem very different they are kind of the same or similar. And then we will understand that all of those problems can be dealt with in a similar way, algorithmically speaking. They can be understood in similar terms.

So we are organizing the world of problems. That is the important insight to gain from this exercise of reducing problems to one another.

Before we can use reductions to prove \mathcal{NP} -completeness, we need the seed. We need the first \mathcal{NP} -complete problem, the first \mathcal{NP} -hard problem. And we need to be able to prove that every problem in \mathcal{NP} is reducible to our problem. That is our first task and that is the hard part. The rest is much easier and actually very similar to what we have been doing before on unsolvability.

In order to find our first hard problem, we will be using roughly speaking the same kind of strategy as before, namely: We will first cook up a complete problem, something that involves Turing machines, that is not necessarily very natural – kind of an artificial problem. This artificial problem will have the nice property that it is easy to prove that it is complete, and then we will reduce that unnatural problem to one or more known real-world problems by using the familiar techniques. You will recognize that the techniques that we will be using, are in fact just variants of the techniques that we have seen when we were proving uncomputability. So what we will see now will almost be a review in a way.

IN210 – lecture 6

**BOUNDED HALTING problem**

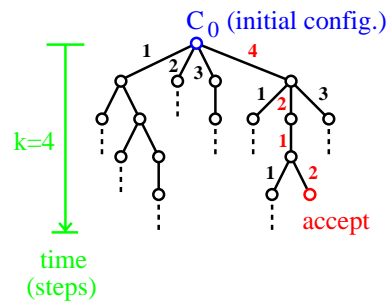
$$L_{BH} = \{(M, x, 1^k) \mid \text{NTM } M \text{ accepts string } x \text{ in } k \text{ steps or less}\}$$

Note: 1^k means k written in unary, i.e. as a sequence of k 1's.

Theorem 1 L_{BH} is \mathcal{NP} -complete.

Proof:

- $L_{BH} \in \mathcal{NP}$



Certificate: (4, 2, 1, 2). The certificate, which consists of k numbers, is “short enough” (polynomial) compared to the length of the input because k is given in unary in the input!

Autumn 1999

3 of 15

Our first cooked-up, unnatural problem is the BOUNDED HALTING problem. I am now defining this problem as a language. This language consists of triples: a Non-deterministic Turing Machine code M , an input x and a number k written in unary, meaning just a sequence of k 1's. It will be obvious in a moment why this k is written in unary. And this triple is in the language L_{BH} if NTM M happens to accept the string x in k step or less – in other words if one of the possible computations of NTM M on input string x reaches a halting configuration in k or less steps.

Notice that this is just the most natural thing. This is in fact the Halting problem extended a little bit so that it has a little bit of complexity flavor. We are not only asking whether M halts on x , but whether M actually halts on x in k steps or less. And notice also that M happens to be a non-deterministic Turing machine. The rest is the same.

We are now going to prove that this BOUNDED HALTING problem is \mathcal{NP} -complete. And the proof of course is done in two steps as always. Step 1: BOUNDED HALTING is in \mathcal{NP} . It is sufficient to show that there is a membership certificate which a NTM can guess and then check deterministically in polynomial time that the instance is a positive instance, if it is. So every positive instance, every member of the set, should in principle have a ticket which proves that he is a member.

So what is this certificate? What does it look like in this case? It is basically just a sequence of numbers that helps us get rid of non-determinism. Non-determinism is just a bunch of non-deterministic choices. In every step of com-

putation a NTM chooses one of several possible transformations.

In the computation tree on the foil there is an accepting configuration which is marked red. How can a poor deterministic algorithm know that there is a place like this? Well, given the non-deterministic choices that the NTM does during the accepting computation, a deterministic algorithm can easily verify that there is an accepting configuration after k steps, by following the path to the accepting configuration. And because k is written in unary, this verification can be done in time polynomial in the length of the input triple.

The non-deterministic choices can be recorded by enumerating the possible transformations from each configuration. In our example there are 4 configurations leading from the initial configuration C_0 . The first number in the certificate "(4, 2, 1, 2)" says: "Take the 4th one!" So the deterministic Turing machine says: "OK, I follow the 4th one. Here I am. What now?" The ticket says: "Take now the 2nd one, go down, then take the only one, which is 1. And then take the 2nd one again, and there you are." So the sequence of numbers in the certificate identifies uniquely the path to the accepting configuration.

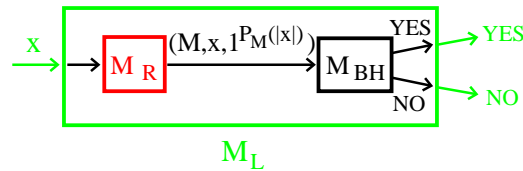
So to turn non-determinism into determinism, all you need is the specification of the non-deterministic choice in every step – that is the certificate.

So given an instance of BOUNDED HALTING and given a certificate, what is the deterministic algorithm that checks whether x is accepted by M in 1^k steps? Well, just use the certificate and the universal TM to simulate M on input x for k steps, and see if in the end this lead to acceptance. This simulation is very easy, and it can be done in polynomial time with respect to the length of the input triple because the number k – the length of the path, the number of steps – is written in unary!

IN210 – lecture 6



- $L_{BH} \in \mathcal{NP}$ -hard



- For **every** $L \in \mathcal{NP}$ there exists by definition a pair (M, P_M) such that NTM M accepts every string x that is in L (and only those strings) in $P_M(|x|)$ steps or less.
- Given an instance x of L the reduction module M_R computes $(M, x, 1^{P_M(|x|)})$ and feeds it to M_{BH} . This can be done in time polynomial to the length of x .
- If M_{BH} says 'YES', M_L answers 'YES'. If M_{BH} says 'NO', M_L answers 'NO'.

Autumn 1999

4 of 15

Part two of the proof: We have proven that L_{BH} is in \mathcal{NP} . Now we want to prove that L_{BH} is \mathcal{NP} -hard, namely that every problem in \mathcal{NP} is polynomial-time reducible to L_{BH} . We want to show that there is a reduction from every problem in \mathcal{NP} . It sounds like a difficult task, but it is actually trivial.

And here is why it is trivial: We know by definition that every language in \mathcal{NP} has a NTM M and a polynomial P_M , such that M accepts every string x that is in L and only those strings, in $P_M(|x|)$ steps or less. This follows from the definition of the class \mathcal{NP} .

So given an instance x of L the reduction module M_R compute the triple $(M, x, 1^{P_M(|x|)})$. M is a constant, namely a NTM code, and x we just copy. The only thing to compute really, is this polynomial $P_M(|x|)$. It is quite easy to show that a TM or an algorithm can compute a polynomial in polynomial time. We will not prove it here – it is a small technicality – but it is worth thinking about how it can be done.

$P_M(|x|)$ can be computed in polynomial time, and then M_R outputs $P_M(|x|)$ 1's, which is trivial work. So this whole transformation is done in time which is polynomial in the length of x .

So given x , by computing this triple we have reduced this instance of L to an instance of BOUNDED HALTING. And it follows from the definition of L_{BH} that this instance of BOUNDED HALTING is a positive instance if and only if x is a positive instance of L . This reduction can always be done in polynomial time by an algorithm, so in effect we have a polynomial-time reduction from an arbitrary language L in \mathcal{NP} to BOUNDED HALTING.

So this is very nice. We have our complete problem, which is rather surprising. It is surprising that there is 'the hardest problem' in this large class of problems, \mathcal{NP} . 'Hardest problem' means that every problem is polynomial-time reducible to it, so that if we can solve that hard problem efficiently – if we can solve our BOUNDED HALTING problem efficiently – we can solve all the problems in \mathcal{NP} efficiently. That is good news. We also have this interesting concept of completeness.

But the ugly part is that this BOUNDED HALTING – I mean, who cares? Who really cares whether a Turing machine halts in k steps or less?

IN210 – lecture 6

**SATISFIABILITY (SAT)**

The first real-world problem shown to be \mathcal{NP} -complete.

Instance: A set $C = \{C_1, \dots, C_m\}$ of **clauses**. A clause consists of a number of **literals** over a finite set U of boolean variables. (If u is a variable in U , then u and $\neg u$ are literals over U .)

Question: A clause is **satisfied** if at least one of its literals is TRUE. Is there a **truth assignment T** , $T : U \rightarrow \{\text{TRUE}, \text{FALSE}\}$, which satisfies all the clauses?

Example

$$I = C \cup U$$

$$C = \{(x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_2), (x_1 \vee x_2)\}$$

$$U = \{x_1, x_2\}$$

$T = x_1 \mapsto \text{TRUE}, x_2 \mapsto \text{FALSE}$ is a satisfying truth assignment. Hence the given instance I is **satisfiable**, i.e. $I \in \text{SAT}$.

$$I' = \begin{cases} C' = \{(x_1 \vee x_2), (x_1 \vee \neg x_2), (\neg x_1)\} \\ U' = \{x_1, x_2\} \end{cases}$$

is not satisfiable.

Autumn 1999

5 of 15

For the next step in building the \mathcal{NP} -completeness theory, we want a natural, real-world, nice, interesting \mathcal{NP} -complete problem. And the first such problem that was proven \mathcal{NP} -complete, was SATISFIABILITY.

An instance of SATISFIABILITY is a set of clauses – c_1 to c_m – over a finite set U of variables. What are clauses? Well, clauses involve some variables or negated variables – called literals – linked to together by logical OR-functions. We say that a clause is satisfied or satisfiable if at least one of the literals in the clause is satisfied or satisfiable – has 'TRUE' as its truth value.

A question that is associated with an instance of SATISFIABILITY is whether there is a truth assignment to all the variables which satisfies all the clauses. A truth assignment is a mapping of the variables to the values 'TRUE' or 'FALSE'. Or in other words, assigning the values 'TRUE' and 'FALSE' to the variables.

I have made an example on the foil. I is an instance of SATISFIABILITY. It consists of three clauses and the universe is the two variables x_1 and x_2 . We want to see whether there is a truth assignment which assigns the values 'TRUE' and 'FALSE' to x_1 and x_2 such that all of the clauses are satisfied.

I claim that we have a satisfying truth assignment if we say that x_1 is 'TRUE' and x_2 is 'FALSE'. Let's check: x_1 is 'TRUE' therefore the first clause $(x_1 \vee \neg x_2)$ is satisfied. $\neg x_1$ is 'FALSE', but since x_2 is 'FALSE', then $\neg x_2$ is 'TRUE', so the second clause $(\neg x_1 \vee \neg x_2)$ is satisfied also. And then x_1 is 'TRUE', therefore $(x_1 \vee x_2)$ is 'TRUE'. So all three clauses are 'TRUE', and this is indeed a satisfying truth assignment.

Given a satisfying truth assignment it is easy to verify in polynomial time

that an instance is a positive instance. Therefore, as it is often the case, the solution itself is the certificate of membership needed for proving that a decision problem is in \mathcal{NP} . So we know that SATISFIABILITY is in \mathcal{NP} .

I' is an instance that is not satisfiable. This is why: We have $(x_1 \vee x_2)$ so we know that one of variables must be 'TRUE'. And then we have $(x_1 \vee \neg x_2)$, and then we know that x_1 actually must be 'TRUE' for the first two clauses to be satisfiable. But the third clause $(\neg x_1)$ is only 'TRUE', if x_1 is 'FALSE'. So there is no truth value to x_1 that satisfies all three clauses. Therefore this instance is not satisfiable. It is a negative instance.

IN210 – lecture 6



Theorem 2 (Cook 1971) SATISFIABILITY is \mathcal{NP} -complete.

Proof – main ideas:

BOUNDED HALTING

“There is a
computation”



SATISFIABILITY

“There is a
truth assignment”

computation \rightsquigarrow (computation) matrix

Example: input $(M, 010, 1^4)$

	b	b	b	b	b	b	b	h	b
	b	b	b	b	b	b	b	q_3	b
	b	b	b	b	b	b	q_2	0	b
	b	b	b	b	b	q_1	0	0	b
	b	b	b	b	s	0	1	0	b

Computation matrix A is polynomial-sized (in length of input) because a TM moves only one square per time step and k is given in unary.

Autumn 1999

6 of 15

Now we want to prove that SATISFIABILITY is \mathcal{NP} -complete. We have already seen that SAT is in \mathcal{NP} . It remains to prove that SAT is \mathcal{NP} -hard. We are going to do that by reducing the BOUNDED HALTING problem to it.

You have a variant of this proof in your G&J textbook. Here I will just go through the main ideas. I will stay pretty much on the high level and explain to you what is really involved in this kind of proof.

On the high level you will see that there is nothing new, really. We are doing what we were doing before, namely: We have a TM problem, and we are reducing it to a real-world problem. We do it essentially by using this old idea of a computation matrix where we represent the configurations as rows, and then we represent the rules of a TM M and the input x with a bunch of templates.

Let us be a little more careful, just because this is the first reduction we have seen. Let's see how one actually goes about making these reductions. The first thing one should do is to put the two problems in front of oneself on a table. We put the problem that is being reduced on the left-hand side, and the problem that that problem is being reduced to on the right-hand side. And then we look at those two problems. What are they about?

An instance of BOUNDED HALTING is a positive instance if there is a computation of the NTM – and so on and so forth – that leads to acceptance after that many steps. But the point is that there is a *computation*. That is the certificate. That is what we need for membership. That is what an instance needs to be a positive instance.

What about SATISFIABILITY? There an instance is a positive instance if there

is a truth assignment that satisfies all the bla, bla, bla, bla – right? So the point is that this computation on the one side, corresponds to a truth assignment on the other side. And we better find a kind of a mapping that maps nice computations into nice truth assignments, meaning 'Yes'-instances to 'Yes'-instances and 'No'-instances to 'No'-instances. That is the obvious idea.

And the only thing we are going to do that is slightly technical is that we do the mapping with the help of this big matrix. This is really the natural way to think about a Turing machine in terms which are not so much Turing machine like, which are more a kind of organized and nice.

So imagine for a moment that we are dealing with our old friend, the TM that is going to verify whether the input string is '010' – the TM that we know the best. This TM happens to be deterministic, but don't worry about that – deterministic machines are special cases of non-deterministic ones. This will allow us to just go through the main ideas.

In our example the input x is '010' and k happens to be 4. So this is a positive instance. Now notice the following: In undecidability theory the computation matrix was a kind of an infinite matrix, but that is not what we can use in complexity, because in complexity everything is limited with time. But fortunately, since we know that the question is whether the machine accepts in k steps or less, we don't have to look beyond k steps. It is enough too look at k steps.

What can the machine do in k steps? If the head starts scanning the first '0' in the input (the arrow at bottom of matrix on the foil), then after k steps it cannot go further then k squares away on the tape, because the machine moves only one square at the time. And then we know also that we don't need to look at this matrix beyond k rows away from the first row, because that is how much time we have.

So, we end up with a matrix that has $2k + 1$ columns and $k + 1$ rows. And notice that the size of this matrix is polynomial in k , and since k is given in unary, the size is also polynomial in the length of the input instance of BOUNDED HALTING. That is essential.

So this matrix, which we will use in our proof, is polynomially bounded. It is a polynomial thing. And in our minds we are now being trained to interpret this word polynomial as short and manageable and nice. So we know that we have a nice matrix here because it is of polynomial size. We can now do all sorts of computations, and if those computations are polynomial, then the overall computation will be polynomial because we are dealing in a polynomial way with polynomially many things – that gives us a polynomial kind of computation, intuitively speaking. So we have something nice to work with.

IN210 – lecture 6



tape squares \mapsto **boolean variables**

Ex. Square $A(2, 6)$ gives variables $B(2, 6, 0)$, $B(2, 6, b)$, $B(2, 6, \overset{q_0}{0})$, etc. – but only polynomially many.

input symbols \mapsto **single-variable clauses**

Ex. $A(1, 5) = \overset{s}{0}$ gives clause $(B(1, 5, \overset{s}{0})) \in C$.

Note that any satisfying truth assignment must map $B(1, 5, \overset{s}{0})$ to TRUE.

rules/templates \mapsto **“if-then clauses”**

Ex.

	d	
a	b	c

 gives $((B(i-1, j, a) \wedge B(i, j, b) \wedge B(i+1, j, c)) \Rightarrow B(i, j+1, d)) \in C$.

Note: $(u \wedge v \wedge w) \Rightarrow z \equiv \neg u \vee \neg v \vee \neg w \vee z$

Since the tile can be anywhere in the matrix, we must create clauses for all $2 \leq i \leq 2k$ and $1 \leq j \leq k$, but only polynomially many.

Given an instance of BOUNDED HALTING, we have created this matrix – at least in our mind. Now we want to take this process one step further. We want to move from the world of matrix and templates where BOUNDED HALTING lives, into the world of logic where SAT lives. SATISFIABILITY is a problem in logic.

How do we move from computation matrix and templates to logical things? This is very important and completely crucial: We do it step by step. We translate each entity in the world of matrix and templates to the world of logic. We express each essential property of the computation matrix and the Turing machine rules/templates in the language of SAT, which is variables and clauses. And we must do the translation in such a way that the BOUNDED HALTING instance is a positive instance if and only if the translated SAT-instance is a positive instance, meaning: "There is a computation..." if and only if "There is a truth assignment...". This is an essential insight.

First of all we translate the squares in the matrix. We do that by introducing a number of logical variables $B(x, y, z)$ for every entry $A(x, y)$ in the matrix. Entry $A(x, y)$ gives rise to logical variables $B(x, y, 0)$, $B(x, y, 1)$, $B(x, y, b)$, $B(x, y, s/0)$ and so on – as many B -variables as there are characters which can be placed in the entry $A(x, y)$ in the matrix. So each of the $B(x, y, z)$ variables will express the fact that character z occupies the spot $A(x, y)$ in the matrix. Notice that there will only be polynomially many B -variables all together because we have a polynomial number of squares and a polynomial number of different characters to put in each square.

Our goal is that exactly one of these $B(x, y, z)$ variables will end up being

'TRUE' in a satisfying truth assignment, namely the variable that corresponds to the character which is actually written in spot $A(x, y)$ during the computation.

The first row of the computation matrix is fixed for a given instance, because it contains the input string. How do we translate the input string? We do it by introducing $2k + 1$ single-variable clauses, one clause for each square on the input row. If $A(1, 5) = \overset{s}{0}$ – as in our example – then we include the clause $(B(1, 5, \overset{s}{0}))$ into the set C of clauses. Because this is a single-variable clause, every satisfying truth assignment must map the variable $B(1, 5, \overset{s}{0})$ to 'TRUE'. And as I said before, if $B(1, 5, \overset{s}{0})$ is 'TRUE' it will mean that the character $\overset{s}{0}$ is written in position $(1, 5)$ in the computation matrix.

Now we want to represent the computation of the TM, and for us the computation is the templates. The template

	d	
a	b	c

 is saying that if there is a row j in the computation matrix with an ' a ' in position $i - 1$, a ' b ' in position i and a ' c ' in position $i + 1$, then there should better be a ' d ' in position i of the next row $j + 1$.

We represent the same information in logic by using a "if-then clause". We include the clause $(B(i - 1, j, a) \wedge B(i, j, b) \wedge B(i + 1, j, c) \Rightarrow B(i, j + 1, d))$ in the set C of clauses. Of course in an actual clause of SAT there are only allowed to be OR-operators between the variables, but we can get rid of 'implication' and 'and' by using the following rule from logic: $(u \wedge v \wedge W) \Rightarrow z \equiv \neg u \vee \neg v \vee \neg w \vee z$.

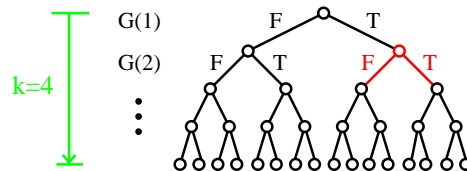
Each such if-then clause is saying: If these three characters are seen in the row below, then this character should better be seen in the row up. And we do this for every i from 2 to $2k$, and for every j from 1 to k . So we will have quite many clauses for each template, but there will be polynomially many only all together, because a polynomial times a polynomial is still a polynomial.

IN210 – lecture 6



non-determinism \mapsto “choice” variables

Ex.



$G(t)$ tells us what non-deterministic choice was taken by the machine at step t . We extend the “if-then clauses” with k choice variables:

$$(G(t) \wedge \text{“a”} \wedge \text{“b”} \wedge \text{“c”} \Rightarrow \text{“d”}) \vee (\neg G(t) \wedge \dots)$$

Note: We assume a **canonical NTM** which

- has exactly 2 choices for each (state, scanned symbol)-pair.
- halts (if it does) after exactly k steps.

So we are able to translate squares, input and TM rules into logic. The last important piece is non-determinism. We have been dealing with a deterministic Turing machine in our example, but non-determinism is essential because we know that the input machine to BOUNDED HALTING is non-deterministic so this kind of idea doesn't really work. But we know that we can turn non-determinism into determinism by guessing which path the NTM will follow. So we introduce the guess variables.

Imaging for a moment that our Turing machine always has exactly two non-deterministic choices – not four, not one, but two. Imagine also that if it halts, then it does so after exactly k steps. We say that the machine is canonical. The assumption that the machine is canonical does not restrict generality in any way because it is easy to turn an arbitrary machine into a canonical one by inventing some new states and some new things, and it can be done in polynomial time. These are technicalities.

So for a canonical machine non-deterministic choices amount to choosing one of the two binary values. We can say '0' or '1', or we can call them 'TRUE' or 'FALSE'. So if we are interested in a computation which has length k , then we introduce k $G(t)$ -variables which tell us for every step t what non-deterministic choice was taken by the machine in that step. If we have these guesses, then we effectively have turned the machine into a deterministic machine, because if we know which of the possible two rules was taken at every step, those rules themselves they are deterministic things.

So given all the values of $G(t)$ where t ranges from 1 to k , everything is deter-

ministic. We extend the "if-then clauses" with choice variables:

$$(G(t) \wedge B(i-1, j, a) \wedge B(i, j, b) \wedge B(i+1, j, c) \Rightarrow B(i, j+1, d) \vee (\neg G(t) \wedge B(i-1, j, a) \wedge B(i, j, b) \wedge B(i+1, j, c) \Rightarrow B(i, j+1, e))$$

The meaning of this is that if choice variable $G(t)$ has truth value 'TRUE', then we choose the first rule, otherwise we choose the second one.

For each rule template we create such clauses for all t 's from 1 to k and all i 's from 2 to $2k$ and all j 's from 1 to k . It will be a lot of them, but only polynomially many in the size of the BOUNDED HALTING input. We also have to deal with the fact that there is not one, but several templates corresponding to the same rule, but that is just a technicality. It can be done.

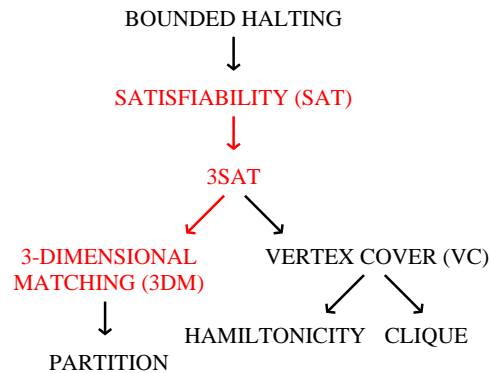
This completes the proof, which is really only a sketch. If we do all these translations, then we will assure that instance of BOUNDED HALTING is a 'Yes'-instance if and only if the translated SAT-instance is a 'Yes'-instance. And the translation – or reduction – can be done in time polynomial in the size of the input $(M, x, 1^k)$ to BOUNDED HALTING.

Since BOUNDED HALTING is \mathcal{NP} -complete and polynomial-time reducible to SAT, that means that SAT is also \mathcal{NP} -hard. Because if SAT can be solved in polynomial time, then so can BOUNDED HALTING and every other problem in \mathcal{NP} .

IN210 – lecture 6



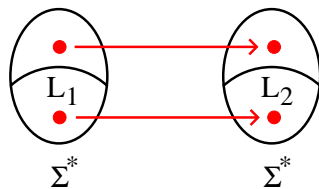
Further (basic) reductions



Polynomial-time reductions (review)

$L_1 \propto L_2$ means that

- $R : \Sigma^* \rightarrow \Sigma^*$ such that
 - $x \in L_1 \Rightarrow f_R(x) \in L_2$ and
 - $x \notin L_1 \Rightarrow f_R(x) \notin L_2$



- $R \in P_f$, i.e. $R(x)$ is polynomial computable

Autumn 1999

9 of 15

So by Cook or by crook we have now established as a fact that SAT is \mathcal{NP} -complete. We can use that fact to prove that all sorts of other problems are \mathcal{NP} -complete, and that is what we are going to do. And by doing so we are going to learn about those problems and about the techniques used to prove \mathcal{NP} -completeness, and we will gain all sorts of insights. We will understand in the end that all \mathcal{NP} -complete problems are really just variants of each other, even though when you meet them in the street, you will never think that they belong together; they are so different.

On this foil I am showing some basic, practical \mathcal{NP} -complete problems. The arrows show what polynomial-time reductions we will use to prove them \mathcal{NP} -complete.

Remember that a polynomial-time reduction R from L_1 to L_2 means two things: One, that all strings x in L_1 are mapped to strings in L_2 , in such a way that 'Yes'-instances in L_1 are mapped to 'Yes'-instances in L_2 and 'No'-instances in L_1 are mapped to 'No'-instances in L_2 . And two, that the reduction can be done (computed) in polynomial time in the size of the input x .

(This is a blank page)

IN210 – lecture 6

SATISFIABILITY \propto 3-SATISFIABILITY

SAT

Clauses with any
number of literals



3SAT

Clauses with
exactly 3 literals

- C_j is the j 'th SAT-clause, and C_j' is the corresponding 3SAT-clauses.
- y_j are new, fresh variables, only used in C_j' .

 C_j $(x_1 \vee x_2 \vee x_3) \mapsto$ C_j' $(x_1 \vee x_2 \vee x_3)$ $(x_1 \vee x_2) \mapsto (x_1 \vee x_2 \vee y_j), (x_1 \vee x_2 \vee \neg y_j)$

$$(x_1) \mapsto (x_1 \vee y_j^1 \vee y_j^2), (x_1 \vee \neg y_j^1 \vee y_j^2),$$

$$(x_1 \vee y_j^1 \vee \neg y_j^2), (x_1 \vee \neg y_j^1 \vee \neg y_j^2)$$

$$(x_1 \vee \dots \vee x_8) \mapsto (x_1 \vee x_2 \vee y_j^1), (\neg y_j^1 \vee x_3 \vee y_j^2),$$

$$(\neg y_j^2 \vee x_4 \vee y_j^3), (\neg y_j^3 \vee x_5 \vee y_j^4),$$

$$(\neg y_j^4 \vee x_6 \vee y_j^5), (\neg y_j^5 \vee x_7 \vee x_8)$$

Question: Why is this a proper reduction?

Autumn 1999

10 of 15

The next problem we will prove \mathcal{NP} -complete is a kind of a restricted version of SAT, called 3SAT. We want to prove that 3SAT is in \mathcal{NPC} because it turns out that 3SAT is easier to use as the \mathcal{NPC} problem in other proofs.

3SAT has exactly three literals in each clause. This is just a restricted version of SAT, so of course 3SAT is in \mathcal{NP} . The second part of the proof is reducing SAT to 3SAT – showing that given an 'Yes'-instance ('No'-instance) of SAT we can create an 'Yes'-instance ('No'-instance) of 3SAT in polynomial time.

We will do this reduction by introducing some new variables. And as always when I am doing these reductions, I am just showing you what to do, waving my arms a little. But you want to verify that what I am doing can actually be done by an algorithm. That is the essential part.

Here is what the algorithm is doing: It will be turning the SAT clauses into 3SAT clauses, one by one. This is an example of a technique which in your G&J textbook is called *local replacement*.

If the original SAT clause happens to have three literals already, there is nothing to do. You just copy it. If the SAT clause have two literals, then we introduce a fresh y variable, and this variable is indexed j , where j is the "name" of the SAT clause. In this way we secure that y_j exists only in the 3SAT clauses that correspond to the SAT clause being translated. So we don't have to worry about the y_j variable appearing anywhere else.

Given the SAT clause $(x_1 \vee x_2)$ we make the two 3SAT clauses $(x_1 \vee x_2 \vee y_j)$ and $(x_1 \vee x_2 \vee \neg y_j)$. It turns out to be irrelevant what truth value we choose to assign to the variable y_j . Because no truth assignment to variable y_j can make both of these clauses be 'TRUE'. They are both 'TRUE' if and only if at least one of x_1 or x_2 are 'TRUE'. Which is exactly the meaning of the SAT clause $(x_1 \vee x_2)$.

So we have translated $(x_1 \vee x_2)$ into two 3SAT clauses which say exactly the

same: One of x_1 or x_2 must be set to 'TRUE' for the two clauses in C'_j to be 'TRUE'. Remember that we use OR within the clause, and AND among the clauses.

If we have only one literal in the clause, then we play the same game. Now we need two additional variables, y_j^1 and y_j^2 , and all their combinations. So we end up with four clauses as shown on the foil. But since we have all the combinations of these y -variables, then they really make no difference. Because no truth assignment to y_j^1 and y_j^2 can make all four of these clauses 'TRUE'. In exactly one of the four clauses both literals with y -variables will be 'FALSE', and therefore the truth value of that particular clause will depend on x_1 alone – which was the meaning of the original SAT clause (x_1).

If x_1 is 'TRUE', then of course all four clauses in C'_j are 'TRUE'. If x_1 is 'FALSE', then at least one of the four clauses must be 'FALSE', whatever the values of y_j^1 and y_j^2 are, since we have all combinations of y_j^1 and y_j^2 and their negations in combination with x_1 which is now 'FALSE'.

If the clause C_j happens to have more than three literals, then we introduce the y_j -variables in a slightly different way: Suppose that C_j has 8 literals, x_1, \dots, x_8 . In 3SAT we will then have 6 clauses that corresponds to C_j . The first clause will be (x_1, x_2, y_j^1) , the second is $(\neg y_j^1, x_3, y_j^2)$ and the third is $(\neg y_j^2, x_4, y_j^3)$. You can see the pattern now, right? The last three clauses are $(\neg y_j^3, x_5, y_j^4)$, $(\neg y_j^4, x_6, y_j^5)$ and $(\neg y_j^5, x_7, x_8)$.

These y_j 's they sort of connect the x -variables into a chain. What is the idea? C_j is saying that at least one of the x_1 to x_8 must be 'TRUE' in order for C_j to be 'TRUE' or satisfied. I claim that these 6 clauses in C'_j they say exactly the same: at least one of x_1 to x_8 must be 'TRUE' in order for all 6 clauses to be satisfied.

This is a little subtle, but also extremely important, because it illustrates what we mean by a reduction. Remember that a reduction must map 'Yes' to 'Yes' and 'No' to 'No'. So we must prove that if C_j is satisfiable then all 6 clauses in C'_j are satisfiable. But we must also prove that if C_j is not satisfiable, then it is impossible to satisfy all 6 clauses in C'_j .

Let's look at 'Yes' to 'Yes' first: C_j is satisfied if at least one of x_1 to x_8 is satisfied. If x_1 or x_2 is satisfied, then we can satisfy all clauses in C'_j by setting all the y_j 's variables to 'FALSE'. If x_3 is 'TRUE', then we set y_j^1 to 'TRUE' and all other y_j 's to 'FALSE'. If x_4 is 'TRUE', then we set y_j^1 and y_j^2 to 'TRUE' and all other y_j 's to 'FALSE'. So the pattern is clear: If x_i is 'TRUE', then we can satisfy all 6 clauses by setting y_j^i up to y_j^{i-2} to 'TRUE' and the other y_j 's to 'FALSE'. This means that if x_7 or x_8 are 'TRUE', then we set all y_j 's to 'TRUE'. You should use some minutes at home to convince yourself that this argument is indeed a valid one.

So we have shown that if any of the x 's are 'TRUE' – meaning that C_j is satisfiable – then there is a truth assignment to the y_j 's so that C'_j is also satisfied.

Now we have to do the opposite. We have to prove that if all x 's are 'FALSE' – which means that C_j is not satisfiable – then there is no way to satisfy all 6 clauses in C'_j . This is how we prove that:

Since x_1 and x_2 are both 'FALSE', then for the first clause in C'_j to be 'TRUE', y_j^1 must be 'TRUE'. Since y_j^1 is 'TRUE', then $\neg y_j^1$ in the second clause is 'FALSE'. But x_3 is also 'FALSE', so for the second clause to be satisfied, we need to set y_j^2 to 'TRUE'. The same pattern repeats itself for the third, fourth and fifth clause: Because x_4 , x_5 and x_6 are all 'FALSE', y_j^3 , y_j^4 and y_j^5 need to be 'TRUE'. But then we have a problem with the last clause: $\neg y_j^5$ is 'FALSE' because y_j^5 is 'TRUE', but x_7 and x_8 are also both 'FALSE', so the last clause cannot be satisfied. All together this amounts to a proof that if C_j is not satisfiable, then neither is C'_j .

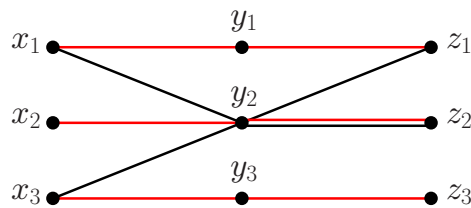
This concludes the reduction from SAT to 3SAT. You should convince yourself that this is a proper reduction, meaning that it can be computed in polynomial time. All together this amounts to a proof that 3SAT is \mathcal{NP} -complete.

IN210 – lecture 6

**3-DIMENSIONAL MATCHING (3DM)**

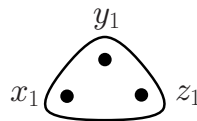
Instance: A set M of triples (a, b, c) such that $a \in A, b \in B, c \in C$. All 3 sets have the same size q ($|A| = |B| = |C| = q$).

Question: Is there a **matching in M** , i.e. a subset $M' \subseteq M$ such that every element of A, B and C is part of exactly 1 triple in M' ?

Example

$$M = \{(x_1, y_1, z_1), (x_1, y_2, z_2), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_3, y_2, z_1)\}$$

We will use sets with 3 elements to visualize triples:



So we have that 3SAT is \mathcal{NP} -complete. We are moving on to the next problem which is the 3-DIMENSIONAL MATCHING problem. This will take us from the world of logic into something which is a set-theoretical problem or a matching problem or a hypergraph problem. We can also call it a kind of a very practical problem these days, because it is a variant of the stable-marriage problem or the matching problem, which is the first problem we have seen.

This time we are trying to marry three things as opposed to two. You can imagine that we are dealing with modern couples and modern families, where not only people are involved but automobiles also. You can marry two people – all right they love each other, but if they happen to like different kind of cars, then they should not be married together.

There are boys, girls and cars – three things – and compatibility constraints among them. So the instance is a set of triples, each element of a triple belonging to different sets. We have triples of the form (a, b, c) such that ' a ' belongs to set A , ' b ' belongs to set B and ' c ' belongs to set C . To make the things simpler we assume that these three sets have the same size q . The question we are asking is: Can we marry everybody? Is there a matching in set M such that everybody is married and that nobody is married two times? You know, obviously two families should not own the same automobile, and no woman or man should belong to two families, and so on.

So this is a natural generalization of the 2-DIMENSIONAL MATCHING problem which we have seen in the first class. The interesting thing here is that while 2-DIMENSIONAL MATCHING has been solved by a polynomial-time algorithm,

3DM is \mathcal{NP} -complete – meaning that it has not been solved by polynomial-time algorithms and it is very unlikely that it will be.

I am showing here on this foil an example of 3DM just to make the things a little more transparent. We have these three sets here – the x 's, the y 's and the z 's – and q is equal to 3, meaning that every set has 3 elements. We have a 5 triples connecting the x 's, y 's and z 's together. Three of them I have colored red. Those three are the stable matching. They are the solution, the marriage. So this is a positive instance.

It turns out to be very easy to show that 3DM is in \mathcal{NP} . The certificate is the stable matching itself, the three red triples. So if a NTM can guess the these three choices, then how much time is required for verifying that they are actually the stable marriage, the solution? Almost no time at all – you just verify that the three triples are in M , the set of triples, and that each x_i , each y_i and each z_i have all been repeated exactly once in the three triples.

You can program this up in no time and you can see that the algorithm is easy and efficient – no exhaustive search, no difficult things, all is polynomial. You need a little bit of practice though to understand really what is polynomial and what is not. This practice you will get from your group exercises.

IN210 – lecture 6



Reductions are like translations from one language to another. The same properties must be expressed.

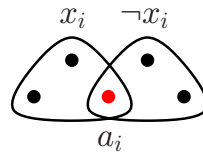
3SAT \propto 3DM

3SAT	\longleftrightarrow	3DM
variables x_1, \dots, x_n	\longleftrightarrow	variables $x_3^j, a_3^j, b_j^2, c_k^1$
literals $x_1, \neg x_1$	\longleftrightarrow	variables $x_1^j, \neg x_1^j$
clauses	\longleftrightarrow	triples (x_1^j, b_j^1, b_j^2) $(\neg x_3^j, b_j^1, b_j^2)$
$C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$		
"There exists a sat. truth assignment"	\longleftrightarrow	"There is a matching"

"There is a truth assignment T "

- $\exists T : \{x_1, \dots, x_n\} \rightarrow \{\text{TRUE}, \text{FALSE}\}$
- $T(x_i) = \text{TRUE} \Leftrightarrow T(\neg x_i) = \text{FALSE}$

The second property is easily translated to the 3DM-world:



$T(x_i) = \text{TRUE} \longleftrightarrow x_i$ is not "married"

We now move on to show a reduction from 3SAT to 3DM. This is going to be a little bit involved, because this is an example of the most difficult kind of reduction, which is asking for creating certain components, which represents certain properties of language we are reducing from. Your G&J textbook calls this kind of technique *component design*.

If we are reading a textbook where the reduction is already done, then we just read and say: "OK, this looks nice and they seem to know what they are doing." But we will assume that we are dealing with a fresh problem, because we are also trying to learn here in this class how actually one would apply these techniques to a completely new problem. We are trying to build up some intuition, to create some intuitive understanding of how one goes about proving \mathcal{NP} -completeness. The technical details of the proofs can be found in your G&J textbook.

So, first of all we need a \mathcal{NP} -complete problem to reduce to 3DM. 3SAT is a natural choice because it is also a '3' problem. They kind of seem similar. So a little bit of intuition helps to see what problem should be reduced to our new problem. Another reason for choosing 3SAT is that it's basically the only \mathcal{NP} -complete problem we know yet.

We are reducing SAT to 3DM. We are translating from one language to another. In the language of 3SAT an instance is encoded in terms of variables, literals and clauses. In the language of 3DM we have variables and triples. So we see right away a certain similarity which allows us to translate things into one another:

In 3SAT we have variables, in 3DM we have also variables. But we don't have literals in 3DM. That could be a problem. What about the clauses? They seem similar to triples somehow – you have 3 things in clauses and you have 3 things in triples. The elements of the clauses are literals, so it is natural then to create variables in 3DM that are also literals. So we will represent 3SAT variables and literals by 3DM variables, while clauses will become triples.

There are also several properties that need to translate from the world of 3SAT to the world of 3DM. The main property, the crucial fact which we need to translate is: "There exists a satisfying truth assignment." This is completely central, because it is this property that distinguishes the negative instances of 3SAT from the positive instances of 3SAT. The corresponding property in 3DM is: "There exists a matching."

We must translate this property in such a way that if the 3SAT instance has a satisfying truth assignment, then the reduced 3DM-instance has a (perfect) matching. Because we want to map 'Yes'-instances to 'Yes'-instances and 'No'-instances to 'No'-instances. We must also ensure that the reduction, the translation, can be done in polynomial time.

First of all we need a truth assignment – we need a choice of 'TRUE' and 'FALSE' to the logical variables. We want to translate this notion of truth assignment to logical variables, into the language of 3DM where we don't have logical variables or truth assignments, but only choices of triples.

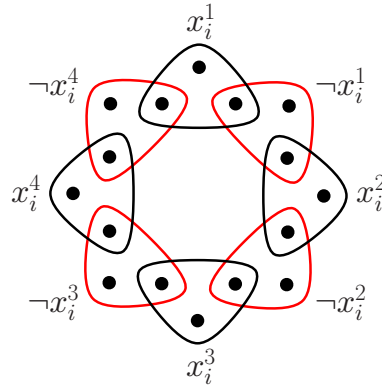
So we are doing something which is called component design. We are creating a kind of a monster, a bunch of 3DM-triples which are going to express the same fact as a truth assignment would express in 3SAT.

"There is a truth assignment T" means two things: One, every logical variable is assigned the value 'TRUE' or 'FALSE'. Two, if x_i is 'TRUE' then $\neg x_i$ is 'FALSE' and vice versa. The second fact is translated to the 3DM world by introducing a new variable a_i which is only used in two triples – one triple where x_i is a member and one triple where $\neg x_i$ is a member. Since every 3DM variable must be married in a positive instance, then one of the two triples containing a_i must be chosen. If the triple containing x_i is chosen, that means that x_i is already used – it is married. This will correspond to SATvariable x_i being assigned the value 'FALSE'. If on the other hand the triple containing $\neg x_i$ is chosen, then x_i is free for use later in the reduction. This will correspond to SATvariable x_i being 'TRUE'.

IN210 – lecture 6



A literal x_i can be used in many clauses. In 3DM we must have as many copies of x_i as there are clauses:



- Either all the black triples must be chosen (“married”) or all the red ones!
- If $T(x_i) = \text{TRUE}$ then we choose all the red triples, and the black copies of x_i are free to be used later in the reduction. And vice versa.
- We make one such **truth setting component** for each variable x_i in 3SAT.

Autumn 1999

13 of 15

A 3SAT literal can be used in many clauses, but a 3DM variable can only be chosen (married) once. So we need as many copies of x_i and $\neg x_i$ as there are clauses in the 3SAT-instance. Of course, not all clauses will contain the literal x_i , but we will do some cleaning up in the end that will take care of unused copies.

We need to assure that either all the x_i copies are married or all the $\neg x_i$ copies, because a variable is either 'TRUE' or 'FALSE'. Therefore we make a huge component of all the x_i and $\neg x_i$ copies as shown on the foil (we assume there are 4 clauses). The 8 anonymous "inner" variables are fresh variables not used anywhere else, and because all 8 want to be married exactly once, then it is easy to see that either all black triples must be chosen or all the red ones.

If the truth assignment has made x_i 'TRUE', then we choose all the red triples, so that all the black copies of x_i are free to be used later in the reduction. But if on the other x_i is 'FALSE', then we choose all the black triples, so that the $\neg x_i$ are available – free to be married later.

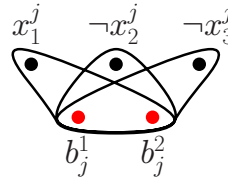
We must make one such truth-setting component for each variable x_i in the 3SAT-instance, but there will be only polynomially many and each of them will be polynomially large, so everything is fine. These truth-setting gadgets take care of the variables and of their truth assignments.

IN210 – lecture 6



“ T is satisfying”

We translate each clause (example:
 $C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$) into 3 triples:



- b_j^1 and b_j^2 can be married if and only if at least one of the literals in C_j is not married in the truth setting component.
- If we have a satisfiable 3SAT-instance, then all b_j^1 and b_j^2 -variables ($1 \leq j \leq m$) can be married.
- If we have a negative 3SAT-instance, then some b_j^1 and b_j^2 -variables will not be married.

Autumn 1999

14 of 15

Now we need to express the fact that the truth assignment is a *satisfying* truth assignment. This means that every clause contains at least one literal which is 'TRUE'. How do we represent the clauses? By triples, of course. Given a clause $C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$ we construct 3 triples as shown on the foil, using two fresh b_j variables – 'fresh' again meaning that they are only used in these tree triples. It is easy to see that b_j^1 and b_j^2 can be married if and only if at least one of the literals in C_j is "single", meaning that it didn't get married in the truth-setting part. This will correspond to the fact that at least one of the literals in clause C_j has been assigned truth value 'TRUE'.

So we have expressed the fact that one of the literals in a clause must be 'TRUE', and if we construct 3 triples as on the foil for every clause in the 3SAT-instance, then we know that in a stable marriage every clause must be satisfied.

This is the heart of the reduction: The 3SAT-instance is satisfiable if and only if all b_j^1 and b_j^2 variables can be married. Let's see why this is a valid claim: Suppose that we have a 3SAT-instance with n variables and M clauses. If the 3SAT-instance is satisfiable then we choose triples in the truth-setting component in such a way that every literal that has been assigned the value 'TRUE' by the satisfying truth assignment, is available at this point. Since a satisfying truth assignment means that at least one literal in each clause C_j is 'TRUE', that literal is available for marriage with b_j^1 and b_j^2 . So all b_j^1 and b_j^2 can be married.

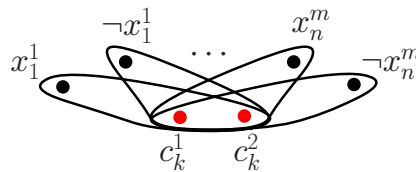
On the other hand, if the 3SAT-instance is not satisfiable, then no matter how we choose triples in the truth-setting component, at least one b_j^1 and b_j^2 can not be married.

IN210 – lecture 6



Cleaning up (“Garbage collection”)

There are many x_i^j who are neither married in the truth setting components nor in the “clause-satisfying” part. We introduce a number of fresh c -variables who can marry “everybody”:



- There are $m \times n$ unmarried x -variables after the truth setting part.
- If all m clauses are satisfiable then there will remain $(m \times n) - m = m(n - 1)$ unmarried x -variables.
- So we let $1 \leq k \leq m(n - 1)$.

Autumn 1999

15 of 15

So now we are basically finished. We have translated the essential properties of a positive 3SAT-instance into the language of 3DM and into properties that a positive instance of 3DM has to satisfy.

It remains to do a little cleaning up or "garbage collection". Even if we have a satisfying truth assignment, there will be lots of x_i^j variables that neither got married in the truth-setting component nor in the clause-satisfying part. We introduce a number of fresh c -variables to deal with those, as shown on the foil.

How many c -variables do we need? Let's do a little bit of counting: Given M clauses and n variables, we will make m copies of each of the n x_j literals and m copies of each of the n $\neg x_j$ literals. The satisfying truth assignment marries half of them, but there will still be $m \times n$ unmarried ones. If all clauses are satisfiable, then m of them will be used to marry the b -variables. That leaves $(m \times n) - m = m(n - 1)$ unmarried x -variables.

So if we have a satisfying 3SAT-instance, then we need $m(n - 1)$ c_k^1 -variables and $m(n - 1)$ c_k^2 -variables to ensure that there is a perfect matching in the corresponding 3DM-instance. We don't need to care about the case of having a negative 3SAT-instance, because then we cannot marry all the b_j 's anyway – as explained earlier – so there will be no matching.

This completes the reduction. To formally prove that what we have now sketched is the proper reduction that we need, would amount to proving two things: One is that this is a reduction, namely that it maps positive instances to positive instances, and negative instances to negative instances. And the other part is that this can be done by an algorithm in time polynomial in the size of

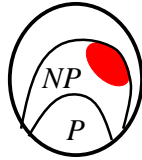
the 3SAT-instance. More details are given in the G&J textbook.

3.7 Lecture 7

IN210 – lecture 7



Review



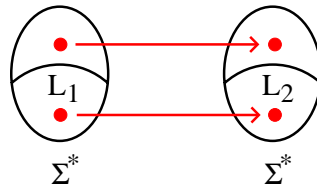
- 'hardest' problems in \mathcal{NP}
- all efficiently reducible to one another

● NP-complete

Polynomial-time reductions

$L_1 \propto L_2$ means that

- $R : \Sigma^* \rightarrow \Sigma^*$ such that
 - $x \in L_1 \Rightarrow f_R(x) \in L_2$ and
 - $x \notin L_1 \Rightarrow f_R(x) \notin L_2$



- $R \in P_f$, i.e. $R(x)$ is polynomial computable

G&J:

3.1.3-3.1.5, 3.2

Autumn 1999

1 of 15

So, what are we doing? We are studying \mathcal{NP} -completeness, and we are doing actually many things at the same time here, but one of the things is: We want to look at the class of problems, and by understanding and studying one class of problems we develop the whole methodology. We develop a whole pile of insights which will help us to deal with all kinds of other classes and problems.

The point is intuitively that we want to bunch together the problems that are kind of similar, that look like a nice family, and somehow we want to be able to prove that those problems really belong together.

So \mathcal{NP} -complete problems will be one such family of problems. And their basic similarity is that they all seem difficult, hard nuts to crack. Many people tried, nobody succeeded to crack anyone of them in the sense of solving them efficiently. So they are all hard in a way, and in a sense they are hardest in the big class \mathcal{NP} . They are all efficiently reducible to one another, meaning that if one can solve any one of those problems efficiently, then all of them are automatically solved efficiently. So they sort of cling together, these problems.

And the key of the technique that we will be using is the notion of the polynomial-time reduction, which will allow us reduce one problem to another – in fact to reduce all problems in class \mathcal{NP} to each problem in class \mathcal{NP} -complete. Now we are studying how to create those reductions.

Proving that a certain language L_1 is polynomial-time reducible to language L_2 amounts to proving two facts. On the foil we show a function which maps the

Sigma-star set to itself, namely all strings to all strings. A reduction is a function which given a string will create another string. Intuitively the first string is an instance of the first problem, and the second is an instance of the second problem. This function has the property that it maps positive instances to positive instances and negative instances to negative instances. So it is a kind of a translation.

So if the right answer to the question asked in language 1 is 'Yes', then the right answer of the instance produced by the reduction R should better be 'Yes' also. And vice versa: If the right answer to the question in L_1 is 'No', then the right answer to the L_2 -instance created by R should also be 'No'. That is what we mean by a reduction.

In addition to that a polynomial-time reduction must be efficiently computable. It is a function computable in polynomial time in the length of input to L_1 . This is important because if the reduction is computable in polynomial time, then if we can answer the question asked in language L_2 efficiently, then by virtue of the fact that the composition of two efficient functions is efficient, we know that the question in L_1 also can be answered efficiently. And, given that the answer to this question L_2 is really the same as the answer to this question L_1 by the first property of the reductions, then we know that if the second language can be solved efficiently, so can the first one.

So this kind of reduction gives us exactly what we want in order to be able to say that if problem 2 is easy so is problem 1. But it also tell us that if problem 1 is difficult then problem 2 is also difficult, because if we on the contrary could solve problem 2 efficiently, then by using the efficient reduction we could also solve the difficult problem 1 efficiently. So problem 2 must be at least as difficult as problem 1.

That is what we are trying to prove. We are trying to prove for new problems that they are difficult. We already have some difficult problems, which are this artificial BOUNDED HALTING problem and the real-world problems SATISFIABILITY and its reduced version 3SAT and 3-DIMENSIONAL MATCHING.

IN210 – lecture 7

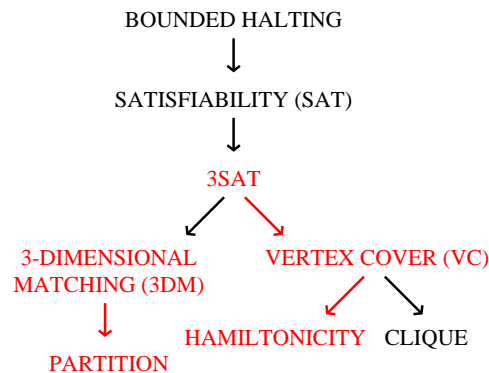


Proving L is \mathcal{NP} -complete

- $L \in \mathcal{NP}$: show a short “ticket” (succinct certificate)
- $L \in \mathcal{NP}$ -hard: show a poly-time reduction from a known \mathcal{NP} -hard problem.

For (decision versions of) search problems $L \in \mathcal{NP}$ is usually easy because the solution is the certificate.

Further (basic) reductions



Autumn 1999

2 of 15

Proving \mathcal{NP} -completeness consists of proving two facts. One, that your language is in \mathcal{NP} . Usually this is easy, because it amounts to just showing that there is a "short" ticket. And if we are talking about decision versions of search problems, which is usually what we are doing, then the ticket is most often the solution. So if we are talking about HAMILTONICITY, the ticket is the Hamiltonian path, if we are talking about SAT, the ticket is a satisfying truth assignment, and so on.

The fact that L is in \mathcal{NP} -hard, which is the second fact we have to prove for \mathcal{NP} -completeness, amounts to showing a reduction from a known \mathcal{NP} -hard problem, which effectively means that your problem is at least as hard as your original \mathcal{NP} -hard problem, by virtue of the property of these reductions.

So this is the technique, and what we want to do now is we want to apply this technique to various problems and understand both the problems and the technique. Today we will show three more problems \mathcal{NP} -complete, namely PARTITION, VERTEX COVER and HAMILTONICITY.

(This is a blank page)

IN210 – lecture 7

**PARTITION**

Instance: A finite set A and sizes $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

Question: Can we **partition** the set into two sets that have equal size, i.e. is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$$

3DM \propto PARTITION

We first reduce 3DM to SUBSET SUM where we are given A , as in PARTITION, but also a number B , and where we are asked if it is possible to choose a subset of A with sizes that add up to B .

3DM		SUBSET SUM
sets and		
triples (subsets)	\mapsto	numbers
“There is a matching M ”	\mapsto	“There is a subset with total size B ”

Autumn 1999

3 of 15

The two real-world \mathcal{NP} -complete problems that we have seen so far – SATISFIABILITY and 3-DIMENSIONAL MATCHING – they are rather similar. They are both talking about variables and things. Now we move from variables to numbers. We are going to see a \mathcal{NP} -complete problem which at the outset does not look at all similar to the problems that we have seen so far.

Our new problem is called PARTITION and it is a number problem. We are given a finite set A , and for each element of A we are given a size. So ‘size’ is a positive integer. And then the question about this bunch of positive integers is: Is there a subset A' of A such that the sum of sizes of all elements in A' is the same as the sum of sizes of the remaining elements? In other words: Can we partition the set into two sets that have equal size?

Since the last problem we have been looking at was the stable marriage problem, we can talk here about the stable divorce problem. You are given a bunch of objects in the household and you are divorcing. You don’t want to live with your wife or roommate any more. And you want to divide everything so that everybody is happy, meaning that both persons get the same value in the end. Can you do that or not, that is the question.

Ideally you would like to have a computer program that does this because divorces are very frequent these days and people go through it many times. You don’t want to spend the rest of your life dividing things. You just want to run the program and be done with the whole thing and then begin a new life.

We are going to see how to reduce the natural stable marriage to stable di-

voiced – 3DM to PARTITION. And this is surprising because these two languages seem completely different – in 3DM you have triples of variables and in PARTITION you have numbers.

We will first reduce 3DM to a variant of PARTITION called SUBSET SUM. In SUBSET SUM we are given a set of numbers A – as in PARTITION – but also a number B , and we are asking: Is it possible to choose a subset of A with sizes that add up to B ?

We know that a reduction is like translating from one language to another – the same properties must be expressed and also the basic building stones which an instance is made of. In the world of 3DM an instance consists of three sets W , Y and Z of equal size and a set M of triples of elements drawn from those three sets. Somehow we need to translate these objects into numbers. The basic property of a positive 3DM-instance is: "There is a matching M' ". This has to be translated into the corresponding SUBSET SUM-property: "There is a subset with total size B ."

IN210 – lecture 7



Difficulty: We need to translate from subsets with 3 elements (triples) to numbers.



Solution: Use the **characteristic function** of a set!

Example

Given set $U = \{x_1, x_2, \dots, x_n\}$ and subset $S = \{x_1, x_3, x_4\}$. The characteristic function of S is a binary number with n digits and bit 1, 3 and 4 set to 1: $\underbrace{101100 \dots 0}_n$.

There is a matching $M' \leftrightarrow$ There is a subset M'
 $\sum_{M'} \text{sizes} = B$

It is natural to set $B = \underbrace{111 \dots 11}_n$, since each element in the universe is used in exactly one of the triples in the matching.

Technicality: Carry bits!

$01_b + 10_b = 11_b$, but also $01_b + 01_b + 01_b = 11_b$.

Autumn 1999

4 of 15

So in a 3DM-instance you have a set M of triples. How can we turn triples, which are subsets consisting of exactly 3 elements, into numbers? That is the basic question. And there is a very natural way of doing it, which has to do with a very basic concept called the *characteristic function of a set*.

So suppose as an example that we are dealing with the finite set consisting of the elements x_1, x_2 up to x_n . So $U = \{x_1, x_2, \dots, x_n\}$ are the universe from which we are going to draw subsets (triples). Let's look at the subset $S = \{x_1, x_3, x_4\}$. How do we represent S as a number?

The natural way to represent sets like S as numbers is to represent the universe as a bunch of binary digits. In our example the universe is mapped into $1, 1, \dots, 1, 1 - n$ of these. Then each subset can be represented by a corresponding binary number of size n , where a '1' in position i means that x_i is a member of the subset. So subset S would become $101100 \dots 0$ (n digits).

So with n binary digits we can represent every possible subset of U as a unique binary number. This is called the characteristic function of a set. This little trick, which is really the most natural way of representing sets by numbers, is all that we need for the reduction in addition to some salt and pepper.

We must also translate the basic 3DM-property "There exists a matching M' ". Given that we can represent each element (triple) in M by a unique n -digits binary number, then the natural corresponding SUBSET SUM property is "There is a subset M' of M such that the sizes (numbers) in M' add up to B ."

What should the number B look like? The natural value for B is $11 \dots 11$ (n of them) since each element in the universe must be used in exactly one of the

triples in M' .

Now we have the basic picture of the reduction. The only difficulty which complicates the solution is the carry bits that can arise when we do addition: For example $01 + 10 = 11$ when added binary, but because of the carry bit, $01 + 01 + 01$ is also 11. In some way or another we must ensure that the i -th digit in the sum is a '1' if only if one of the numbers added had a '1' in place i , and not as result of two '1' at place $i + 1$ being "carried over" to place i .

IN210 – lecture 7

**3DM-instance:**

$$M \subseteq W \times X \times Y$$

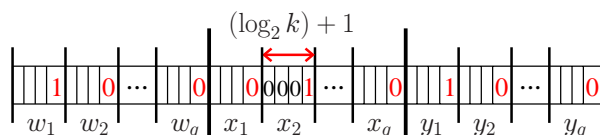
$$W = \{w_1, w_2, \dots, w_q\}$$

$$Y = \{y_1, y_2, \dots, y_q\}$$

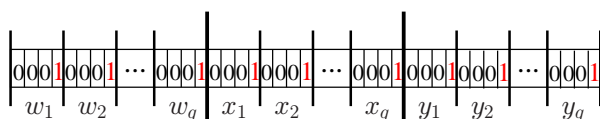
$$Z = \{z_1, z_2, \dots, z_q\}$$

$$M = \{m_1, m_2, \dots, m_k\}$$

- For each triple $m_i \in M$ we construct a binary number:



- This PARTITION/SUBSET SUM number corresponds to the triple (w_1, x_2, y_1) .
- By adding $\log_2 k$ zeros between every "characteristic digit", we eliminate potential summation problems due to overflow / carry bits.
- We make B as follows:



Autumn 1999

5 of 15

Now we start working on the details. We are creating numbers that correspond to triples. Each triple in a 3DM-instance consists of three elements, one from each of the sets W , X and Y . Since each set has q elements the binary number that "characterizes" a triple (w_1, x_2, y_1) needs to have at least $3q$ bits – one bit for each element. But using only $3q$ bits is not enough because of the carry bit problem.

The solution is to add a number '0'-bits between each bit in the characteristic number. Then we get a binary number as shown on the foil. I have indicated what "bit-zone" corresponds to which 3DM-element. We construct such a number for each triple in the 3DM-instance.

It turns out that it is enough to add $\log_2 k$ zeros between each "characteristic digit" to eliminate the carry bit problem, where k is the number of triples in the 3DM-instance. Why is $\log_2 k$ zeros enough? Because then we will have $\log_2 k + 1$ bits available for each characteristic digit. There will only be k numbers in the SUBSET SUM instance and even if, say, element x_2 is used in all k triples, $1 + 1 + 1 \dots$ added k times is only a $\log_2 k + 1$ digit binary number. So there will never be a overflow to the characteristic digit representing x_1 .

We also needs to add $\log_2 k$ zeros between each '1' in B – the sum we ask for. That is shown on the foil.

This is the end of the reduction to SUBSET SUM. If the 3DM-instance is a positive instance, then there is a set of triples – a matching – such that each element in X , Y and Z is used exactly once. But that means that if we sum up all the numbers corresponding to triples in the matching, the sum would equal

B , because every characteristic digit is added one and only one time.

On the other hand, if the 3DM-instance is a negative instance, then there is no matching such that each element is in exactly one triple. There is at least one element which are not used in any triple. But then there is also no way to make a subset sum which equals B , because there will be at least one characteristic digit missing, namely the digit corresponding to the element which is not in the matching.

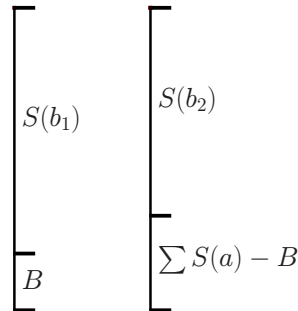
So the reduction is a proper reduction. We also need to prove that the reduction is polynomial-time computable. I won't do the argument here, but just mention that the crucial point is the length of the characteristic numbers, which has to be polynomial in the length of the 3DM input. You should convince yourself that length is indeed polynomial.

IN210 – lecture 7



SUBSET SUM \propto PARTITION

- We introduce two new elements b_1 and b_2 .
- We choose $s(b_1)$ and $s(b_2)$ so big that every partition into two equal halves must have $s(b_1)$ in one half and $s(b_2)$ in the other.



- We let $s(b_1) + B = s(b_2) + (\sum s(a) - B)$.
- We can pick a subset of A which adds up to B if and only if we can split $A \cup \{b_1, b_2\}$ into two equal halves.

Autumn 1999

6 of 15

The final little technicality is that we have now reduced 3DM to the SUBSET SUM problem, not to PARTITION. It turns out to be very easy to reduce SUBSET SUM to PARTITION by adding two huge elements b_1 and b_2 to the set. We choose $s(b_1)$ and $s(b_2)$ so big that every partition into two equal halves must have $s(b_1)$ in one half and $s(b_2)$ in the other, because $s(b_1) + s(b_2)$ would be larger than all the other elements added together.

We choose $s(b_1)$ and $s(b_2)$ so that $s(b_1)$ plus B is exactly equal to $s(b_2)$ plus the total sum of the original numbers minus B – see the figure on the foil. If there is a subset which adds up to B , then $s(b_1) + B$ will be equal to $s(b_2) + (\sum s(a) - B)$, so we have a positive PARTITION instance. But if there is no subset with sum equal B , then we cannot make two equal partitions, because we would need a subset sum which equals B to do that.

So it turns out that the problem of deciding whether a bunch of numbers are such that a subset adds up to B , can be reduced to the question whether it is possible to partition those numbers plus $s(b_1)$ and $s(b_2)$ into two equal halves.

(This is a blank page)

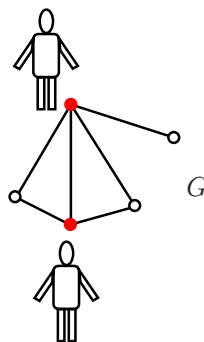
IN210 – lecture 7

**VERTEX COVER (VC)**

Instance: A graph G with a set of vertices V and a set of edges E , and an integer $K \leq |V|$.

Question: Is there a **vertex cover** of G of size $\leq K$?

“Can we place guards on at most K of the intersections (vertices) such that all the streets (edges) are surveyed?”



Autumn 1999

7 of 15

As you might have understood I am not really proving \mathcal{NP} -completeness here. I am just giving you insights and telling you about high-level ideas so that you can understand the kind of thinking that goes in proving \mathcal{NP} -completeness, or understanding \mathcal{NP} -completeness proofs. But if you are asked to do a proof, then the proof should be honest-to-God proof with all the elements of a proof. So you would need to prove A, B, C and D. This you will see more of in the groups and in your G&J textbook. The G&J textbook contains formal \mathcal{NP} -completeness proofs for the problems we have dealt with on a high level here in class. Hopefully the lecture together with everything else will be a whole, allowing you to prove \mathcal{NP} -completeness and understand the proofs.

On the highest level we are seeing different kind of problems and how they can be turned into one another, and the basic ideas and issues that are involved. So we have seen how set problems and the problem of choosing subsets from a set, can be turned into number problems. We have seen also how logic problems, involving truth- and falsehood, can be turned into a kind of a marriage problem or choosing subsets.

Now we are going to see how a logic problem can be turned into a graph problem. VERTEX COVER is a graph problem and to introduce it, imagine that we are given a graph and that that graph actually is a part of a map of a city where roads are edges and intersections are vertices of the graph. We are asked to place guards on the intersections so that they can survey (look after) all the streets.

And the question asked in VERTEX COVER is: "Is it enough to use K guards?"

Can we survey all the streets with K guards or less?" You know, the police department is interested in this question because they want to survey all the streets, but they don't want to pay too many people.

In the little toy instance on the foil, the question is: Given this graph – this little city – are two policemen enough or not? And the answer is 'Yes', so this is a positive instance. With K equal 2 and this graph, there is a way to place K equals 2 guards that can survey all the streets. That position of the guards we call the VERTEX COVER, because we are covering all the edges by choosing these two red vertices here. So in a graph-theoretic language we would ask the question: Is it possible to choose a subset of the vertices such that the subset has K elements or less and every edge has an endpoint in at least one the elements of the subset?

IN210 – lecture 7

3SAT \propto VC

3SAT

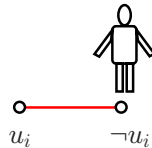
literals
clauses
"There exists a sat.
truth assignment"

VERTEX COVER

vertices
subgraphs
"There is a VC
of size K "



literals \rightarrow vertices



- A guard must be placed in either u_i or $\neg u_i$ for the street between u_i and $\neg u_i$ to be surveyed.
- If we only allow $|V|$ guards to be used for all $|V|$ streets of this kind, then we cannot place guards at both ends.
- Placing a guard on u_i corresponds to the 3SAT-literal u_i being TRUE.
- Placing a guard on $\neg u_i$ corresponds to the 3SAT-literal $\neg u_i$ being TRUE (and the u_i -variable being assigned to FALSE).

Autumn 1999

8 of 15

As usual the prove that VERTEX COVER is in \mathcal{NP} is trivial, because given K vertices corresponding to the placement of the K guards, we can in no time check that all edges are covered.

The difficult part of the \mathcal{NP} -completeness proof is the reduction from a known \mathcal{NP} -complete problem. We will show a reduction from 3SAT to VERTEX COVER, and again we will look at it as a translation from one language to another. The absolute main idea is to translate the property "There is a satisfying truth assignment" to "There is a vertex cover of size K ". But to do that we need to translate the two building stones of a SAT-instance, namely the variables/literals and the clauses.

We know that a truth assignment assigns each boolean variable the value 'TRUE' or 'FALSE'. How do we express the same fact in the language of VERTEX COVER? It turns out to be very easy. We represent each variable by an edge, such that the two vertices corresponding to the edge, they are marked u_1 and $\neg u_1$. By virtue of the fact that this is an edge we know that a guard must be placed either here at u_1 or at $\neg u_1$, or maybe on both ends, in order for this particular edge to be covered (surveyed).

We make one such component for each variable in the 3SAT-instance. If we then limit the numbers of guards to $|V|$ for all $|V|$ streets of this kind, then we cannot place guards at both ends. We are effectively saying: "OK guy, you can choose which intersection to be at – u_i or $\neg u_i$ – but you cannot be at both".

In this way we ensure that either u_i or $\neg u_i$, but not both, must have the truth value 'TRUE'. Placing a guard on u_i corresponds to the 3SAT-literal u_i being

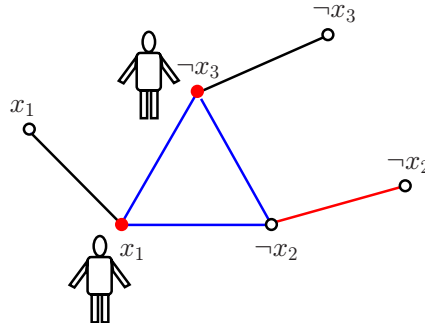
'TRUE'. Placing a guard on $\neg u_i$ corresponds to the 3SAT-literal $\neg u_i$ being 'TRUE'.

So this little trick takes care of the literals and the truth assignment to the logical variables.

IN210 – lecture 7

clause \mapsto subgraph

For clause $C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$ we make the following subgraph:



- We need guards on two of three nodes in the triangle to cover all three (blue) edges.
- If we are allowed to place only two guards per triangle, then we cannot cover all three outgoing edges.
- All 6 edges can be covered if and only if at least one edge (red) is covered from the outside vertex.
- By connecting the subgraph to the “truth-setting” components, this translates to one of the literals being TRUE (guarded)!

Autumn 1999

9 of 15

The second issue is translating the clauses. We translate each clause of 3SAT into a triangle subgraph where each vertex corresponds to a literal in the clause. For clause $C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$ we make the subgraph shown on the foil.

We need guards on two of the three nodes in the triangle to cover all tree edges. But if we were allowed to place only two guards per triangle, then we cannot cover all three outgoing edges, no matter how we place the two guards. There will be at least one edge which has to be covered from its outer endpoint. By connecting the triangle subgraph to the truth-setting components, this translates to the requirement that at least one of literals in the clause being 'TRUE', meaning guarded by a policeman.

So if one of outgoing edges is covered from its outer endpoint, meaning that the corresponding literal is 'TRUE', then we can place the two guards such that they cover the other five edges. But if none of the outgoing edges is covered from its outer endpoint, meaning that all literals in the clause are 'FALSE', then there is no way to cover all six edges with two policemen.

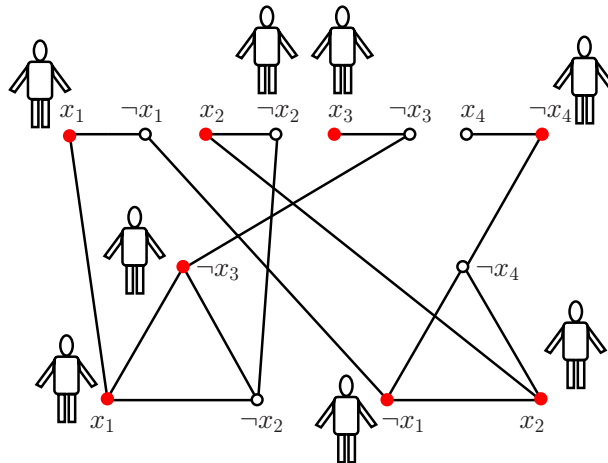
This means that we map positive 3SAT-instances to positive VC-instances, and negative 3SAT-instances to negative VC-instances.

IN210 – lecture 7

**Example****3SAT-instance:**

$$U = \{x_1, x_2, x_3, x_4\} \quad (n = 4)$$

$$C = \{\{x_1, \neg x_2, \neg x_3\}, \{\neg x_1, x_2, \neg x_4\}\} \quad (m = 2)$$



- Total number of guards $K = n + 2m = 8$.
- Should check that the reduction can be computed in time polynomial in the length of the 3SAT-instance ...

Autumn 1999

10 of 15

I will be putting it all together by showing you an example. Given four 3SAT-variables x_1, x_2, x_3, x_4 and the two clauses $\{x_1, \neg x_2, \neg x_3\}$ and $\{\neg x_1, x_2, \neg x_4\}$ we construct the following VC-instance. We need four guards in order to cover the truth-setting edges, and two times two guards for covering the clause-triangles. This means that K – maximum number of guards allowed – should be 8. In general K is $n + 2m$ where n is the number of boolean variables and M the number of clauses.

We get a satisfying truth assignment by setting, for example, x_1 and x_3 to 'TRUE' and x_2 and x_4 to 'FALSE'. It is then easy, as shown on the foil, to cover each triangle gadget by using two guards. So we have a positive VC-instance also.

As usual, we should convince ourselves that the reduction is polynomial-time computable. Here this amounts to showing that given a 3SAT-instance consisting of a bunch of variables and clauses encoded in some alphabet, the reduction algorithm can compute the description of the VC graph – a vertex set and an edge set – in time polynomial in the length of that 3SAT-instance. The crucial point would be to prove that the number of vertices and edges is polynomial in the number of 3SAT variables and clauses.

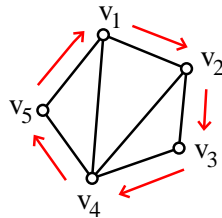
IN210 – lecture 7

**HAMILTONICITY**

Instance: Graph $G = (V, E)$.

Question: Is there a **Hamiltonian cycle/path** in G ?

Is there a “tour” along the edges such that all vertices are visited exactly once? (a Hamiltonian *cycle* requires that we can go back from the last node to the first node)



Autumn 1999

11 of 15

Our last and final reduction is from VERTEX COVER, involving a graph and a number, to HAMILTONICITY, involving only a graph. HAMILTONICITY is kind of the proverbial \mathcal{NP} -complete problem, or the beginning of it. Usually when people say \mathcal{NP} -completeness they immediately think of a problem which is called TRAVELING SALES PERSON’S problem (TSP), and HAMILTONICITY is just really a very reduced version of that problem.

TSP is the most studied \mathcal{NP} -hard problem and we are going to see it later on. HAMILTONICITY will allow us to move into that arena and deal with problems such as that.

Given a graph as input, the question related to HAMILTONICITY is whether the graph is Hamiltonian, or whether it has a Hamiltonian path or a Hamiltonian Cycle. As we have said before those two are equivalent problems. A Hamiltonian path is a “tour” along the edges such that all vertices are visited exactly once. We see that in the example graph on the foil there is a Hamiltonian path: We can go from v_1 to v_2 to v_3 to v_4 to v_5 – all the vertices are visited once – and then we can go back to v_1 again if we want a Hamiltonian cycle.

IN210 – lecture 7

VC \propto HAMILTONICITY

VC

edges



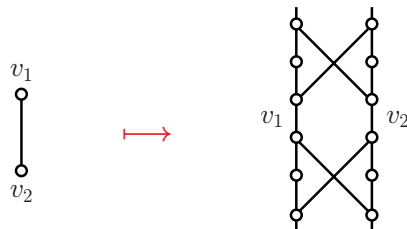
vertices

 K guards

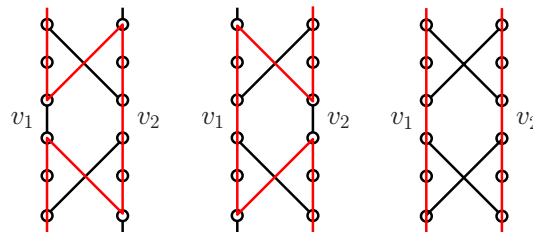
HAMILTONICITY

edge gadgets

how gadgets are connected

 K selector nodesedges \mapsto edge gadgets

A Hamiltonian path can visit the vertices in the edge gadget in one of three ways:



We want this to correspond to guards being placed on v_1 or v_2 or both v_1 and v_2 , respectively.

Autumn 1999

12 of 15

We have already shown that HAMILTONICITY is in class \mathcal{NP} , so what remains in the \mathcal{NP} -completeness proof is the reduction. By now you should be familiar with the strategy of proving \mathcal{NP} -completeness, namely translating the objects and properties such that the polynomial-time computable reduction maps 'Yes'-instances to 'Yes'-instances and 'No'-instances to 'No'-instances. A VERTEX COVER (VC) instance consists of edges, vertices and a number K , representing guards. We will translate edges into edge gadgets (components). The vertices will correspond to how we connect the gadgets. The number K will be translated into K so-called *selector nodes*.

I will show you how each object is translated. We start with the edges: Given an edge between nodes v_1 and v_2 , we make a edge gadget consisting of 12 nodes and 14 edges, as shown on the foil. The gadget is connected to the outer world by four edges, one from each corner node. The clever idea is that a Hamiltonian path can visit the edges in the edge gadget in exactly one of three ways:

One possibility is to enter the gadget at the upper-left node, visiting all the nodes, and then leave the gadget from the lower-left node. This will correspond to a guard being placed on vertex v_1 .

Possibility two is symmetrical, but instead of enter at the left side, we enter at the upper-right node and leave at the lower-right node. This will correspond to a guard being placed on vertex v_2 .

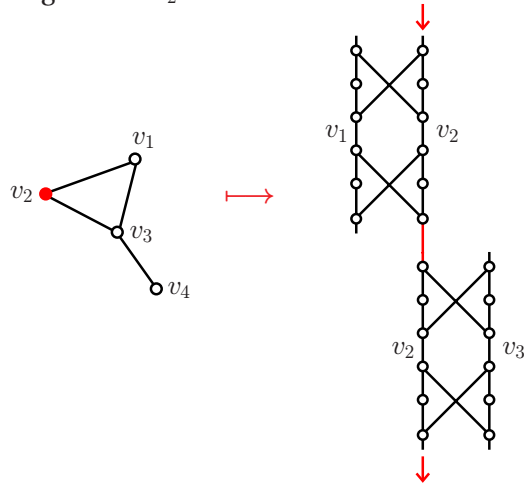
There is also a third possibility where we visit the gadget two times, the first time visiting the nodes on the left (or right) side and the next time visiting the other side. This will correspond to having guards both at v_1 and v_2 .

IN210 – lecture 7



vertices \mapsto how gadgets are connected

For each vertex v_2 , we connect together **in serial** all edge gadgets corresponding to edges from v_2 :



- Any Hamiltonian path entering at the v_2 -side (red arrow) can visit (if necessary) all vertices in the serially-connected gadgets and will eventually exit at bottom on the v_2 -side.
- This corresponds to the VC-property that a guard on v_2 covers all outgoing edges from v_2 .

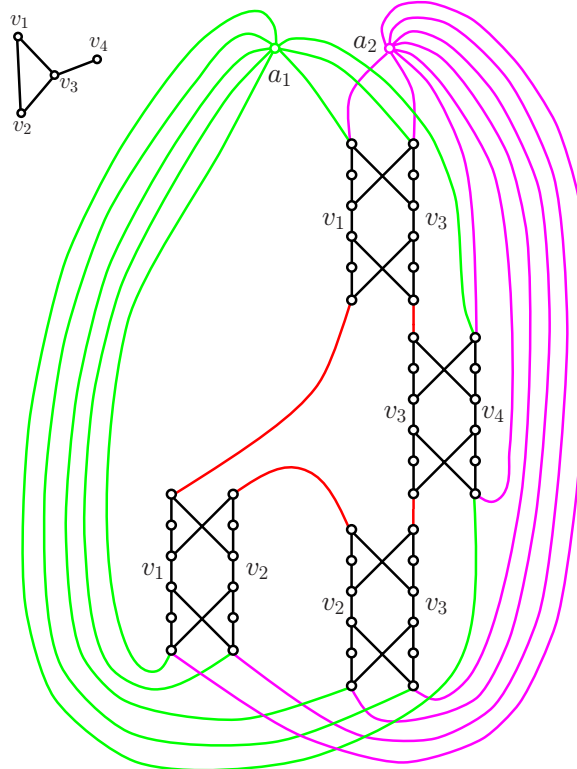
The gadgets have the comfortable property that if a Hamiltonian path enters on one side, then it exists also on the same side. This allows us for each vertex v_2 to connect in serial all gadgets corresponding to edges from v_2 . An example is shown on the foil. Any Hamiltonian path entering at the v_2 side of the first gadget in the linked list can visit (if necessary) all vertices in all gadgets in the list, and it will eventually exit at the bottom node on the v_2 side of the last gadget in the list. This will correspond to the VC-property that a guard placed on vertex v_2 covers all outgoing edges from v_2 .

IN210 – lecture 7



K guards \mapsto K selector nodes

We finish the construction by introducing K selector nodes a_i which are connected with all "loose" edges:



Autumn 1999

14 of 15

All that remains is to introduce K selector nodes a_i which are connected with all "loose" edges. Each selector node will correspond to a guard. A small example graph is shown on the foil.

Each selector node will say: "You can pass through me, of course, but only once. And in fact you have to pass through me once." So passing through a_1 would mean using one of these two available guards. Passing through a_2 would mean using the second guard.

These guards they are connected with all possible vertices in all possible ways (a_1 and a_2 is connected with both ends of all vertex gadget lists) so that they can go anywhere they like. They can go to any vertex gadget they like, but they cannot go to more than one. And that one vertex gadget will represent the intersection, or vertex, where the guard is standing.

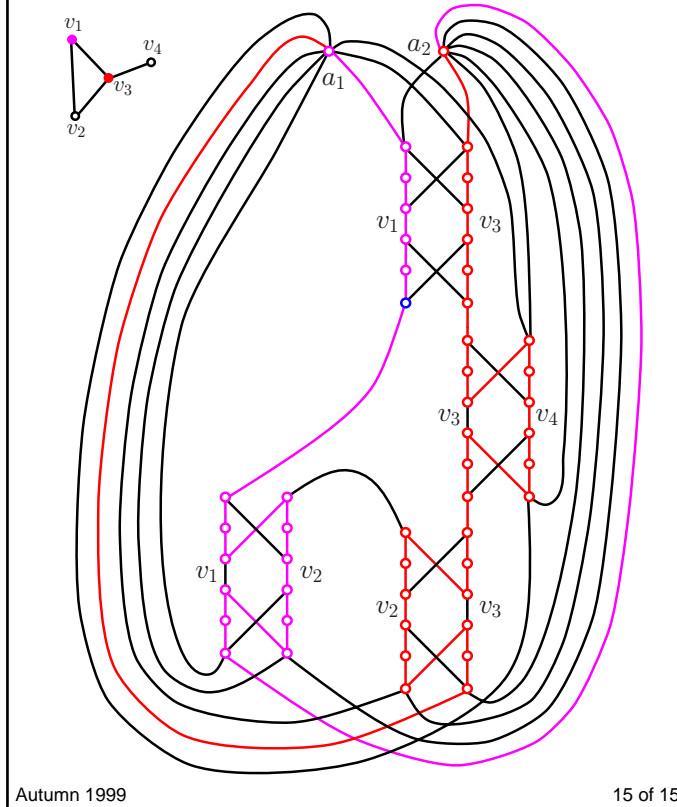
IN210 – lecture 7



There is a VC
which uses K guards

\Leftrightarrow

There is a
Hamiltonian cycle



The reduction that I have described will hopefully ensure that "there is a VC which uses K guards" in the VERTEX COVER instance if and only if "there is a Hamiltonian cycle" in the translated Hamiltonian Cycle instance. In a formal proof this has to be justified, but here we will just show how it works in our little example graph:

We can cover all nodes in the graph by placing guards on vertex v_1 and v_3 . So the VC-instance is a positive instance, and we would like to find a Hamiltonian cycle in the translated graph:

We start in selector node a_1 and visit (pink line) the a_1 gadget list. We only visit the left-hand side nodes in the gadget representing the edge between v_1 and v_3 because that edge is going to be covered at both endpoints, so a_2 will eventually visit the right-hand side nodes.

After visiting the (v_1, v_2) edge gadget we cannot go back to a_1 so we have to go to node a_2 . We have now visited all edges guarded by the policeman at v_1 . From a_2 we start visiting (red line) the gadget list corresponding to vertex v_3 as shown on the foil. In the end we return to a_1 – completing the Hamiltonian Cycle.

It is quite easy to see that if we cannot cover all edges of the VC-instance with K guards, then there will be at least one edge gadget we cannot reach without visiting the same node twice. That gadget will correspond to the uncovered edge.

As usual, in a formal proof we would have to show that the reduction can be computed in polynomial time in the length of the VC-instance.

So this is the end. I hope that these little hints will help you understand the \mathcal{NP} -completeness proofs in the G&J textbook. By all means read them, because what I presented here were not proofs, they were just high-level ideas that would help you understand the whole thing.

In the next lecture we will actually build a lot of things on top of this \mathcal{NP} -completeness theory. We will talk about other classes, other kinds of problems. We will get a map of complexity classes and hopefully understand what sort of complexity classes and what sort of problems are there out there. And then in the end – that is going to be like the remainder of the class, quite a few lectures – we are going to see that this whole point of view has its limitations.

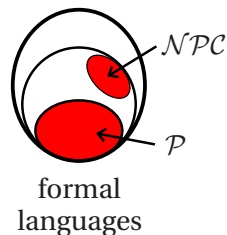
We will be able to in a certain sense solve all these difficult problems by using techniques which are suitable for these difficult problems. These techniques are available because this whole approach to complexity has certain limitations. We are going to see those. For the time being, we are orthodox, we are nice, and we are basing all of complexity theory on this what we have now, which is the worst-case best-solution paradigm. We will see everything in more detail later.

3.8 Lecture 8

IN210 – lecture 8



Strategy: Understand two classes in-depth.



Next:

- Extend the “map of classes”.
- Show how to use it for organizing our views on
 - problems
 - solutions
 - other issues

G&J:
5.0-5.1

Autumn 1999

1 of 14

We have seen last time how to reduce problems in \mathcal{NP} to one another. And by doing so we have gotten acquainted with those problems, learned to recognize their physiognomy so that when you meet a \mathcal{NP} -complete problem in the street you recognize him right away and say: "Oh, this must be one of those \mathcal{NP} -complete guys." That is good because a lot of practically important, intractable problems live in that class, so it is important to be able to recognize them.

Also by learning how to prove \mathcal{NP} -completeness – by learning all these techniques and things – we learn the basic tools of the trade, and then we are able to apply them in all walks of life. So we are both gaining insights about the \mathcal{NP} -complete and \mathcal{P} problems, and learning the basic techniques.

We have those two classes now: \mathcal{P} and \mathcal{NP} . And \mathcal{P} for us represents for the time being well-solved problems, \mathcal{NP} the difficult guys – problems that are difficult to deal with. What we are going to do next is: We will extend this map of classes by adding other classes to it, so that we have a proper map, not just two countries. And then we are going to show how to use that map for some basic work which we want to do. We want to organize our view on 1) problems – classifying problems according to complexity, 2) solutions – how to choose algorithms for problems that live in different classes, 3) other issues – and we will see that those other issues are also very interesting. We can really study all kinds of things based on the map of complexity classes. So let us first create this map, and then we show how to use it.

IN210 – lecture 8

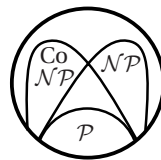


Other classes

Def. 1 (Co- \mathcal{NP}) A language L is in class Co- \mathcal{NP} if its complement L^c is in \mathcal{NP} .

Note: The complement of a formal language L is

- **formally** all the strings (over the given alphabet) that are not in the set L .
- **informally** the reverse property, i.e. all the instances that don't have the property corresponding to language L .



Example: NON-MATCHING and NON-HAMILTONICITY are in Co- \mathcal{NP} .

Our next class is Co- \mathcal{NP} . You can think of Co- \mathcal{NP} as being a kind of a twin brother or sister of \mathcal{NP} . The definition is that Co- \mathcal{NP} consists of languages that are complements of \mathcal{NP} -languages. A formal language is a set of strings, so the complement is the complement of that set, namely all the strings that are not in that set – all the strings being all the strings over the given alphabet.

That is the formal definition. The informal definition and the more meaningful one is that a language corresponds to a property, say HAMILTONICITY. And you can think of the complement of that language as being all the instances that don't have the property, instead they have the NON-HAMILTONICITY property. The difference between the two views is that in a set-theoretic complement, in addition to Non-Hamiltonian instances you would have all sorts of things, like strings that are not graphs at all. But those are not interesting. So you basically ignore them, and an algorithm can easily check whether something is a properly defined representation of a graph, or just kind of a nonsense string.

So eliminating the nonsense strings is easy for an algorithm. It follows that for all practical purposes the appropriate way to think of the complement of a language is in terms of the complement being the reverse property. As an example, we know that HAMILTONICITY and MATCHING is in \mathcal{NP} . It follows that NON-HAM and NON-MATCHING is in Co- \mathcal{NP} , where non-matching is all bipartite graphs that don't have a matching and NON-HAM is all graphs without a Hamiltonian path (cycle). Those are Co- \mathcal{NP} languages. We think of NON-HAM as all graphs which don't have a Hamiltonian path (cycle), but as explained above, formally NON-HAM also include all strings which don't encode graphs.

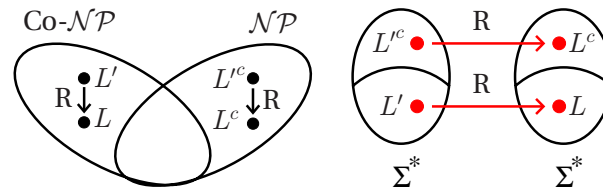
IN210 – lecture 8



Lemma 1 A language L is Co- \mathcal{NP} -complete if its complement L^c is \mathcal{NP} -complete.

Proof:

A reduction R from L'^c to L^c is also a reduction from L' to L .

**Example:**

- NON-HAMILTONICITY
- NON-SATISFIABILITY
- NON-CLIQUE
- NON- ...

are Co- \mathcal{NP} -complete problems. They don't seem to have "ID's" (efficient membership proofs).

Autumn 1999

3 of 14

We have \mathcal{NP} -complete languages and \mathcal{NP} -complete problems. What about Co- \mathcal{NP} -complete problems? There is a little lemma which says that a language L is Co- \mathcal{NP} -complete if its complement L^c is \mathcal{NP} -complete. So languages in Co- \mathcal{NPC} are simply complements of \mathcal{NP} -complete languages.

How do we prove this lemma? The lemma turns out to be trivial. The proof is just an observation that a reduction R that reduces a language L'^c to L^c , actually also reduces L' to L . The reason being that the reduction R maps 'Yes'-strings to 'Yes'-strings and 'No'-strings to 'No'-strings. The 'Yes'-strings in L'^c and L^c are 'No'-strings in L' and L , while 'No'-strings in L'^c and L^c are 'Yes'-strings in L' and L . That is the only difference, so R does the job in both cases.

This means that NON-HAMILTONICITY, NON-SATISFIABILITY, NON-CLIQUE, NON-3D-MATCHING – all kinds of non-problems – live in the class Co- \mathcal{NPC} . They are sort of negatively defined problems. But what is their physiognomy? What is interesting about them? The interesting property is that Co- \mathcal{NPC} problems don't seem to have ID's. They don't have a ticket. They don't have efficient proofs of membership.

This is an essential difference between NON-HAMILTONICITY and HAMILTONICITY. If I want to prove to you that I am a Hamiltonian graph, I show you a Hamiltonian path and you can easily check: "OK, go from here to here to here to here to here, and then you go back." So his is checked efficiently. But if I want to prove to you that I am non-Hamiltonian, what do I say? "I don't have a Hamiltonian cycle." But how can you verify this? It seems you basically have to test all possibilities. There doesn't seem to be an easy way to verify NON-

HAMILTONICITY.

So in that sense, although \mathcal{NP} -complete problems and $\text{Co-}\mathcal{NP}$ -complete problems both require exponential time to be solved exactly, typically these $\text{Co-}\mathcal{NP}$ -complete problems will be harder to deal with than \mathcal{NP} -complete problems. They will require somewhat different techniques, and some of the techniques that apply to \mathcal{NP} -complete problems they will not apply here. We will see more about that later when we will be studying alternative techniques and approaches to solving problems.

IN210 – lecture 8

**Def. 2 (Polynomial Space, PSPACE)**

$$PSPACE = \bigcup_k SPACE(n^k)$$

(**Note:** Space complexity is measured as number of tape squares used on (the **work tape** of) a TM.)

Lemma 2 $NTIME(f(n)) \subseteq SPACE(f(n))$

Proof: A TM moves its head one square at a time.

Corollary 1 $\mathcal{NP}, Co\text{-}\mathcal{NP} \subseteq PSPACE$

Examples of PSPACE-complete problems:

- QUANTIFIED BOOLEAN FORMULAS
 $(\exists x_1)(\forall x_2)(\exists x_3) \cdots (Qx_n)B$
- Generalized games ($n \times n$ boards)

Def. 3 (Exponential time, EXP)

$$EXP = \bigcup_k TIME(2^{n^k})$$

Lemma 3 $SPACE(f(n)) \subseteq TIME(k^{f(n)})$ for some constant k .

Proof (idea): The maximum number of distinct configuration with $f(n)$ tape squares used is $|Q| \cdot f(n) \cdot |\Gamma|^{f(n)}$.

Corollary 2 $PSPACE \subseteq EXP$

Autumn 1999

4 of 14

Our next class is going to be Polynomial Space (PSPACE). Without going into details space classes are defined completely analogously to time classes. Space means the number of squares used on the Turing machine tape. And usually to study the space usage properly, the Turing machine is redefined a little bit, so that it has a work tape – a special one – in addition to the input/output tape. The input tape is read only, the output tape is write only, while the work tape is used for its internal computations.

If you see things like Logarithmic Space, which is an important class that is studied in complexity theory, you should not be surprised, thinking: How can a machine possibly use logarithmic space when it has to see all those n squares on the input tape in order to read its input? The explanation is: A machine can use logarithmic space on its work tape, and that is really what is measured.

The space complexity class PSPACE is defined analogously to PTIME. It consists of all the languages which can be recognized on a Turing machine using a polynomial number of squares on the work tape.

Naturally we want to place this new class PSPACE on the map. We have a lemma which says that $NTIME(f(n))$ is contained in $SPACE(f(n))$. The proof is just a simple observation that a Turing machine cannot possibly use more space than time, because it just moves its read/write head one square at the time. So in n time steps the TM can have used at most n squares on the tape. The consequence of this little lemma is that \mathcal{NP} and $Co\text{-}\mathcal{NP}$ are contained in PSPACE.

Then we can ask what sort of problems are in PSPACE, and in particular: What are the PSPACE-complete problems, if they exist? Obviously all solvable

problems that we have seen so far – including MATCHING, HAMILTONICITY, shortest path in a graph, all the nice problems and less nice problems – they are in PSPACE. But they are not characteristic for PSPACE. They are all too easy! When we study a new class then we want to see the complete problems, because they are the ones that characterize the class – they are the typical hard problems in the class.

So what are the typical hard problems in PSPACE? What are the PSPACE-complete problems? If SAT is the original, proverbial kind of \mathcal{NP} -complete problem, and if NON-SAT is the original Co- \mathcal{NP} -complete problem, then QUANTIFIED BOOLEAN FORMULAS (QBF) would be the basic PSPACE-complete problem. QBF is this kind of thing: You have n boolean variables – x_1 to x_n – and a quantified boolean expression consists of a whole bunch of quantified boolean variables followed by a boolean expression without quantification.

Do you know this language of quantifiers? You will see this in all sorts of classes later on. You will not be able to live in this place very long without learning some logic. But to say it very briefly:

Boolean variables are like all other variables in this world, they are some kind of unknowns. They take values from the set $\{0, 1\}$ or $\{\text{'TRUE'}, \text{'FALSE'}\}$. So if you usually think of variables as being some kind of numbers, then boolean variables are the same thing except they can be only zero or one. And then there are quantifiers, two of them: This existential quantifier says “there exists a x_1 ”, and this universal quantifier says “for all x_2 ”.

And these two quantifiers they are what mathematicians used quite a bit for stating their theorems. This is how they got into logic, because logic is just a formalization of mathematical reasoning. The reasoning that goes in here is: There exists a value for x_1 , such that any way you choose the value for x_2 , there still exists a value x_3 such that, and so on. And the boolean formula B is a kind of a condition. It is a requirement. It says: This is how those values have to be.

And if this is too abstract, then a more concrete and more real-world version of the same thing is generalized games. What are games? Think of anything like checkers or chess or go. In those games you use a board to play on, but any game like chess which is played on 8 by 8 squares or go which is played usually on 19 by 19, they are really finite.

We would like to answer questions like “How difficult is it to play chess by computer?” But in order to apply the concepts and techniques we have learned so that we can study the complexity of the games, you have to generalize games in the sense that you don’t have an 8 by 8 fixed, finite board, but you have an infinite board. You have an n by n board. Everything else is basically the same.

That is how you generalize the games. You have to do it in order to apply this kind of reasoning to them, because anything that is finite in complexity theory is just constant time – it’s easy. But as soon as you generalize, then you can ask the basic question. And what is the basic question in a game? You look at the game, there is something on the board, and you ask: “Does white have a winning strategy?”

What does that mean? It means that white would like to do a move in such a way that no matter what the black guy does – for every move that the black guy does – there exists a good move for the white such that no matter what the black guy does subsequently, there is still a good move for the white, and so on. And then B will say what a good move is: It is that all the rules are obeyed and the end result is a victory.

So you can think of this whole boolean formula in terms of a game, as encoding what we mean by a winning strategy. A lot of generalized game-like problems will be PSPACE-complete.

We will see this a little bit later also, but this existential quantification and

universal quantification, they should remind us of two things – namely of the classes \mathcal{NP} and $\text{Co-}\mathcal{NP}$. \mathcal{NP} -properties are typically: "There *exists* something" – there exists a certificate, there exists an ID, there exists a Hamiltonian cycle in the graph, there exists a satisfiable truth assignment to the variables, and so on. $\text{Co-}\mathcal{NP}$ properties say: "There exists no" – there is no Hamiltonian cycle, there is no satisfiable truth assignment. We can say this in a more positive way: For *all* truth assignments T , T does not satisfy the formula. Or, for all permutations of the vertices in the graph, the permutation is not a Hamiltonian cycle. So the universal quantifier is in a way linked to $\text{Co-}\mathcal{NP}$.

This alternating existential and universal quantifiers actually gives rise to an interesting part of complexity theory – there are all sorts of ideas related to that – but ultimately if you take the alternation n steps, then you get something which is PSPACE-complete (if the game is not too easy, for example if white can win by force in x moves from the opening position then there are no reason to search n moves ahead).

Our next class is going to be Exponential time (EXP). I am giving you really just a few basic classes. This map is very, very big. There are, I believe, of the order of 100 classes that people have studied, but not all classes are equally interesting or equally important.

Exponential time will be the hardest solvable class we will be looking at. It is defined as the union over k of classes $\text{TIME}(2^{n^k})$. Again the definition is analogous to Polynomial time and Polynomial space, except that here we have the exponential function instead of the polynomial.

To place this EXP class on the map and to place other classes in relation to it, we have a little lemma which says that the $\text{SPACE}(f(n))$ is contained in $\text{TIME}(k^{f(n)})$ for some constant k . And the proof is again very easy: It is just an observation that the maximum number of distinct configurations that one can create by using $f(n)$ squares on the tape, is $|Q|$ (the number of states) times $f(n)$ (the number of squares used) times $|\Gamma|$ (the size of the tape character set) to the power $f(n)$.

Recall that a configuration of a Turing machine is defined as a state, the position of the read/write head and the contents of the tape. There are $|Q|$ possible states, $f(n)$ possible positions of the read/write head on the tape if we are using $f(n)$ squares, and each square can have one of the $|\Gamma|$ characters. $f(n)$ squares together can contain $|\Gamma|^{f(n)}$ possible strings.

What is the meaning of this statement that the number of distinct configurations is limited by this expression? Well, in this much time a machine that uses no more than $f(n)$ squares, must repeat a configuration. Now, the fact that a Turing machine repeats a configuration means that it is in exactly the same situation in which it was before – the same content of the tape, the same state and the same position of the read/write head. This further means that the machine is in an infinite loop, if it is a deterministic machine. A deterministic machine that repeats its configuration will repeat that configuration after the same number of steps again – because it is deterministic. So it will never halt.

If the machine is non-deterministic, then you can throw away all computation that has happened between two repetitions of the configuration, and come up with a shorter computation that does the same. So $\text{SPACE}(f(n)) \in \text{TIME}(K^{f(n)})$ means that if a problem can be solved by a deterministic TM (or a NTM because $\text{SPACE}=\text{NSPACE}$) using a maximum of $\mathcal{O}(f(n))$ space, then there exist a TM (but not necessarily the same machine in the case of a NTM, because we might have to throw away part of the computation to get one that is short enough) that solves the same problem using no more than $\mathcal{O}(2^{n^k})$ step of time.

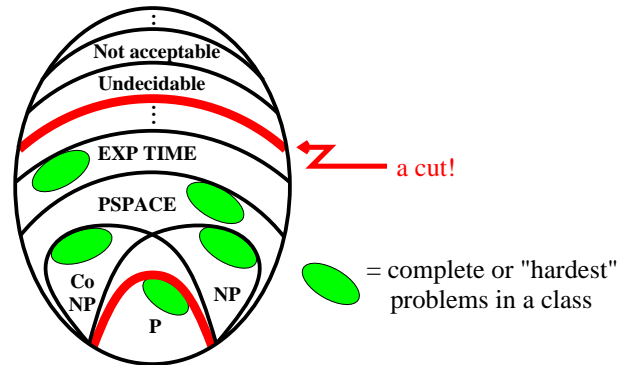
The consequence of this is that PSPACE is contained in EXP.

(This is a blank page)

IN210 – lecture 8



Map of classes



Are inclusions proper (\subsetneq)?

- $\mathcal{P} \subsetneq \mathcal{NP}$?
- $\mathcal{P} \subsetneq \text{PSPACE}$?

Autumn 1999

5 of 14

With these few new classes and the old ones we can construct a map of classes, and we can spend some time looking at this map, just understanding how the things fit together. And then after that we can use the map as we use maps: to place different objects there, to study different countries. When you meet a new guy, a new problem, then the first thing you ask is: "Where are you coming from, you have this strange accent."

So the first thing you want to do when you meet a problem in your life is to place it on this map of classes. And as soon as you do that, you know something about the problem. You know its character and you know how to deal with him. You know what sort of algorithmic approaches are appropriate for one kind of problems, and what are appropriate for other kinds of problems. You know what sort of difficulties to expect from the problem. In other words, you know its physiognomy.

Here on this foil is our basic map. We have class \mathcal{P} , and actually \mathcal{P} has also complete problems. I must say that all these issues are studied properly in class IN394 which is taught usually in spring, and there we prove actually all kinds of things and do this work more rigorously. Here I am trying just to give you enough insight and then enough theoretical kind of background to feel comfortable on that side, but mainly we deal with the basic insights. And then if you become interested in actually doing this work properly, then you take the IN394 class.

So problems that are complete for \mathcal{P} is one of the things that we won't be studying in this class, but they exist. And then we have \mathcal{NP} and $\text{Co-}\mathcal{NP}$ with

their complete problems. And \mathcal{P} is contained in both \mathcal{NP} and $\text{Co-}\mathcal{NP}$. We also have an interesting little area in the intersection between \mathcal{NP} and $\text{Co-}\mathcal{NP}$ which is not known to be in \mathcal{P} , and which has a kind of a story of its own. We will look at that area later.

And then we have PSPACE as a superset, with PSPACE-complete problems. We have Exponential time with EXP-complete problems, and then all sorts of classes of solvable problems unbelievably difficult on top of this.

The big red borderline between \mathcal{P} and the rest is the properly solved problems versus the not so well solved problems. We also have another practically important borderline which separates the problems that can in principle be solved by algorithms from those that cannot. So this big red upper borderline partitions the map of all problems into two halves – the solvable ones and the unsolvable ones. Among the unsolvable problems we have problems that are acceptable but not decidable, and then problems that are not even acceptable, and so on. So there is also a kind of hierarchy in the upper half of the map.

So this is our map and then there are some important facts about the map. We know that \mathcal{P} is included in \mathcal{NP} and that \mathcal{P} is included in $\text{Co-}\mathcal{NP}$. We also know that \mathcal{NP} and $\text{Co-}\mathcal{NP}$ are included in PSPACE and that PSPACE is included in EXP by virtue of the two small lemmas we have seen. The interesting question is: Are those inclusions proper?

Since nobody has proven yet that we cannot solve an \mathcal{NP} -complete problem in polynomial time – even though we suspect that it is so – then \mathcal{P} may in fact be equal to \mathcal{NP} , not just included in \mathcal{NP} . " $\mathcal{P}=\mathcal{NP}$?" is one of the big, big open questions in computer science.

In fact \mathcal{P} may even be equal to PSPACE. We don't know for sure. We don't know in the sense that we cannot prove that there are problems that are in PSPACE but not in \mathcal{P} . That is what the question whether the inclusion is proper, means: Is there really a problem in the bigger class which provably isn't a member of the smaller class?

IN210 – lecture 8



Do complexity classes really exist?

Theorem 1 (Time Hierarchy Theorem) *If $f(n) \geq n$ is a proper complexity function then*
 $TIME(f(n)) \subsetneq TIME((f(2n+1))^3)$.

Basic message: Given more time we can (provably) solve more problems.

Corollary 3 $\mathcal{P} \subsetneq EXP$

In other words: $L \in EXP\text{-complete} \Rightarrow L \notin \mathcal{P}$.

Autumn 1999

6 of 14

Because we haven't been able to answer those questions, then it is natural to ask whether these complexity classes really exist at all. In other words: Could it be that all of these things that we are calling the classes, that they are really just one class? Could it be that these classes just reflect our lack of knowledge?

It turns out that we are able to prove that the complexity classes really exist. This fact is expressed in theory as the Time and Space Hierarchy Theorems. I am here just quoting the Time Hierarchy Theorem. The proof is easy – it is actually an easy diagonalization – but it is not so interesting, and we do it in the IN394 class. Here I am only showing you the theorem and interpreting what it means, so that you know how this theory is really constructed – that is important.

The theorem says something technical which is not very interesting but I will say what it is: If $f(n)$ happens to be bigger than n and a proper complexity function . . . Proper complexity function means that you can count in time $f(n)$. You can imagine $f(n)$ as being a kind of a number that is defined in such a way that no machine can compute that number. This little technical half-sentence in the beginning of the theorem just excludes those odd cases.

So $f(n)$ is basically just a reasonable kind of function – as we think of functions. Its values are bigger than n , and it is not something that cannot even be computed. The theorem says that as long as $f(n)$ is a reasonable thing, then $TIME(f(n))$ is contained properly in $TIME((f(2n+1))^3)$.

What this is saying is that given more time, you can solve more problems. That is the basic message. And then given even more time, you can solve even more problems. So the message of the Time Hierarchy Theorem is that there is

a whole hierarchy of complexity classes, in fact an infinite one. They are real.

The corollary, also proven in the IN394 class and also rather easy, is that \mathcal{P} is properly contained in Exponential time. This is what we do know for sure. It means that if we have an EXP-complete problem, then we know for sure that the problem is not solvable in polynomial time. So we have actually problems that are provably intractable. They are the EXP-complete problems.

Unfortunately, or maybe fortunately, those problems are not so practically interesting as the \mathcal{NP} -complete problems. So this is why we prefer to study \mathcal{NP} -completeness, although \mathcal{NP} -completeness is from a theoretical point of view not such an elegant thing as EXP-completeness, where we know for sure that those problems that are EXP-complete, are not polynomial-time solvable.

IN210 – lecture 8



Are complexity classes practically relevant?

Assumptions, abstractions, modeling simplifications are possible weak points of theory as a model:

- Problem \rightsquigarrow formal language
 - search/optimization \rightsquigarrow decision
- Solution, algorithm \rightsquigarrow Turing machine

Alternative approaches to computation:

 - Analog computation
 - Biological computers
 - Neural networks
 - Quantum computers
 - ...
- Real time \rightsquigarrow # of TM steps (discrete, finite)
- Real intractability \rightsquigarrow worst case / best solution

Alternative approaches to algorithm design & analysis:

 - Approximation
 - Average-case analysis
 - Randomized algorithms

Autumn 1999 · · ·

7 of 14

Another interesting, related question is: Are these classes relevant in practice? Is this complexity theory *really* practically relevant, or is it just something that theoreticians like to deal with because they like to deal with abstract things, but these abstract things have nothing to do with practical reality?

That is a serious question. And a more kind of practical view of this question is: We would like to make a program that plays chess, but a theoretician comes along and says: "Look, this chess playing, it is PSPACE-complete. Nobody has come up with an efficient algorithm for this, so you can just forget it."

But then I am saying: "Wait a minute, your whole theory is based on Turing machines, formal languages and things, but this is not what we really have in reality. Suppose I come up with a new kind of computer. maybe a new kind of computer can play chess very fast." And I might continue: "And your complexity is based on completely solving a game, but I am just interested in finding a reasonable good move. And After all, in 1997 the highly parallelized machine Deep Blue beat the human chess world champion Garri Kasparov 2.5 – 3.5 in a match, so chess cannot be that difficult."

We might feel we understand the complexity theory now, but it is equally important to try to understand the basic assumptions on which this theory is based. Those assumptions were the result of our basic operations 'abstraction' and 'formalization'. So we want to understand how we actually formalized all the practical things, and then be critical towards those formalizations so that we can find new venues of approach, new possibilities.

We have formalized problems as formal languages. This means that we are

assuming that the practical complexity of a real-world search or optimization problem is measured in a reasonable way by a decision version of the problem – because deciding (solving) a formal language means answering 'Yes' or 'No'. Is this a reasonable assumption?

We have formalized solutions or algorithms as Turing machines. A TM is a kind of an odd thing. It is very simple and basic, and it's discrete – it's finite. So we can think of other possibilities like analog computers, computers that are not digital at all, computers that are based on the law of quantum physics instead of classical physics. People think about biological computers, neural networks, quantum computers, etc. maybe they can solve \mathcal{NP} -complete problems fast?

So the fact that a problem is proven difficult by our theory, does not mean that it is in an absolute sense difficult. But it does mean by virtue of the basic thesis that we have seen – the computational complexity thesis – that with the existing digital computers, difficulty with a Turing machine means difficulty. Intractability with this theory means practical intractability.

This is related to the next abstraction that we did, namely modeling real time as the number of TM steps. That is also a very big kind of assumption, and as we will see later, it turns out that it doesn't apply to quantum computers based on quantum physics. But these assumptions by and large work because there is such a huge difference between exponential time and polynomial time.

It turns out that exponential-time algorithms are completely different kinds of algorithms than polynomial-time algorithms. So what this distinction between polynomial time versus the rest captures, is really whether your problem is solved by one kind of algorithms or whether you need to use another kind of algorithms.

The fourth and for us most interesting abstraction is that we are modeling what we intuitively mean by a real-world problem being difficult or intractable, by using a very limited kind of approach: We are studying the worst possible instances and we require the best possible solution.

We will come back to this issue later and talk about it in depth, but let me just say now that there is something very odd about this assumption. One might argue: "Look, there are all kinds of possible instances. If you have n squares on the tape, there are basically exponentially (in n) many possible instances. Some of those possible instances can be really difficult to deal with. maybe most of them are not. But you focus on those worst guys."

And further: "And then you are looking for the best solution. For example in the TSP you have lots of possible tours, lots of possible solution. But you want the absolutely best one. Out of the exponentially many, you want the very best. Why do you want the very best? Maybe a reasonable good solution is good enough."

So this last assumption will give us a venue of dealing with \mathcal{NP} -complete or difficult problems by using alternative approaches to algorithm design and analysis. We can design approximation algorithms which don't necessarily produce the very best solution. We can do average-case analysis and use algorithms which are good on average and may work very well in practice, but which in the worst case are not so good. We can use randomized algorithms – algorithms that can toss coins – and possibly all kinds of other things.

IN210 – lecture 8



Placing problems on the map

- Problem \leftrightarrow complexity
- Complexity = performance of best possible algorithm
- Lower bound \leq complexity \leq upper bound
 - Upper bound \rightarrow algorithm
 - Lower bound \rightarrow reduction, etc.

Autumn 1999

8 of 14

The next basic issue now that we have the map is: How do we place problems on the map? Given a new problem, how do we find the appropriate place for the problem on the map? We have seen the basic strategies for proving \mathcal{NP} -completeness and things like that. More generally, what the placement on the map should reflect, is the complexity of the problem – how difficult the problem is to solve.

Generally an estimate of the complexity will consist of two things: One thing is the upper bound on the complexity and another thing is the lower bound – upper bound and lower bound being the two bounds between which we try to squeeze the actual complexity.

An upper bound on the actual complexity is usually an algorithm. So we show an algorithm and we say: "Look, we can solve this problem in time $\mathcal{O}(n^3)$. Here is an algorithm. Here is a proof." A lower bound means that even the best possible algorithm cannot solve the problem faster. So to prove a lower bound means to show that even the best possible algorithm can not perform better than this.

Coming up with upper bounds is the issue of algorithm design, while proving lower bounds is more what we mean by complexity studies. Lower bounds is really the more interesting side, from a theoretical viewpoint. Showing an efficient algorithm is sometimes difficult, but in principle it is a straightforward procedure. But how do you show that even the best possible algorithm can not do any better than your given bound? To prove that we use reductions and all kinds of other methods – we will see some of them in the IN394 class.

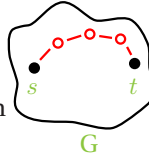
So this is how we place the problems on the map – by basically showing lower bounds and upper bounds. In some way you have seen that already in \mathcal{NP} -completeness proofs because an upper bound is the little NTM that proves that a problem is in \mathcal{NP} by guessing a solution (ticket) and verifying it in polynomial time. A lower bound is the \mathcal{NP} -hardness proof: You are reducing a difficult problem to your problem and thereby showing that your problem is at least as difficult as the other difficult problem. So in principle these two steps are there.

IN210 – lecture 8

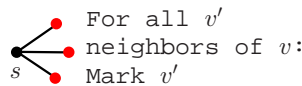


Examples (insights)

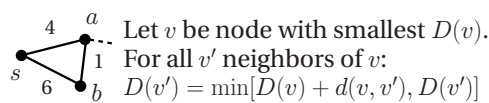
Finding a path in a graph



- **any** path: “greedy” algorithm (linear time)



- **shortest** path: iterative improvement, Dijkstra (polynomial time)



- **longest** path: NP-complete (exhaustive search, backtracking, exponential time)

HAMILTONICITY \propto Longest path

Autumn 1999

9 of 14

Then of course you want to have insight. You want to know when you see a problem, how difficult the problem is. This insight you gain basically by studying lots of problems. That is a part of the work that we have to do here, and in particular this is what you will do in your group exercises quite a bit – seeing different problems, studying them and understanding them. We are just going to see a few problems now, just to gain some kind of basic ideas of how difficult different variants of a problem typically are.

Here we see a basic problem of finding a path in the graph, and some variants of that basic problem. So given a graph we pick two vertices s and t , and then we ask different questions about the possible ways of getting from s to t . Possibly the edges in this graph may have numbers associated with them, called distances. So the graph becomes a kind of a map of a country or a city.

First you ask: Is there a way to get from place s to place t at all – from your source to your destination (target)? And the question whether there is any path in the graph from s to t , is also related to the question of the connectedness, or connectivity, of the graph: Is the graph connected or not? Is there a way, for every pair of vertices, to come from the first element to the second element in the pair, using the edges of the graph?

Such questions, they are easy. They are answered by algorithms which are in essence “greedy”. In this class greedy means the kind of algorithm that does not ever correct something that it has already done – there is no backtracking. A greedy algorithm chooses always the local optimum as its next choice.

A greedy person just assembles things and never gives them away. A greedy algorithm will just do its work and never undo or improve what it has done. That is the intuition. So, for example, how do you solve this ‘any path’ problem? Well, you start from your node s , and then you mark all its neighbors. You mark

the first node, the second node and the third node. And then when you see a marked guy you know that he is definitely reachable from your s .

Then you visit all the marked guys and mark their neighbors. If eventually you mark all the nodes, then you say: "This graph is connected, I can go anywhere I like." Or if you are interested in reaching node t : If you eventually mark node t , then you know that there is a path from s to t , otherwise there is not.

The point is that you never have to unmark a marked node. A marked node simply means that it is reachable, and if something is reachable, then it is reachable. Nothing to correct. And this kind of approach usually leads to a linear-time algorithm, $\mathcal{O}(n)$ – to something that is very efficient.

Now we look at another variant of the problem: We are interested not just in coming from s to t , but actually in the shortest possible way from s to t . How difficult is this? It turns out that we cannot use this kind of greedy approach here, because sometimes the partial solutions need to be improved on. Look at this little example here on the foil: Coming from s to a costs 4 units, say it's 4 kilometers. From s to b is 6 kilometers. So in the first iteration you will mark a as being on distance 4 from s and b as being on distance 6 from s . But there is actually a shorter way to come to b , which is through a . Because from a to b is only 1 kilometer, which gives a total of 5 kilometers. We need to be able to correct or improve the tentative solution.

If $D(v)$ is the overall distance from node s to node v , and small d 's are the individual little road distances, then iterations of the following procedure can be shown to produce the correct solution: "Let v be the unmarked node with smallest $D(v)$. Mark v , and for all neighbors v' of v do the following: Set $D(v')$ equal to the minimum of the old $D(v')$ and $D(v) + d(v, v')$." So the algorithm first selects the unmarked node v with the smallest distance from s . Then it marks v , and for all neighbors v' checks if we can get a new minimum distance from s to v' by going through v . If that is the case, then we update $D(v')$ because we have found a shorter way from s to v' . This procedure continues until all nodes have been marked (visited).

This iterative-improvement algorithm is known as Dijkstra's algorithm and you have seen it already in IN115. Dijkstra's shortest path algorithm is not linear, but it is polynomial – typically $n^2 \log n$ for dense graphs (graphs with "many" edges). Dijkstra's algorithm is efficient because although it uses iterations, it never does any backtracking in the graph – once a node is marked, it is never unmarked again. So intuitively the algorithm is polynomial because every node and every edge is processed only once, and the number of edges and nodes is of course polynomial in the length of the input – the nodes and the edges *are* the input.

Then you might wonder about the longest simple path from s to t (simple means visiting no vertex more than once) How difficult is it to decide whether there is a simple path of length K or more? It turns out to be \mathcal{NP} -complete because HAMILTONICITY is ultimately that. Given an unweighted graph and two nodes t and s with an edge between them, then you can ask: "What is the longest simple path from s to t ?" If the longest path happens to be all other vertices, $n - 1$, then you know immediately that the graph is Hamiltonian. Otherwise you can do the same test for all other edges, and then answer HAMILTONICITY. If all answers are "No, there are no path of length $n-1$ or more", then you know that the graph is not Hamiltonian. So HAMILTONICITY is reducible to longest path by this simple reduction.

So these three problems seem very similar, but they are really very different. 'Any path' is solvable by this simple greedy algorithm. 'Shortest path' requires a little bit more sophisticated algorithm, but it is still polynomial time. 'Longest path' is \mathcal{NP} -complete and most likely needs exponential time.

IN210 – lecture 8



Matching

1DM	2DM	3DM
greedy	iterative	exhaustive
(lin. time)	improvement	search
$\mathcal{O}(n)$	$\mathcal{O}(n^3)$ alg.	$\mathcal{O}(2^{2n})$ alg.

Satisfiability

1SAT	2SAT	3SAT
$(x_1) \wedge \dots$	$(x_1 \vee x_2) \wedge \dots$	$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge \dots$

Generalization: Matroid intersection

1 matroid	2 matroids	3 matroids
-----------	------------	------------

Autumn 1999

10 of 14

Now we are looking at a few more examples just to gain some basic insights about the nature of these problems – how different seemingly similar problems can be in complexity. Our next example is 1DM, 2DM and 3DM. What is 1-DIMENSIONAL MATCHING? Well, it is like marriage in some middle-eastern countries where one man is compatible, possibly, with several women – he can marry all of them!

So in that kind of matching a greedy approach works well. Look at the example on the foil. 'a' and 'b' are girls while '1' and '2' are boys. Suppose you marry 'a' with 1. Then you don't have to correct it, because 'b' independently can also be married with 1. Every girl on the left side has to be married because an unmarried girl is a shame for the whole family. But the men they can marry multiple girls. So if you have a daughter, then you just marry her with anyone of the boys she likes – it doesn't matter to whom. Because your next daughter can marry the same guy, that is alright. So you can use the greedy approach. You never have to correct or improve what you have done.

But in 2DM you can make mistakes. Say you marry 'a' and 1. There is a kind of good matching that marries everybody, but if you marry 'a' and 1, then you spoil everything. So what do you do? A good 2DM-algorithm will be searching for ways to improve on the existing solution. Without going into much detail: Such a way would consist in a path that alternates between the edges – the combinations that are taken in the current marriage, and those that are not. So for example if 'a' and 1 are married, then an improvement would be found if we find a path that has a non-married couple and then a married couple and then

a non-married couple, and then possibly a married couple and a non-married couple, and so on.

So in that path we would have, say, K non-married couples and $K - 1$ married couples. And what we would do then is to swap: We would swap the married guys and the non-married guys. So that this red edge would no longer be red, but this edge and this edge that previously were not red, are become red. So we improve the solution. So that simple idea actually gives rise to various matching algorithms. And those algorithms they are not linear time, but they are polynomial – something like time of order $\mathcal{O}(n^3)$ or something like that.

3DM as we have seen, seems to require exhaustive search. The complexity of the algorithm would typically be of the order of $\mathcal{O}(2^n)$ or $\mathcal{O}(3^n)$ because there are of the order $n!$ different matchings to be tried.

Interestingly the situation with SAT is similar. If you have 1SAT, which you typically don't have because it is a very trivial problem, then just about any naive approach works. 2SAT requires quite a bit of thought. It turns out to be solvable in polynomial time, but not with a greedy algorithm. And again you can think about how you would solve 2SAT with a polynomial-time algorithm. So there are polynomial-time algorithms, but they are based on this kind of approach: You do something and then you correct and improve until you find the real solution. 3SAT as we have seen, is \mathcal{NP} -complete.

Is there some kind of general rule? There seems to be a big difference between 1-, 2- and 3-situations. And it turns out that there is a pattern. This insight can be generalized as the so-called *matroid intersection problem*. Matroid intuitively here being a condition, while intersection being kind of satisfaction. In 1DM you are satisfying one condition because only girls need to be married – we don't care about the boys. So in 1DM you have one matroid, and the problem is easy.

If two conditions need to be satisfied simultaneously then you have a two matroids intersection problem. To solve it in polynomial time you need some kind of matching approach.

If you have three matroids, the problem is already horrible. Satisfying three people at the same time, or three kinds of conditions, leads to intractability, intuitively speaking.

There will be more about matroids next time (lecture 9) when we talk about subset systems.

IN210 – lecture 8



Placing real-world problems on the map

- **Decision** problems ... is there a ...
- **Search** problems ... find a ...
- **Optimization** problems ... find the largest/smallest ...

Def. 4 Problem L is \mathcal{NP} -easy if there exists a problem $L' \in \mathcal{NP}$ such that $L \propto L'$.

Note: We generalize the notion of polynomial reduction (\propto) in order to deal with search/optimization problems:

- We allow a polynomial number of “calls” to $M_{L'}$ instead of just 1.
- We allow the output to be a general string instead of just YES or NO.

(G&J calls this **Turing reduction**, \propto_T)



Note: If $L \in \mathcal{NP}$ -easy and $\mathcal{P}=\mathcal{NP}$ then $L \in \mathcal{P}$, i.e. L is not any more difficult/complex than \mathcal{NP} -complete problems.

So studying lots of problems – understanding what sort of problems live in different areas on map – is like complexity theory geography. It is like adventure of travel. You travel to different places and really understand what the situation there is, what sort of people live there, and so on. That turns out very useful for understanding not only the complexity theory but also life, in the world of computation, in general.

We move on to another related issue, which is how to place real-world problems on the map. So far we have only been talking about the decision problems, formalized by formal languages. The real-worldliness of the decision problems can easily be challenged. Are we really interested in just answering 'Yes' or 'No' in real life? Usually not. Usually you are interested in finding some kind of solution, computing a function – all sorts of things.

So the question is now: How well can this theory that is based on Yes's and No's, help us to deal with real-world problems where we have to produce all sorts of answers?

We have seen decision problems where the question is: *Is there a Hamiltonian path, and things like that.* Now we move on to search problems where we have to *find* a Hamiltonian path. And then eventually we will have optimization problems where we not only want to find a TSP-tour of length K or less, but actually find the *shortest* TSP-tour in the graph. Sometimes we want to find the smallest something, other times we are asking for the largest solution. Those are optimization problems where we want to find the solution which optimizes a certain criteria.

First we look at search problems. Are search problems more difficult than the corresponding decision problems? That is now the question. If you have a search problem, then the search problem is easily modeled by a decision problem. The question is: Does this modeling preserve complexity, or do we also by simplifying the problem make the problem easier?

The basic message is that in most cases, by turning a search problem into a decision problem, the problem is not made any easier. And this I am going to show you now by formalizing the whole issue. We formalize the issue by defining the expression \mathcal{NP} -easy. This of course relates to \mathcal{NP} -completeness, but it uses a more generalized version of the polynomial reduction.

We generalize the notion of polynomial reduction in order to deal with search and optimization problem, where the solution is not only 'Yes' or 'No'. We allow a reduction $R: L \propto L'$ to do polynomially many "runs" on the M'_L machine, not only one. M'_L is the Turing machine which decides L' . We also allow the output of the reduction to be a general string instead of just 'Yes' or 'No'.

The justification for allowing a more generalized notion of a polynomial reduction is that a polynomial number of calls to a polynomial subroutine is still a polynomial algorithm. So if L' can be solved in polynomial time, so can L also.

The G&J textbook calls this generalized reduction for a *Turing reduction* and write \propto_T , but we will most of the time stick to our usual notation. Which reduction we use will hopefully be clear from the context.

We say that a problem is \mathcal{NP} -easy if there exists another problem in \mathcal{NP} such that our problem is polynomial-time (Turing) reducible to that problem. So if \mathcal{NP} turns out to be solvable in polynomial time, then our problem is also solvable in polynomial time. ' \mathcal{NP} -easy' intuitively means 'not any more difficult than \mathcal{NP} -complete problems', because an \mathcal{NP} -easy problem can be reduced to a problem in \mathcal{NP} which in turn can be reduced to any \mathcal{NP} -complete problem.

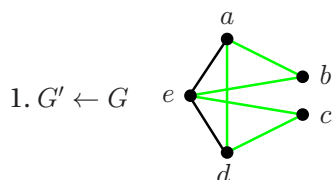
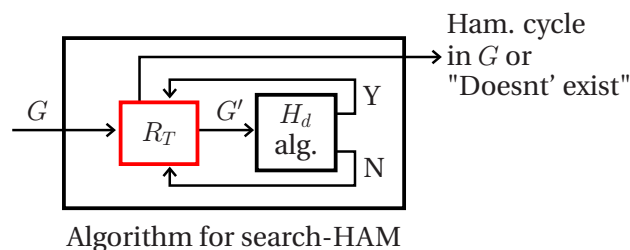
IN210 – lecture 8



Example (of basic technique)

Theorem 2 Search version of HAMILTONICITY is \mathcal{NP} -easy.

Proof: (Turing) reduction to decision version of HAMILTONICITY:



2. Run H_d on G' . **If** $H_D(G') = \text{'NO'}$ **then** output "Ham. path doesn't exist" **else** remove an edge from G' :

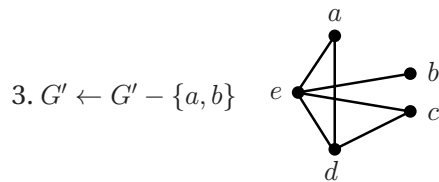
I am now going to prove to you that the search version of HAMILTONICITY is \mathcal{NP} -easy, by (Turing) reducing it to the decision version of HAMILTONICITY. But I will in fact accomplish more than that. What I will be showing you is a kind of a procedure, a pattern, which can be used for showing that the search version of just about any problem in \mathcal{NP} , is \mathcal{NP} -easy. Things like search-SAT, search-3DM and so on.

I will show you how to solve search-HAMILTONICITY by using the reduction on the foil. What is the reduction? Suppose that we have an algorithm for the decision version of HAMILTONICITY. The decision algorithm is going to say 'Yes' or 'No', depending on whether its input G' is Hamiltonian. If it says 'Yes', then this line back to R_T is activated, if it says 'No', then this other line back to R_T is activated. After at most a polynomial number of calls on the decision algorithm, R_T will output a Hamiltonian cycle if there exists one, or answer "No, there is no Hamiltonian cycle in G ."

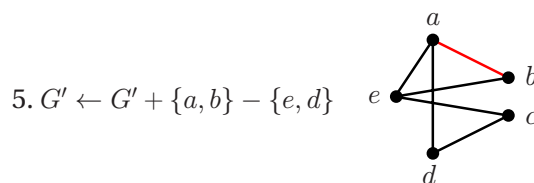
So in that way R_T will reduce the search version of HAMILTONICITY to the decision version of HAMILTONICITY. I will use the little graph G on the foil to illustrate how exactly R_T works.

First R_T will send G as input to H_D – the decision algorithm for solving HAMILTONICITY. If H_D says 'No', then we know that the graph is not Hamiltonian and we are finished. If H_D says 'Yes', then we know that definitely there is a Hamiltonian cycle in the graph. It is a matter of finding it.

IN210 – lecture 8



4. Run H_d on G' . **If** $H_D(G') = \text{'YES'}$ **then** remove another edge from G' (goto step 3), **else** mark edge $\{a, b\}$, return $\{a, b\}$ to G' and remove an unmarked edge:



6. ... eventually the graph will only have n marked edges left, namely the Hamiltonian cycle.

Autumn 1999

13 of 14

If H_D answered 'Yes' given G as input, then R_T will visit the edges of G one by one and for each edge do the following: It will remove an edge, say (a, b) , from the graph and send the resulting G' back to the decision algorithm. If H_D answers "Yes, G' is still Hamiltonian", then we remove another edge and repeats the procedure. In our example the decision algorithm will say "No, my graph is no longer Hamiltonian". Then we know that the removed edge (a, b) is a part of the Hamiltonian cycle. So we place (a, b) back into the graph and label it red, meaning: "Don't touch this edge anymore, this edge is a part of the solution."

Then we look at another unmarked edge, say (e, d) , removes it from the graph and send the resulting graph G'' back to the decision algorithm. H_D says "Yes, this graph is still Hamiltonian". Then we know that we can in fact remove (e, d) from the graph and forget about it, because it is not part of the (last) Hamiltonian cycle that we are looking for.

So our graph becomes simplified. And we keep doing this for every edge, and in the end we end up with only n red-marked edges. Those red edges are a Hamiltonian cycle, so we output them.

This reduction is polynomial because we do polynomial work in each iteration and we make at most one call to H_D for each edge in the graph. If you have a polynomial-time algorithm for decision-HAM, then the whole thing is a polynomial-time algorithm for search-HAM. The point is that you have only $\mathcal{O}(n)$ calls to H_D , so whatever the complexity of decision-HAM is, your complexity is n times that. We are adding a polynomial factor to the complexity, not more. And polynomial factors we don't worry about in this case.

IN210 – lecture 8



Def. 5 Problem L is \mathcal{NP} -equivalent if L is both \mathcal{NP} -hard and \mathcal{NP} -easy.

Examples: Search versions of HAM, CLIQUE, SAT, ... are \mathcal{NP} -equivalent.

Optimization problems – are they harder?

We say that language (problem) L is in class \mathcal{DP} if it can be defined by $L = L_1 \cap L_2$ where L_1 is an \mathcal{NP} language (property) and L_2 is a Co- \mathcal{NP} language (property).

Example: Opt-TSP is \mathcal{DP} -complete. When we say

“ t is the shortest tour in G and t has length= K ”

we say two things:

- There is a tour of length K in G .
(\mathcal{NP} -property)
- There is no tour of length $K - 1$ in G .
(Co- \mathcal{NP} -property)

Question: Is Opt-TSP \mathcal{NP} -equivalent?

Autumn 1999

14 of 14

We say that a problem is \mathcal{NP} -equivalent if it is both \mathcal{NP} -hard and \mathcal{NP} -easy. So we know now that the search version of HAM and SAT and 3DM and all of these problems that we have seen, that they are \mathcal{NP} -equivalent. We know how to prove \mathcal{NP} -hardness. Now we know to prove \mathcal{NP} -easiness also. Those two amounts to a proof of \mathcal{NP} -equivalence.

There is a subtle point here – we are now in fact stretching the definition of \mathcal{NP} -hardness a little bit. Formally we have defined the reduction in an \mathcal{NP} -hardness proof as a mapping between two formal languages such that 'Yes'-instances go to 'Yes'-instances and 'No'-instances go to 'No'-instances. But search problems are not a formal language – they are not solved by algorithms saying 'Yes' or 'No'. Instead they say 'Here is a solution' or 'No, there is no solution'. We want to show \mathcal{NP} -hardness for, say, search-HAM by reducing decision-HAM to it. So we must allow the reduction to map 'Yes'-instances of decision-HAM to 'Here is a path'-instances of search-HAM, and 'No'-instances to 'No, there is no path'-instances. This is a technicality but it illustrates the potential conflict between simplicity and formality in modeling.

We also have optimization problems where we are asking not only for *some* Hamiltonian path or *some* satisfying truth assignment, but we are asking in a way for the best possible of all solutions. We want to find a good way of modeling this new situation where we are asking for the best solution. And I want to show you that this asking for the best, actually has some flavors of not only \mathcal{NP} -ness, but also of Co- \mathcal{NP} -ness. That is important to understand because optimization is an important side of computation – very important.

Let us look at the Traveling Sales Person's problem that should be familiar by now. Opt-TSP asks for the shortest tour in a weighted graph – a tour meaning visiting all the nodes (cities) exactly once. Now, suppose I am an algorithm and I am giving you this kind of shortest tour, and I am saying: "Look, this tour here, this is the shortest tour." Now you measure this tour and you see that it is K kilometers long.

By saying that this tour is the shortest tour, I am really saying two things. I am saying: "Look, there is a tour of length K in this graph". And this 'there is' should remind you of the class \mathcal{NP} . At the same time I am saying: "There is no tour which is of length $K - 1$ in this graph" – because K is just the best one. And this 'there is no' should remind you of the class $\text{Co-}\mathcal{NP}$. So a solution to the optimization variant of TSP involves solving essentially a \mathcal{NP} -complete problem and a $\text{Co-}\mathcal{NP}$ -complete problem.

Without going into details, I will just mention here that optimization problems are properly modeled by a class which is not \mathcal{NP} or $\text{Co-}\mathcal{NP}$, but something that involves both. And the class is called \mathcal{DP} . The class \mathcal{DP} involves languages or properties that can be defined in terms of an \mathcal{NP} language or property, and a $\text{Co-}\mathcal{NP}$ language or property. Actually it is proven that the optimization version of TSP is complete for the class \mathcal{DP} . So it is an example of a DP-complete problem.

I am going to mention a lot of things in this class, and then they will be properly dealt with in the class IN394. There is some advantage in keeping this class kind of informative – kind of a place for people who are interested more in practical thing and for some people who are interested in theory. And then IN394 is the kind of class that is for people who are really interested in theory. So there things are properly proven. Here they are often just mentioned.

The next question for you is whether Opt-TSP is \mathcal{NP} -equivalent or not. That question we will answer in the next lecture.

3.9 Lecture 9

IN210 – lecture 9

non-solvable

Not acceptable

Undecidable

EXP TIME

PSPACE

Co NP

P

NP

intractable

well-solved

= complete problems

Task I

Create map of problems

- abstraction (\rightsquigarrow)
- techniques (diagonalisation/reduction)
- insights
- complete/characteristic problems

Task II

Organize/study problems

- place abstract/real-world problems on the map
- 'walk' the map (get to know the places & 'countries')

Autumn 1999

1 of 13

G&J:
5.1

What is it that we are really doing here? That is always our first question. Now we are going to answer this question by claiming all the way to the top of the pyramid and just looking at the whole class.

We want this class to be a kind of a bridge between the complexity theory and the real world of computation, machines, algorithms, programs and things. Presently this bridge is kind of non-existent, it's broken: The theoreticians live on one side of the river, and the so-called applied people or practical people, programmers and so on, they live on another side. And typically they don't talk much to each other, these people. There are many reasons for that. One of the reasons is a kind of difference in mentality. Theoreticians are people who are interested in nice proofs, abstract things, graphs, complexity classes, and so on – they speak one language. Practical people they speak another language. They talk about programs, about deadlines, about various things. And they think that theoreticians live up in the air – that theoreticians have no connection with earth and that what they do is largely irrelevant. If you swim to the other side of the river and visit the other guys, the theoreticians will tell you that those programmers they know how to program, but they don't really know how to think. Because they sort of just make the program if they can. If they cannot, then they wave their hands and say: "I don't know why this doesn't work."

So the point is to make kind of a bridge to join these two extremes and bring this organized theoretical thinking into reality. And vice versa: Bring the real-

world insights into theory, and see that theory is really just a model of the real world. And this model of the real world can be and should be used to organize all the issues that exist in the real world, so that we end up being an academic discipline, not just a kind of an ad-hoc adventure.

Now we will finally see the overview of the whole IN210 class. The first task of our class was to create this map of classes shown on the foil. And that meant first of all defining and using some abstraction techniques, which allowed us to define abstract objects such as formal languages and Turing machines. Those objects were abstractions or models of real-world objects such as problems and solutions/algorithms.

Then we have learned some techniques which allowed us to organize those abstract formal languages into complexity classes. And those techniques were most notably diagonalization and reduction. We used diagonalization for proving the first difficult problem – difficult being non-solvable or intractable – and then we used reductions for showing that some other problem is as difficult as the first guy.

Once we had those techniques we were able to actually organize some abstract problems into classes, and then look at those problems and gain insights about what sort of problems live in what sort of classes. We got to know the classes from the inside.

The most characteristic problems for a class were the complete problems, which allowed us to understand the spirit of the class. They told us what sort of really hard problems are living in a certain class.

We have seen the class \mathcal{P} that models the well-solved problems. And then we have seen the classes \mathcal{NP} and $\text{Co-}\mathcal{NP}$ with \mathcal{NP} -complete as the interesting hard class. We have also seen $\text{Co-}\mathcal{NP}$ complete problems, PSPACE with its complete problems and Exponential time with its complete problems. We have learned about the Time Hierarchy Theorem which says that there are in fact infinitely many complexity classes.

The big red line in the middle of the map cuts the map in two separate parts – the solvable problems and the unsolvable problems – and that is what we studied first. We have seen that some problems are not even acceptable, while others – like the Halting problem – are acceptable but not decidable. Some problems are more unsolvable than others, so to say.

That is roughly speaking our map. What remains to be done is another three things: Task number two is to see how all sorts of problems can be organized into these maps. And when I say 'all sorts of problems' I mean both the real-world problems and the mathematical problems which are really abstractions of real-world problems. So that when you meet a problem in real life, you know where it belongs. You can place the problem on the map and apply all these machinery that we have developed, to your problem.

So the first thing is placing the abstract or real-world problems on the map. How do we do that? And then another thing is to walk around the map and get to know the places and countries a little bit. It is like complexity theory tourism. We wander around and little by little we get to know the places. We become acquainted with this world of computation and problems.

We will not really be doing these tasks in sequence. We are doing a little bit of this, and a little bit of that all the time – which is more natural. So we will be walking around the map, visiting different places, later on also.

IN210 – lecture 9

**Task III**

Organize/study solutions

- 'classical' (worst-case/best solution) approaches
- 'alternative' approaches to algorithm design/analysis
 - approximation
 - average-case
 - randomized/probabilistic
- 'alternative' machines
 - parallell computers
 - quantum computers
 - ???

Task IV

Organize/study other issues

- logic \leftrightarrow computation
- expressive power of
 - programming languages
 - query languages
 - logic
- cryptography

Autumn 1999

2 of 13

Today I will talk a little more about how to organize problems by placing them on the map. And then we will move on to the third task which is to organize and study solutions or algorithms. We will first study the classical approaches to algorithm design – 'classical' meaning worst-case & best solution.

The idea here roughly is that for each of these areas on the map there will be some approaches to designing algorithms that are appropriate. So that when we place a problem on the map, we will know not only about the difficulty of the problem, but also what sort of approaches would be appropriate for solving it.

What you have seen in all your classes so far, are these classical kind of approaches. Later on we will see some alternative approaches to algorithm design and analysis, which will allow us to actually solve well all sorts of intractable problems that are too difficult by the first approach and by the first criteria. So we will be able to deal with these difficult problems in certain ways. Those ways will be approximation, average-case complexity and algorithms, probabilistic (randomized) algorithms and complexity.

Then we will be looking a little bit at alternative machines. Turing machines represent well the existing "classical" computers by the computational complexity thesis. But what about other possible computers like parallel computers or biological computers or quantum computers – there are all kinds of ideas. The natural question is: How do those ideas correspond to what we are doing in this class?

The fourth and final task, which we unfortunately don't have time to cover

in-dept in this class, will be to organize and study other issues. Now we have studied problems and solutions. What other issues exist? There is a whole issue of thinking or reasoning or logic – defining things by their properties. Let's say: "Give me all the people who work in this organization, who are older then 60 years and who have a salary less then 200.000 kroner per year", because we want to raise their salaries. This is a domain where we are defining properties without necessarily giving instructions on how to pick all the people or all the elements that have that property.

That is the domain of logic, and then we will relate logic and computation. This logic is used quite a bit in computer science also, for example in database query languages or certain kind of programming languages like Prolog. So in general we will be interested in knowing how difficult those questions are, depending on the language in which they are expressed.

So we will be studying the expressive power of programming languages, the expressive power of query languages, and the expressive power of logic. Let me explain right away what this 'expressive power', means:

If you are designing a general purpose programming language, then you would like to say that your programming language is in a way good in the sense that it can define all possible computations.

Imagine you have just designed Java. But there are all kinds of languages in this world. Now you want to say that your language Java is able to express all the computations that Pascal or C++ and so on are able to. Can you say that? You certainly can, because it is enough to simulate the Turing machine – to show that your language can express all the computations that the Turing machine can.

That amounts to proving that your programming language is Turing complete. There are however situations where you want to restrict the power of the language, for example in the database query languages. In the database query languages you would like to be able to say just: "Give me those elements in the database that has a certain property." So you give the user the ability to define properties. You use logic naturally for database queries.

But at the same time you would not like to allow the user to define queries that are too difficult computationally. Because then the program that is answering those queries, might take forever. So there is interest in designing query languages that define all polynomial-time computable queries, and only those. More generally there is interest in defining what a query language – or a kind of logic – can specify in terms of complexity and complexity classes. So that's a large area, and that large area is what we call the expressive power of programming language and the expressive power of logic. We will speak about this in the very last lecture.

And then the final subject will be the big and important area of cryptography, where we are studying the basic issue of encrypting messages and signatures and things, and sending them around the network. This is a very basic problem area because today computers are used in all sorts of transactions, and whether or not computers can properly speaking be used in things like banking – where they naturally can be used and should be used – depends heavily on whether we can trust those messages that come to the other side. You know that anybody can cut a wire and put a piece of hardware in and just transfer the money from yours account to his, and things like that.

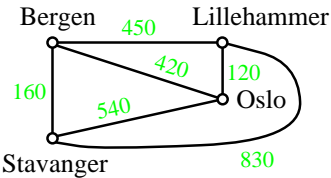
So for this whole big world of networks and computers to be trustworthy we need cryptography, and this is one place in this world where high complexity actually helps us! Because the only way to do cryptography is really through having cheating to be very difficult computationally. And we will see that complexity theory is completely essential there also.

IN210 – lecture 9



Example problem: TSP

TRAVELING SALES PERSON (TSP) is the most studied optimization problem.



Instance: Distance matrix

	Oslo	Bergen	Stav.	Lilleh.
Oslo	0	450	540	120
Bergen	450	0	160	450
Stavanger	540	160	0	830
Lillehammer	120	450	830	0

Question:

- (decision version) Given integer K , **is there** a tour of length K or less?
- (search version) Given integer K , **find** a tour of length K or less.
- (optimization version) Find the **shortest** tour.

Autumn 1999

3 of 13

Now we are back at the second basic task. We are going to see a little bit about how to organize and study problems, how to place them on the map. And the problem that has been studied the most in theory, which is also a kind of a transition problem between real-world problems and abstract problems, is the TRAVELING SALES PERSON (TSP) problem.

Its instance is basically a map of cities and distances. On this foil we have a map with four cities in Norway. And this map is represented in the computer by a distance matrix where we have distances for all pairs of cities. So we know that from Stavanger to Bergen it takes 160 kilometers, and from Oslo to Stavanger 540, and so on.

And then there is this basic question, which is the question of a tour through the cities which visit each city exactly once. The traveling sales person is selling something and going through all the cities at least once a week. And the traveling sales person wants to save gasoline. So the question is the question of the shortest tour.

You see that it is an important question because once the tour is chosen, then possibly every week this tour is going to be taken, maybe every day. So over the years the savings or costs are enormous. So it is actually very important to find this shortest tour.

This problem is like a pattern. Many optimization problems are of this kind – the problems of designing something in the best way, and so on and so forth. So it is actually essential to find the best solution – the shortest possible one.

There are three different variants of this problem. One is the decision ver-

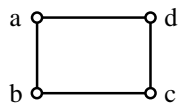
sion, which given an integer K asks: *Is there* a tour of length K or less? Another one is the search version: Given integer K , *find* a tour of length K or less. And then we have an optimization version, which finds *the shortest* tour. This optimization version is what we are typically interested in.

So optimization is more reality, while the decision version is more theory. The decision version is what we model be using the formal languages. So going down this list of questions, we are adding more and more reality to our picture. The question is: How well does this decision version represent the reality (the optimization version)?

IN210 – lecture 9

Is TSP \mathcal{NP} -equivalent?TSP is \mathcal{NP} -hard

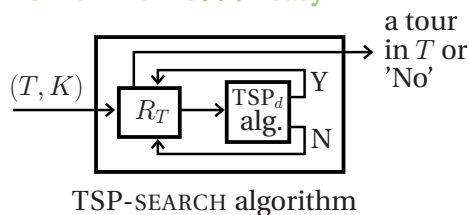
Proof: HAMILTONICITY \propto TSP

	\propto	<table style="border-collapse: collapse; text-align: center;"> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;"></td> <td style="padding: 2px 5px;">a</td> <td style="padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">c</td> <td style="padding: 2px 5px;">d</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">a</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">c</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">d</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> </tr> </table>		a	b	c	d	a	2	1	2	1	b	1	2	1	2	c	2	1	2	1	d	1	2	1	2
	a	b	c	d																							
a	2	1	2	1																							
b	1	2	1	2																							
c	2	1	2	1																							
d	1	2	1	2																							

$$K = n(= 4)$$

TSP-DECISION is \mathcal{NP} -complete

- TSP $\in \mathcal{NP}$: 'guess' tour
- TSP $\in \mathcal{NP}$ -hard

TSP-SEARCH is \mathcal{NP} -easy

Autumn 1999

4 of 13

We have introduced some concepts last time that allow us to deal with more and more realistic problems and questions using our theory. Those concepts were \mathcal{NP} -easy, \mathcal{NP} -hard and \mathcal{NP} -equivalent – those three terms. We have defined them. So we have seen that a language or problem is \mathcal{NP} -equivalent if it is both \mathcal{NP} -hard and \mathcal{NP} -easy.

Intuitively \mathcal{NP} -hard means (at least) as hard as the hardest problems in \mathcal{NP} . \mathcal{NP} -easy means as easy as problems in \mathcal{NP} , meaning: If we can solve problems in \mathcal{NP} efficiently, then we can also solve \mathcal{NP} -easy problems efficiently. So if I reduce my problem to a problem in \mathcal{NP} efficiently, then I have proven that my problem is \mathcal{NP} -easy.

We have seen a technique last time for showing that a problem is \mathcal{NP} -equivalent. This technique can be used for showing that the search version of HAMILTONICITY is \mathcal{NP} -equivalent – this is what we did in the class. The same technique can also be used to show that search versions of CLIQUE, SATISFIABILITY and many other problems, are \mathcal{NP} -equivalent.

The next question now is optimization. Is optimization harder? It does sound to be harder, and in a certain sense it is harder than decision. The reason why optimization is actually harder is that it often involves not only \mathcal{NP} properties, but also Co- \mathcal{NP} properties. And we have seen that Co- \mathcal{NP} properties are in a sense more difficult to deal with than \mathcal{NP} properties.

The optimization version of TSP is asking for a shortest tour t in a graph or network. And the statement that t is the shortest tour, is really saying two things: One, there is a tour of length K where K is the length of the tour t in

your instance. And then another statement is: There is no shorter tour – there is no tour of length $K - 1$. We know that the first statement is an answer to an \mathcal{NP} kind of question, and the second is an answer to a $\text{Co-}\mathcal{NP}$ kind of a question. And we have both.

When I say just 'TSP', then I usually mean the optimization version of TSP. The question now is: Is the TSP problem \mathcal{NP} -equivalent? We will see that "Yes, TSP is \mathcal{NP} -equivalent". The basic insight that will come out of that fact and from the procedure for proving that TSP is \mathcal{NP} -equivalent, will be that although our theory is based on formal languages or decision problems and the Turing machines, the map capture what is essential about complexity. So that when we represent even optimization problems by the corresponding decision problems, we don't lose much. We don't lose the complexity. The difficulty is still there in the corresponding decision problem.

To prove that TSP-decision is \mathcal{NP} -hard, we use the easy reduction from HAMILTONICITY. On the foil is an instance of HAMILTONICITY and the corresponding TSP-instance. We have edges and vertices in the original instance. In the TSP instance we have cities which are the vertices, and we have distances which are 1's and 2's. The distance is '1' if the corresponding pair in the HAMILTONICITY graph is an edge, and '2' otherwise. And then we have K which is n , the number of vertices – 4 in our case. The question: "Is there a tour of the cities of length K or less in this instance?" corresponds to the HAMILTONICITY question: "Can we pick a way of visiting all the nodes in the graph such that we are always walking along the edges, visiting each node only once?" This correspondence is clear because if and only if we can make a Hamiltonian cycle by following edges only – corresponding to roads of distance 1 – then are we able to make a tour that has total distance K , or n .

To finish the \mathcal{NP} -completeness proof of TSP we need to show that TSP is in \mathcal{NP} . That part is easy. A non-deterministic machine can guess a tour of length K , and then easily verify – by adding all the numbers – that that tour is indeed a tour of length K .

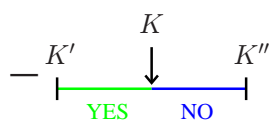
So TSP-decision problem is \mathcal{NP} -complete. To prove that TSP-search is \mathcal{NP} -easy, we construct a TSP-search algorithm by using a TSP-decision algorithm. It gives us a reduction as shown on the foil. That reduction we have already seen. It is the same as the HAMILTONICITY reduction from last lecture. Basically we visit edges one by one, remove the edge and then ask the basic question: Does the graph still have a tour of length K ? If the answer is 'No', then we know that the edge must be included in the tour. If the answer is 'Yes', then it can be removed.

So this reduction is the proof that TSP-search is \mathcal{NP} -easy.

IN210 – lecture 9

TSP-OPT is \mathcal{NP} -easy**Proof:**

- use binary search and TSP-DECISION algorithm to find the optimal K :
 — $K' = 1, K'' = \sum_{i=1, \dots, n \wedge j=i, \dots, n} \text{dist}(i, j)$

— **Repeat:**

If $\text{TSP-dec}(T, \frac{K''+K'}{2}) = \text{YES}$
 then $K'' \leftarrow K$ else $K' \leftarrow K$

- use TSP-SEARCH algorithm to find the tour of length K

 \Rightarrow TSP is \mathcal{NP} -equivalent

Insight: Although based on formal languages (decision problems) and TMs, the 'map' captures/represents what is essential.

Now to optimization. We will show that TSP-optimization problem is also \mathcal{NP} -easy. And the critical part here is finding the K , because if we can find the length of the optimal tour – the shortest possible length, the shortest K – then we can use the search procedure to actually find the tour. So if we can find the K , just the number, then we can use this search procedure that we have just seen, to actually find the corresponding tour.

But how do we find the K ? The problem is that the number of possible K 's is exponential in the size of the instance because all distances are given in binary, not in unary. The instance is a bunch of distances. So we have $\binom{n}{2}$ numbers, distances, in the matrix. And if we add up all those distances, then we get a number which I am here calling K'' . And this K'' is the biggest possible distance that we can imagine occurring – it is definitely bigger than the longest tour.

So this K'' is an upper bound on the longest tour. But since we represent our numbers in binary notation, then this K'' is exponential in the size of those numbers, and in the size of the instance. So we are to begin with dealing with a number which is too big for us. So we cannot just do exhaustive search from K'' to 1 – we cannot just call TSP-decision for each possible K from K'' to 1 to find K , the length of the smallest tour.

The solution to the exponentiality of this number problem, is the standard one – the standard one being the binary search. So to find K we use the decision procedure for TSP and the binary search.

All binary searches go like this: If we have the range from K' to K'' to begin with, we look into the middle point and ask the question, whatever the question

is. In this case the question is: Is there a TSP tour with this K and the original instance? If the answer is 'Yes', then we know that we can reduce the range to the first part. If the answer is 'No' and the answer was 'Yes' for K'' , then we know that we can reduce the range to the second half.

So this binary search allows us to deal with this exponential large number in polynomial time ($\log 2^n = n$) and find the right K . Once we have the right K , we use the TSP-search procedure to actually find the tour that has length K .

This whole thing is the proof that TSP is \mathcal{NP} -equivalent, leading to a basic insight that in a certain sense optimization problems will tend to be not harder than \mathcal{NP} -problems (decision versions of the problems) in a certain sense.

IN210 – lecture 9

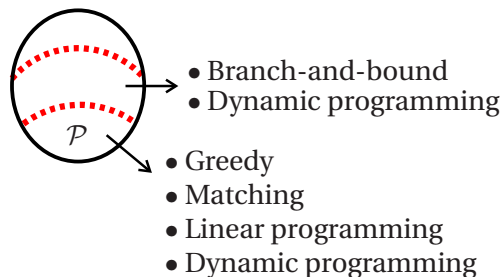


Focus on task III: Organize algorithms/solutions

We look at different algorithm-design paradigms:

- Today: 'Classical' worst-case/best solution approaches
- Next three classes: 'Alternative' approaches

Worst-case/best solution approaches



Autumn 1999

6 of 13

Now we shift focus to task 3, which is: We organize algorithms or solutions by using our map. We have all kinds of algorithms in this world. For every solvable problem there are in fact a number of algorithms. So what can we do to say something general about algorithms?

The key notion here will be the one of algorithm-design paradigm. We will see that there are some basic approaches for solving problems that are used over and over again. And we will also see that some approaches are more appropriate for those intractable or hard problems, while others are more appropriate for those easy or well-solved or polynomial-time problems. So we will take just a quick look at some of these algorithm-design paradigms, and try to understand what they are and how they are used to deal with different classes of problems.

The big challenge now is: When you meet a problem, how do you actually go about finding the solution to the problem? And naturally we build on top of what we already have. So you would look at the problem and try to figure out what the complexity of the problem is – in other words, where on our map the problem is coming from. So you meet a problem and you ask: "Where are you coming from?" And the problem says: "I was born in \mathcal{P} . I am a \mathcal{P} problem."

Then you know how to deal with the problem, more or less. You know that for your \mathcal{P} problem you will be able to find an exact solution even in the worst case – even if your instance is the worst possible instance, because he is coming from a nice country where the nice people live.

To actually solve the problem in a good way, there are all kinds of approaches.

And what we are trying to do now is to systematize at least some of those approaches, and show you some kind of meta-approaches, some basic ways of organizing things. So that you can by seeing just those few ways really have a good understanding of what is involved in designing polynomial-time algorithms on the one hand, and exponential algorithms that are still good. Sometimes we need exponential-time algorithms for solving those intractable problems.

We will look at the following worst-case & best-solution approaches for solving problems in \mathcal{P} : greedy algorithms, iterative improvement (matching), linear programming and dynamic programming. For solving intractable problems as fast as possible, one usually uses branch-and-bound techniques and/or dynamic programming. It is still exponential time, but with as little extra work as possible.

IN210 – lecture 9



Subset systems

A **subset system** is a triple (E, w, τ) where

E is a set

w is weights on the elements, $w : E \rightarrow \mathbb{Z}^+$

τ is a system of subsets closed with respect to inclusion (any subset of a set s in τ is also in τ): $s \in \tau \wedge s' \subseteq s \Rightarrow s' \in \tau$

Combinatorial problem associated with a subset system:

Find $s \in \tau$ with largest sum of weights.

Example: MAX. SPANNING TREE

In MAX. SPANNING TREE we ask, given a weighted graph G , for the **subtree** (subgraph without cycles) of G that has the highest sum of weights.

MAX. SPANNING TREE as a subset system:

E is the set of edges in G .

w is weights of the edges.

τ is the set of subtrees of G . τ is closed with respect to inclusion, because removing an edge from a subtree of G , we still have a subtree of G .

Autumn 1999

7 of 13

The challenge is to try to systematize things that are right now scattered in all sorts of books and all sorts of chapters that correspond to different problems. We don't want different problems, we want to unify things. So we unify things by defining something which is called the *Subset System*. And the subset system is a triple, three things. One is a set E – think of edges. Another one is a bunch of weights we associate with each element of E . The weights are positive integers. And then τ (tau) is a system of subsets of sets from E that is closed with respect to inclusion, meaning: If s is in τ and s' is a subset of s , then s' is also in τ . So τ is a nice kind of closed system.

You don't need to worry too much about details. Just try to follow the basic ideas. The basic idea here is that a lot of problems that we have seen, such as MINIMUM SPANNING TREE, MAXIMUM MATCHING, CLIQUE or TRAVELING SALES PERSON problem and so on, have this basic form. We are given a set, some kind of edges. And then we have weights associated with edges. The weights can be all '1' – then we get HAMILTONICITY instead of TSP. And then we have some kind of criteria for choosing subsets – that's τ . The subsets are the feasible solutions. In CLIQUE we know what a clique is, in TSP we need a tour of length K or less and so on.

The basic combinatorial problem associated with the subset system is: Find a subset s in τ with the largest possible sum of weights. This 'largest' you can normally think of as 'smallest' also – the approach will be completely similar.

So a familiar example, something that you have seen in your IN115-class, is the MINIMUM SPANNING TREE problem. I will use a related problem, the MAXI-

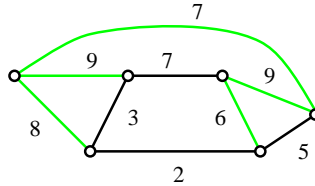
MAXIMUM SPANNING TREE (MST), as an example here. In MST you are asked, given a graph G with positive weights on the edges, to find the subtree of G with the highest sum of weights. A subtree of G is a subgraph without cycles.

MST can be viewed as a subset system: E is the set of edges in G , w is the weights of the edges and τ is the set of all subtrees of G . τ is closed with respect to inclusion because removing an edge from a subtree of G , still give us a subtree.

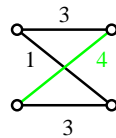
IN210 – lecture 9

**MAX. SPANNING TREE**

- solved by a greedy algorithm in time $\mathcal{O}(n \log n)$:

**MAX. MATCHING**

- greedy algorithms don't work
- solved by an "iterative-improvement" algorithm in time $\mathcal{O}(n^3)$

**TSP**

- \mathcal{NP} -complete
- best-known algorithm is exponential time (exhaustive search)

Autumn 1999

8 of 13

I will just review the basic algorithm for finding the maximum spanning tree. It is the greedy algorithm: You order the edges by weight. Then you visit the edges in the order of decreasing weight, starting with the edge with the largest weight. You construct the maximum spanning tree as you go along – beginning with the empty tree – by looking whether or not the edge in G that you are visiting will complete a cycle. If it doesn't complete a cycle, it is included in the spanning tree.

So in the example on the foil we start with one of the weight 9 edges. It can be included, and also the other weight 9 edge. The weight 8 edge is also fine. Next we include one of the weight 7 edges, but we cannot include both, because then we get a cycle. The weight 6 edge is fine, but none of the remaining three edges can be included without destroying the tree property. So the maximum spanning tree consists of the five green edges.

This is called a greedy algorithm because it never undoes what it has once done. It does what is best in the given situation without looking globally, just looking locally. And it can be proven that this is good enough for solving the MST problem.

So certain problems can be solved by greedy algorithms. Certain other problems cannot – for example the MAXIMUM MATCHING problem cannot be solved by a greedy algorithm. Here is a proof by counterexample:

On the foil is a little instance of the maximum matching. This is the optimization version of the MATCHING problem where we are asking for the matching that maximizes the sum of weights of the elements in the matching. What

you would do with a greedy algorithm is you would pick the edge with largest weight first. So here is an edge with weight 4. You pick that edge and you mess up the solution forever because the largest matching is the one that has these two horizontal edges – 3 and 3. Those two edges gives a total of 6, but once you marry the two people with weight 4 then you have to marry these two other people also, and they carry a weight of 1 only. So the greedy algorithm finds a matching of weight 5 only.

An efficient algorithm for finding the best matching (remember the story about Edmonds from the very first lecture) will begin somewhere, maybe marry these two people of weight 4, and as I briefly showed last time, then it would find a better matching by finding the alternating path. So it is a kind of a gradual-improvement strategy as supposed to the just-assembling-things strategy as in the greedy algorithm.

The greedy algorithms are very efficient, typically linear time or maybe time $\mathcal{O}(n \log n)$ (n is the size of the input, typically $|V| + |E|$ for graphs) because you need time for sorting the edges. This gradual-improvement algorithm for matching is maybe $\mathcal{O}(n^3)$ or something like that. And then there are problems such as TSP that are kind of similar in flavor, but which don't seem to be solved by any polynomial-time algorithm. The best known algorithms for those \mathcal{NP} -complete problems require exponential time, because they basically tries all $n!$ Possibilities.

IN210 – lecture 9



Generic greedy algorithm

```

 $I = \emptyset$ 
while  $E \neq \emptyset$  do
  Pick largest-weight element  $e$ 
  of  $E$ ;
  Remove  $e$  from  $E$ ;
  If  $I + e \in \tau$  then  $I := I + e$ ;
end_while

```

A subset system whose combinatorial optimization problem can be solved by a greedy algorithm is called a **matroid**.

Autumn 1999

9 of 13

Next, we define a generic greedy algorithm for the solution of our generic problem – the generic problem being the one that was associated with the subset system. So this generic algorithm is basically what we have just seen for the MAXIMUM SPANNING TREE. Initially the solution I is the empty set. And then we go through edges in the order of decreasing weight. So we pick the edge with the largest weight.

For some problems where we want to minimize the weights, like the MINIMUM SPANNING TREE, largest here could also be the smallest.



We pick the largest-weight edge e and remove it. If your I and that edge e gives a set that is still in τ , then we put e into I , otherwise we throw it away. You can think of elements of τ as being feasible solutions such as partial matchings or partial spanning trees or tour segments. So if the current solution plus e can still become a good solution, put e into the partial solution.

So this is just the basic greedy algorithm with ' I ' being the bag into which you just put things. And then once the algorithm is finished, you have in your bag exactly the best solution, if the greedy algorithm works.

A *matroid* is the kind of problem for which the greedy algorithm works, by definition. So a subset system whose combinatorial optimization problem can be solved by a greedy algorithm, is called a matroid.

(This is a blank page)

IN210 – lecture 9


Insights


1 SAT	2 SAT	3 SAT
1 DM	2 DM	3 DM
1 COLORING	2 COLORING	3 COLORING
↓	↓	↓
1 matroid	intersection of 2 matroids	intersection of 3 matroids
↓	↓	↓
greedy solutions	generic matroid intersection algorithm	\mathcal{NP} -complete

TSP as an intersection of 3 matroids

(directed graph G , and a TSP-path)

1st matroid: Same as for MIN SPANNING TREE

2nd matroid: Subtrees where each node has at most 1 outgoing edge (all edges in G get weight=1)

3rd matroid: Subtrees where each node has at most 1 ingoing edge (all edges in G get weight=1)

In addition the intersection tree (solution) must have exactly $n - 1$ edges.

Autumn 1999
10 of 13

Now we have a formalism for dealing with all sorts of problems. And we can notice a kind of interesting phenomena: 1SAT is a completely trivial problem, 2SAT – the interesting problem – is solvable in polynomial time, while 3SAT is \mathcal{NP} -complete and seems to require exponential time. 1DM is trivial, 2DM – our standard MAXIMUM MATCHING problem – is interesting and solved by a polynomial-time algorithm, while 3DM is \mathcal{NP} -complete. 1-COLORING is trivial – it is basically graph connectivity, which solved by the greedy algorithm. 2-COLORING is basically matching: Can a graph be colored by 2 colors such that no adjacent nodes get the same color? 3-COLORING is \mathcal{NP} -complete.

So there seems to be some kind of regularity in this whole. We can impose order through using our basic machinery of matroids: We can see these 1-problems as matroids – which they are – and these 2-problems as being intersections of two matroids. And there is a proof to the fact that an intersection of two matroids is solved in polynomial time. There is also a generic matroid-intersection algorithm which is the kind of iterative-improvement algorithm that is used for maximum matching. So if we see our problem as a kind of an instance of an intersection-of-two-matroids problem, then we know the basic approach that works for that problem. And that's this general matroid-intersection algorithm.

Let me use matching as an example: Intuitively you think of matching as satisfying two sets of criteria. On the one side you want to marry the girls exactly once. If that's your only requirement – meaning that the boys can be married multiple times – then you have one matroid. But if you want the girls to be

married once and the boys to be married once, then you have an intersection of two matroids because you are satisfying two conditions.

It turns out that TSP can be defined as an intersection of three matroids. In the case of a directed graph and asking for a TSP path, the three matroids could be as follow: The first matroid is the same as for MINIMUM SPANNING TREE, namely subtrees where we are asking for the smallest total edge weight. The second matroid is subtrees where each node has at most one outgoing edge. Here we consider each edge in G to have weight=1, and we are asking for the largest weight sum, which is equivalent to asking for the largest number of edges. The third matroid is subtrees where we again consider each edge to have weight=1, but now we require that each node has at most one ingoing edge.

These 3 conditions together with the extra requirement that the intersection tree, the solution, must have exactly $n - 1$ edges, give us an \mathcal{NP} -complete problem. This extra requirement that the solution must have exactly $n - 1$ edges is necessary to prevent tours consisting of separate cycles/paths being legal solutions.

It would be useful for you to see the maximum matching algorithm in some detail. I believe that is done in the groups eventually, and that together with what we have done here, will give you enough insight into a certain really broad class of polynomial-time algorithms. This is essential here because we want to be able to recognize an algorithm that is polynomial time, and to recognize a problem that is polynomial time, and to understand how to construct polynomial-time algorithms.

IN210 – lecture 9

**LINEAR PROGRAMMING (LP)**

In the LP we want to

- minimize $\vec{c}\vec{x}$ (i.e. $c_1x_1 + c_2x_2 + \dots + c_nx_n$) such that
- $A\vec{x} \geq \vec{r}$ (A is a $m \times n$ integer matrix, \vec{r} is a m -vector of integers)
- $\vec{x} \geq 0$ (x_i are real numbers)

Example: Diet problem

\vec{x} = diet

i = food type

x_i = amount of food i

c_i = cost of food i per unit

A_{ij} = amount of nutrient j in food i

r_j = required amount of nutrient j

- **Simplex algorithm** is not polynomial in worst-case, but efficient “in practice”.
- **Ellipsoid algorithm** is polynomial in worst-case, but not as fast as Simplex “in practice”.

Autumn 1999

11 of 13

The greedy algorithms and the matching-type algorithms actually cover a very large class of algorithms. Another large class of algorithms and problems has to do with number problems, or with numbers. So a lot of problems either are of this following kind or can be written or represented in this form. This format is called LINEAR PROGRAMMING (LP).

Here is how an instance of a LINEAR PROGRAMMING problem can be formally stated: The problem is to minimize $\vec{c}\vec{x}$ – where \vec{c} and \vec{x} are vectors, not just numbers. So this really means $c_1x_1 + c_2x_2 + \dots + c_nx_n$ – it is the scalar product of two vectors.

So we are minimizing this scalar product subject to some conditions. The first condition is that A – a $M \times n$ integer matrix – times \vec{x} should be greater than or equal to \vec{r} , a vector. This means that $A_{j1}x_1 + A_{j2}x_2 + \dots + A_{jn}x_n \geq r_j$, for all $1 \leq j \leq m$. The last condition is that \vec{x} , meaning all entries x_i , should all be non-negative real numbers.

Let me tell you intuitively what this means, and then you will understand better what this really is, even if you are not very familiar with this formalism of matrices and vectors.

Think of vector \vec{x} as being a diet. x_i is the amount of food type i in the diet. So x_1 is the amount of bread, x_2 is the amount of carrots, x_3 is the amount of milk, and so on. Then think of c_i as being the cost per units of food type i .

So these \vec{x} -es are, say, what you eat through the year. And you want to spend as little money as possible on food, that’s the basic optimization in this world. We stick to it. You may prefer something else – you may prefer better nutrition

or something – but we for the time being minimize money.

And your c -costs multiplied by the amounts will give you the money spend on food during the year, but you do want to satisfy some basic requirements for nutrition. So you don't just want to starve yourself. You are saying: I need this much of vitamin A and this much of vitamin B, this much of protein, and this and that.

So A_{ij} will tell you how much of nutrient j there is in a unit of food i , and then r_j is the required amount of nutrient j in a year. So that you know that your A times \vec{x} – your nutrient contents times the amounts of foods – must be at least as big as the required minimum amount of nutrients. And then you also know that you are choosing your food in such a way that you can just eat a positive amount. So the amounts of food cannot be negative.

This is a very basic problem, and then again a lot of problems can be defined in this form. So LINEAR PROGRAMMING is a very interesting kind of way of looking at problems, and a very potent way of solving problems. Since many problems can be embedded into LINEAR PROGRAMMING, then if we have a good way of solving linear programming problems, then we can solve all sorts of problems by that way.

It turns out that there is actually a good way of solving linear programming problems, and that way is what is called the Simplex algorithm, which I believe you have never seen. That is kind of unfortunate because it is a very interesting algorithm and a very interesting theory.

There are two things to know about the Simplex algorithm. One thing is that it is efficient in practice – really works well for practical problems. The other thing is that in the worst case it is not polynomial. Those worst cases don't really ever appear in practice, but for a long time the situation was such that LINEAR PROGRAMMING was solved by the Simplex algorithm and other variants of this approach, but it was not known whether LINEAR PROGRAMMING is in \mathcal{P} – whether a good worst-case solution exists.

But around 1979 the Ellipsoid algorithm was constructed. The Ellipsoid algorithm is a very complicated algorithm which you probably would never actually want to use in your life. There are better versions of this same approach that actually lead to good solutions, but this algorithm is from a practical point of view very complicated and inefficient. The theoretical important point about the Ellipsoid algorithm is that it is a polynomial-time algorithm. And this algorithm was a proof that LP is in \mathcal{P} .

So LINEAR PROGRAMMING is solvable in polynomial time, and a lot of difficult problems that have this kind of borderline flavor have been solved in polynomial time by similar approaches.

IN210 – lecture 9



INTEGER LINEAR PROGRAMMING (ILP)

- minimize $c\vec{x}$ such that
- $A\vec{x} \geq \vec{r}$
- $\vec{x} \geq 0$
- x_i are integers

Theorem 1 ILP is \mathcal{NP} -hard.

Proof: Reduction from TSP

- distance matrix with elements c_{ij}
- Variable x_{ij} associated with edge (i, j)
- $x_{ij} = 1$ if edge (i, j) in the tour, and 0 if not
- minimize $z = \sum_{i,j=1;i \neq j}^n c_{ij}x_{ij}$
- $0 \leq x_{ij} \leq 1$ and x_{ij} integer
- $\sum_{i=0}^n x_{ij} = 1$, for $j = 0, \dots, n$
- $\sum_{j=0}^n x_{ij} = 1$, for $i = 1, \dots, n$
- + a constraint that excludes tours consisting of “subtours”

Autumn 1999

12 of 13

It turns out that adding one more little condition to LINEAR PROGRAMMING – namely that the \vec{x} -values in our solution must be integers – changes the complexity completely. When we are talking about foods, then of course you can eat 1.79 kilos of potatoes, that is no problem more or less. So usually you are not required to eat integer amounts of food. But if your problem is such that you cannot divide things into small chunks where you have to either take something or not take something, then it turns out that the problem becomes harder. INTEGER LINEAR PROGRAMMING (ILP), as it is called, is in fact \mathcal{NP} -hard.

This is a rather shocking result, because intuition tells us that ILP should be at least as easy as LP because there are less feasible solutions when the \vec{x} -es have to be integers. And a fewer number of feasible solutions normally means faster algorithms because there are fewer cases to consider. But we will see on the next foil why intuition fails us this time.

ILP can be proven \mathcal{NP} -hard and on this foil I have sketched a reduction from TSP. It turns out that TSP, SAT and all sorts of other \mathcal{NP} -complete problems can be written as integer programs fairly easily. So that essentially the theory of INTEGER LINEAR PROGRAMMING, and even of LINEAR PROGRAMMING, can be applied to those problems.

If we have some good ways of solving INTEGER LINEAR PROGRAMMING, then we can really solve all kinds of problems in that same, good way. And that is actually what this world of problems currently looks like, very much. A good solution to TSP – ‘good’ mainly in the sense that they work fairly well in practice – are constructed today by using the INTEGER LINEAR PROGRAMMING theory

and approaches.

I am not going to speak about this a lot, but I just want to say that there is a whole world out there, so it is interesting for us to see how in fact a problem like TSP can be embedded into this theory, how it can be represented. And let me just give you some hints.

In TSP we have a distance matrix. Those distances will be the c_{ij} -values. We are going to choose some edges and not choose others. So we define ILP variables x_{ij} , which will tell us about whether the edge from node i to node j is in the TSP-tour or not. So variable x_{ij} is going to be 1 if edge (i, j) is in the tour, and 0 if (i, j) is not in the tour.


We will write our ILP problem by saying: Minimize z equal the sum of $c_{ij}x_{ij}$ for all possible i 's and j 's, with conditions that the x_{ij} -values must be between 0 and 1. If we also say that x_{ij} 's are integers, then that really means that x_{ij} 's are either 0 or 1.

And then we want to say that we are taking an ingoing edge once for every vertex in the graph. In ILP this can be expressed by demanding that the sum of x_{ij} 's for $i = 1, \dots, n$ is equal to 1. This sum must be 1 for all choices of j -values. Similarly, if we for each i -value demand that the sum of x_{ij} 's for $j = 1, \dots, n$ is equal to 1 then we express the fact that we are taking an outgoing edge once for each vertex.

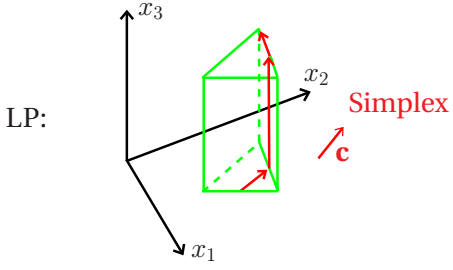
These two last constraints are modeling a simple cycle, because each node in a cycle has exactly one ingoing edge and one outgoing edge. But we need an additional constraint that excludes tours or cycles that consist of a number of "subtours", meaning two or more disjunct cycles.

Together this whole thing is an embedding of TSP into INTEGER LINEAR PROGRAMMING, and at the same time it is proof that ILP is \mathcal{NP} -hard. We have just reduced the TSP to INTEGER LINEAR PROGRAMMING.

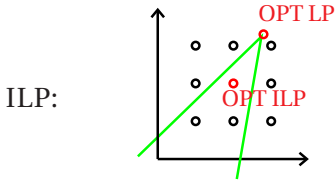
IN210 – lecture 9



Geometric interpretation

LP: 

- It can be shown that all feasible solutions (= points in a n -dim. space) must be inside the **convex hull**.

ILP: 

Algorithm for ILP:

- solve LP-opt (use Simplex)
- find ILP-opt (it should not be too far away)

Note: This algorithm is not pol. time because the ILP-opt point can be exponential far away from LP-opt point given by Simplex/Ellipsoid (arbitrary small angle . . .).

Autumn 1999 13 of 13

A way to think of LP and ILP, and to understand the different approaches for solving those problems, is actually through geometry. It is a very natural and nice way to gain insights and to organize those insights.

Let me tell you how that works: The critical part in these problems is the x_i variables. Think of them as being coordinate axes. So how much of these foods we take, that's determined on the different coordinate axes. And the optimal solution is expressed as a point in the coordinate system.

Now, what about the conditions? What do they look like in this coordinate system or solution space? The overall condition is that $A \times \vec{x} = \vec{r}$. But because \vec{x} and \vec{r} are vectors, this is really M different conditions of the form $a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n \geq r_i$. If we say equals r_i instead of greater than or equal r_i , then this whole condition is a hyperplane in this solution space – it's a plane in a n -dimensional space. So each of these M equations defines a plane.

An inequality defines a half-plane. So with each condition, it is as if we take a sword and cut through the space – dividing the space into one part which is the good solutions, the feasible part, and the other part which is thrown away. And by cutting this way and this way and this way we cut the space many times, and what remains is what is called the convex hull. It's a kind of a body, an n -dimensional body.

So this convex hull, this n -dimensional body, defines your feasible solution space. I have drawn a little example in the 3-dimensional space on the foil. Your best solution must be found inside of this space because it is only points inside this space that satisfy all M conditions. How do we find the best solution?

We look at the cost \vec{c} , which is a vector in this space. The cost vector \vec{c} defines a direction – it is pointing in one direction. And then what you do is you put yourself inside of this n -dimensional body, this convex hull, and you go into the direction of the cost vector as far as possible until you reach a wall and cannot go any further in that direction. Then you sort of slide along the walls until you find yourself in a corner and you say: "This is the best, I cannot go any further."

This is the simplex algorithm. The Simplex algorithm would go along the edges, knowing that the best solution must be in one of the corners. So it is going along the edges, always going more and more in the direction of vector \vec{c} , until it is stuck in a corner – and that corner point is the best solution.

The reason why the simplex algorithm is efficient is because it is natural, usually in a few steps you find yourself in the right corner. The reason why the simplex algorithm is theoretical inefficient is because when you cut with polynomially many planes, you can get exponentially many edges. So you can always construct pathological kind of instances of LP for which the simplex algorithm doesn't work because it will slide along exponentially many walls before it get to the corner of the best solution.

However, if we introduce the requirement that the solutions are integers then we are in deep trouble, as shown on the foil. Because even if we find the optimum for the corresponding linear program – which is this red point here – that point can be arbitrarily large from the actual optimum of the ILP – which must be an integer point inside the feasible space, also shown in red on the foil.

There can be other points which are near to the optimum of ILP, but those points can be infeasible because they are outside the convex hull. And because we can make this angle arbitrarily small, the optimum of ILP can be exponentially far away from the LP solution.

That is the bad news. The good news is that in practice, if you can solve the LP problem, then you will find good solutions to the ILP in the neighborhood.

That's one way of looking at this. Another way of looking is that this whole theory that has been developed around these LP/ILP problems is of a great help in finding approximate solutions, or even relatively fast solutions to \mathcal{NP} -complete problems like TSP. There is a kind of a world championship in how big instances of TSP you can solve, and that sort of issues is dealt with by using this theory.

Next time I am going to finish this review of classical approaches to algorithm design by telling a few words about branch-and-bound and dynamic programming, and then we will move on to alternative approaches to algorithm design.

3.10 Lecture 10

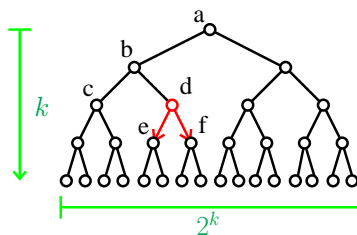
IN210 – lecture 10



Classical algorithm-design paradigms (cont.)

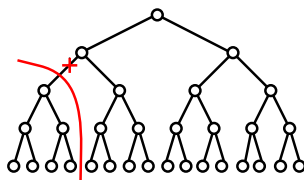
Branch-and-Bound

Branch:



Leaf nodes = possible solutions

Bound:



- Backtracking
- Pruning ('avskjæring')

G&J:
4, 6.0-6.2

Autumn 1999

1 of 22

We have organized solutions or algorithms according to paradigms. We have seen that certain ways of constructing algorithms work well for certain classes of problems – one kind of approach for polynomial time, another kind for problems that we believe cannot be solved in polynomial time. Today we will see also that some approaches (i.e. dynamic programming) are suitable for both \mathcal{P} problems and \mathcal{NP} -complete problems.

Last time we have seen a bunch of design paradigms for problems in \mathcal{P} . Now we are going to see two design paradigms that are widely used for problems that are not known to be in \mathcal{P} , such as \mathcal{NP} -complete problems. Those two paradigms are *branch and bound* and *dynamic programming*. I am going to introduce them just very briefly and schematically. You are going to see more details in the group practices, hopefully.

First we look at branch and bound. I have drawn a kind of a tree on the foil, and you can think of this tree as being some record of all possible computations. The leaves of the tree are all possible solutions to a certain problem, say all possible orderings of vertices or all possible TSP-tours. And those solutions are determined by a bunch of individual choices: You start from a certain node and then you have a certain number of neighbors, and for each of those neighbors you have more neighbors and more neighbors. What choices you make determine the tour. So the root node in the tree represent the first city you visit. If that city has two neighboring towns, then you have two roads to choose be-

tween – represented by two outgoing edges from the root node in this search tree.

You can think also of this tree as being really the computation of a non-deterministic Turing machine. We think of the computation of a NTM as being a tree of configurations, and k as being the non-deterministic polynomial time.

What a branch-and-bound algorithm would do, it would explore all these possibilities, but in a clever way so that it doesn't have to do the same work twice, and so that it doesn't investigate hopeless partial solutions. That's roughly the idea.

There are two characteristic parts of a branch-and-bound algorithm: One is the branching part where an algorithm would explore this three of possibilities in some organized way, say depth-first. And that is already some saving because then it doesn't compute the same prefix path twice, meaning: If the algorithm has followed the path $a - b - d$ and is currently at node d , then it will first try to go to node e . After trying all possibilities from node e , the algorithm somehow has to come back to node d and try the other possible path to node f . A branch-and-bound algorithm doesn't start all over again from node a , computing the $a - b - d$ part again. It reuses that prefix path $a - b - d$. And it does so by *backtracking* from node e to node d , and then going the other way to node f .

So this is how the algorithm branches back and forth, and it is a little bit clever, but not so very clever. What is actually a lot more important here in branch and bound is the 'bound' business. Bounding means that you can come up to a certain partial solution and realize that below there are no really great solutions – 'great' meaning 'not good enough' for optimization problems and 'not a correction solution' for decision problems.

What you do is you produce a kind of a *bound* on the value of the solutions that begin with what you already have. And then you use this bound or estimate, maybe comparing it with the best solution found so far, to decide whether you really want to explore these solutions further or not. For example if we are talking about HAMILTONICITY, then if you are looking at tours beginning with the three vertices a, b, c in that order and (b, c) is not an edge, then it doesn't make sense to look at further solutions beginning with a, b, c . Because (b, c) is already not an edge, so there are no Hamiltonian cycles here.

Or in a TSP problem maybe you would discover a road which is a kind of a bumpy mountain road that is 500 km long between city b and c . And you know that there is this freeway that is really just 100 km long, that would first go from b to d and then from d to c . So you know that the solutions that take this mountain road they are all going to be bad solutions, and you sort of drop this possibility of going from b to c right away and favor going to city d .

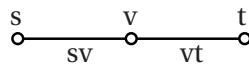
So you just drop – *prune away* – this part of the search tree. And if you do this systematically, then hopefully you end up exploring much fewer solutions than all permutations.

IN210 – lecture 10



Dynamic Programming

- Building up a solution from solutions from subproblems
- Principle: Every part of an optimal solution must be optimal.



Now we compare this branch-and-bound idea with the idea of dynamic programming, which is in a certain sense similar, but in another sense opposite from it. Dynamic programming will not branch and bound. DP will build a complex solution from smaller solutions. So it is usually some kind of building-up process.

So in DP we build up a solution to a problem from solutions to subproblems. There is a kind of a principle which is sometimes called the principle of Dynamic Programming, but it is not really a formal principle. It's more an intuitive idea. The DP principle says that if something is an optimal solution, then every one of its parts must be also optimal.

If we are looking, say, at the shortest path in a graph from point s to point t , and this shortest path goes through a vertex v , then we know that sv must be the shortest way to get from s to v and vt must be the shortest way to get from v to t , otherwise there would be a shorter path from s to t .

So if we build shortest paths between nearby cities, then we use those shortest paths to build paths between further-away cities, and so on. This is how Dijkstra's algorithm for finding the shortest path in a graph works. Dijkstra's algorithm is dynamic programming applied to a problem in \mathcal{P} . We will see dynamic programming applied to an \mathcal{NP} -complete problem in a moment.

With this we finish the part on the classical worst-case & best-solution algorithm design and analysis paradigm.

(This is a blank page)

IN210 – lecture 10



Managing Intractability

- **Idea:** Perhaps the hard instances don't arise in practice?
- Often **restricted versions** of intractable problems can be solved efficiently.

Some examples:

- CLIQUE on graphs with edge degrees bounded by constant is in \mathcal{P} :
const. $C \Rightarrow \binom{n}{C} = \mathcal{O}(n^C)$ is a polynomial!
- Perhaps the input **graphs** are
 - planar
 - sparse
 - have limited degrees
 - ...
- Perhaps the input **numbers** are
 - small
 - limited
 - ...

Autumn 1999

3 of 22

We move on to some ideas on managing intractability. Suppose you encounter an \mathcal{NP} -complete problem, and you prove that your problem is \mathcal{NP} -complete. What then? The problem still has to be dealt with; it still has to be solved. Of course, if the problem instances are small and if it is extremely important to have an exact solution in the worst case – in all possible cases – then you may look at one of those two approaches: branch and bound and dynamic programming.

However if the instances are large, then those algorithm will take forever because they are exponential time. But you still want to do something with the problem, and the question is what. We are now going to look at some possibilities.

The first thing that one would think about is: "Let's look at the problem carefully once more. maybe the problem is not as hard as it seems." And the point of this is that it happens quite often that the instances of the problem that arise in practice have some natural restrictions which make the problem easy.

And if you look at your G&J textbook then you would see that with every problem there is a list of restrictions and statements explaining for what kind of restrictions your problem remains \mathcal{NP} -complete, and when the problem is solvable in polynomial time. So you would typically look at your practical problem, see what restrictions apply and then look at something like the G&J textbook or investigate the problem yourself – trying to see whether the problem remains \mathcal{NP} -complete or \mathcal{NP} -hard under those restrictions.

As an example we will look at the CLIQUE problem, which is deciding whether

your input graph has a clique of size K – a 'clique of size K ' being a complete subgraph with K vertices. If the degrees in your graph – the number of edges from any vertex – are bounded by a constant C , then your problem is always solvable in polynomial time, no matter what K is. This is so because if your degrees are bounded by a constant C , then the largest clique that can be in your graph is C . If your K is larger than C , then you can immediately answer "No, there is no clique of size K in this graph." And if K is equal to C , which is the most difficult case, then you just try all possible sets with C nodes and see if any of them is a clique. How many C -sets are there in a graph with n nodes? The number of possible C -sets are $\binom{n}{C}$, which is $\mathcal{O}(n^C)$. So there are only this many possible cliques of order C , and if C is a constant, then this is a polynomial.

So with this restriction CLIQUE is solvable in polynomial time, and it is a very natural restriction because in practice you don't see very messy graphs. Quite often the graphs are bounded by a constant, or they are planar or in some sense sparse. So you can use these restrictions and see whether your problem remains \mathcal{NP} -hard or difficult under them, and then find good algorithms that solve the problem maybe in polynomial time with those restrictions.

This is if you have graph problems. If you have number problems, then the natural restriction is that the numbers that would make your instance hard if they are astronomically large, in practice end up being somehow limited – maybe small, maybe limited by a constant and so on.

IN210 – lecture 10



Pseudo-polynomial algorithms

Def. 1 Let I be an instance of problem L , and let $\text{MAXINT}(I)$ be (the value of) the largest integer in I . An algorithm which solves L in time which is polynomial in $|I|$ and $\text{MAXINT}(I)$ is said to be a **pseudo-polynomial algorithm** for L .

Note: If $\text{MAXINT}(I)$ is a constant or even a polynomial in $|I|$ for all $I \in L$, then a pseudo-polynomial algorithm for L is also a polynomial algorithm for L .

Autumn 1999

4 of 22

We will now look at these possibilities to manage number problems a little more carefully, because they give us an interesting class of algorithms that are almost polynomial, and that in practice are quite often just polynomial-time algorithms for solving problems that are intractable in the worst-case. Those are the so-called pseudo-polynomial algorithms – algorithms that just about pretend to be polynomial. They are pseudos but in practice they may work very well.

Let me first define what these pseudo-polynomial time algorithms are. Basically you can look at the number value of the largest integer in the instance I . This integer we call $\text{MAXINT}(I)$. If your algorithm works in time which is polynomial in $\text{MAXINT}(I)$ and the size of the instance, then your algorithm is said to be a pseudo-polynomial time algorithm for the problem.

What this means is the following: It means that if for all instances your largest integer happens to be bounded by a constant, or even if it is bounded by a polynomial in the size of the instance, then this pseudo-polynomial time algorithm will really be a polynomial-time algorithm. So if you know that the numbers that appear in the instances are all smaller than 1000, or that they are never bigger than $10n^2$, then you know positively that your pseudo-polynomial time algorithm is a polynomial-time algorithm.

We are now going to look at a pseudo-polynomial algorithm for a problem called 0-1 KNAPSACK.

IN210 – lecture 10

**Example: 0-1 KNAPSACK**

In 0-1 KNAPSACK we are given integers w_1, w_2, \dots, w_n and K , and we must decide whether there is a subset S of $\{1, 2, \dots, n\}$ such that $\sum_{j \in S} w_j = K$. In other words: Can we put a subset of the integers into our knapsack such that the knapsack sums up to exactly K , under the restriction that we include any w_i at most one time in the knapsack.

Note: This decision version of 0-1 KNAPSACK is essentially SUBSET SUM.

0-1 KNAPSACK can be solved by dynamic programming. **Idea:** Going through all the w_i one by one, maintain an (ordered) set M of all sums ($\leq K$) which can be computed by using some subset of the integers seen so far.

Autumn 1999

5 of 22

0-1 KNAPSACK is a restricted variant of INTEGER LINEAR PROGRAMMING, and there is a story that goes with the problem: You are going backpacking and you want to take a bunch of things with you, but you know that you definitely don't want to carry more than 20 kilos. So K is 20 kilos. Every item i that you want to carry in the backpack has a weight w_i , which is an integer. So you want to pack as much as possible, but not exceeding K . In the decision version of 0-1 KNAPSACK the question is: Given this number K and a bunch of weights w_i , can we choose the items so that their total weight ends up to exactly K , to the limit, and such that we carry at most one piece of each item?

The decision version of 0-1 KNAPSACK is in fact identical to the SUBSET SUM problem which we have seen already, so you know immediately that it is \mathcal{NP} -complete. But 0-1 KNAPSACK can be solved by a pseudo-polynomial algorithm. We will call that algorithm for DP because it is a version of the dynamic programming technique. You will recognize the dynamic programming principle here. We are building up a solution from partial solutions in a certain way.

The basic idea behind the DP is this: We go through all the w_i one by one, updating an ordered set M of all sums less than or equal to K , that can be computed by using some subset of the integers seen so far. This will be done by checking whether we get some new sums by adding w_j to any of the sums already in M . If K is a member of M when all the weights have been processed, then we have a positive instance, otherwise we have negative instance.

IN210 – lecture 10

**Algorithm DP**

1. Let $M_0 := \{0\}$.
2. For $j = 1, 2, \dots, n$ do:
 - Let $M_j := M_{j-1}$.
 - For each element $u \in M_{j-1}$:
 - Add $v = w_j + u$ to M_j if $v \leq K$ and v is not already in M_j .
3. Answer 'Yes' if $K \in M_n$, 'No' otherwise.

Example: Consider the instance with w_i 's 11, 18, 24, 42, 15, 7 and $K = 56$. We get the following M_i -sets:

- $M_0 : \{0\}$
- $M_1 : \{0, 11\}$ ($0 + 11 = 11$)
- $M_2 : \{0, 11, 18, 29\}$ ($0 + 18 = 18, 11 + 18 = 29$)
- $M_3 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
- $M_4 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
- $M_5 : \{0, 11, 15, 18, 24, 26, 29, 33, 35, 39, 42, 44, 50, 53\}$
- $M_6 : \{0, 7, 11, 15, 18, 22, 24, 25, 26, 29, 31, 33, 35, 36, 39, 40, 42, 44, 46, 49, 50, 51, 53\}$

Theorem 1 *DP is a pseudo-polynomial algorithm. The running time of DP is $\mathcal{O}(nK \log K)$.*

Proof: MAXINT(I) = $K \dots$

Autumn 1999

6 of 22

Here we give the DP algorithm in pseudo-code: The initialization step is to put 0 in M . Then we process the weights one by one. For each weight w_j ($1 \leq j \leq n$) we do the following: We check whether adding w_j to any of the elements in M would give us a sum which is less than or equal to K and not already in M . If so, then we add this new sum to M . We do this check for each element in M before moving on to w_{j+1} . The only pitfall is that we might add the weight w_j two times if we first add $w_j + m$ to M and then $w_j + (w_j + m)$ to M again. The simplest solution is to compute a new version of M in each iteration, so that we only compare w_j with the sums in the old M set. We indicate this in the algorithm by using subscripts on M .

If K is a member of the final M set, then we have a positive DP instance, otherwise we have negative instance.

On the foil I have shown how the algorithm runs on an example instance. The set of computable sums contains in the beginning only '0'. Then we process the first element which is 11. Naturally we add 11 to M . The next element is 18. Then we have to add both 18 ($0+18$) and 29 ($11+18$) to M . So after two iterations $M = \{0, 11, 18, 29\}$ because that is the possible weights of the knapsack after placing zero, one or both items into the knapsack. The algorithm continues as on the foil.

So this is the dynamic programming algorithm for 0-1 KNAPSACK. What is its complexity? It is a very simple algorithm. It has a loop where j ranges from 1 to n . And then each iteration of the loop takes the work of the order $K \log K$, because we can have up to K numbers (sums) in M and we need $\log K$ time for

searching/insertion into M . So the complexity of the algorithm is $\mathcal{O}(nK \log K)$. Since $\text{MAXINT}(I) = K$ (we can ignore input elements which are bigger than K since they obviously doesn't fit into the knapsack), DP is per definition a pseudo-polynomial algorithm.

How big is this running time? It looks like it is linear time, and it is indeed linear in n – the number of items. However K is a part of the input, and K as a number can in fact be exponential large compared to its own length (number of digits), because it is written in binary. So the running time as a function of the input length is $\mathcal{O}(n \cdot 2^{|K|} \log 2^{|K|}) = \mathcal{O}(nK \cdot 2^{|K|})$.

So to begin with this is exponential time. But if there is a natural limit on K , if we know that our big K is somehow bounded by something, then this is all bounded. If K happens to be constant, this is just linear time. If K happens to be polynomial in n , then $nK \log K$ is a polynomial in n . The running time as a function of the input length is also a polynomial because $|K|$ is of the order $\log n$ when K is a polynomial in n .

So naturally you would look at your problem and see if there is some limitation on K , and if there is a limitation on K , you run to the first store and buy a pseudo-polynomial time algorithm, because that's your good guy – it's polynomial.

IN210 – lecture 10

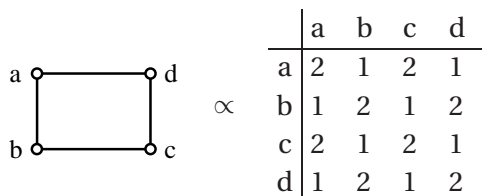


Strong \mathcal{NP} -completeness

Def. 2 A problem which has no pseudo-polynomial algorithm unless $\mathcal{P} = \mathcal{NP}$ is said to be **\mathcal{NP} -complete in the strong sense** or **strongly \mathcal{NP} -complete**.

Theorem 2 TSP is strongly \mathcal{NP} -complete.

Proof: In the standard reduction $\text{HAM} \propto \text{TSP}$ the only integers are 1, 2 and n , so $\text{MAXINT}(I) = n$. Hence a pseudo-polynomial algorithm for TSP would solve HAMILTONICITY in polynomial time (via the standard reduction).



$$K = n (= 4)$$

Autumn 1999

7 of 22

There are number problems which cannot be solved by pseudo-polynomial time algorithms unless $\mathcal{P} = \mathcal{NP}$. We call such problems *strongly \mathcal{NP} -complete*, or *\mathcal{NP} -complete/ \mathcal{NP} -hard in the strong sense*.

To prove for those problems that they are \mathcal{NP} -hard in the strong sense, is usually rather easy. I demonstrate with a familiar example: TSP is strongly \mathcal{NP} -complete. The proof is simply to look at the standard reduction from HAMILTONICITY, where we label an TSP edge with '1' if there is an edge in the corresponding HAMILTONICITY instance, and '2' otherwise, and we ask whether there is a TSP tour of length n or less.

So the standard reduction from HAMILTONICITY creates TSP-instances where the only numbers are 1's, 2's and n . This implies that $\text{MAXINT}(I) = n$, which is obviously a polynomial in n . This means that if TSP can be solved by a pseudo-polynomial algorithm, then this straight-forward reduction plus that algorithm give us a polynomial-time algorithm for HAMILTONICITY. And since HAMILTONICITY is an \mathcal{NP} -complete problem, that means that $\mathcal{P} = \mathcal{NP}$. Remember that if we can solve an \mathcal{NP} -complete problem in polynomial time, then all \mathcal{NP} problems can be solved in polynomial time, by virtue of the definition of \mathcal{NP} -completeness.

So proving that a problem P_2 is \mathcal{NP} -complete in the strong sense amounts to showing that a known \mathcal{NP} -complete problem P_1 can be reduced to P_2 in such a way that all numbers in the P_2 instance always are of limited size, meaning at most polynomial in the length of the original P_1 instance.

Notice the 'unless $\mathcal{P} = \mathcal{NP}$ ' part in the definition of strong \mathcal{NP} -completeness.

This $P=\mathcal{NP}$ is like a big cloud hanging over our heads all the time. Why is it there? Because we haven't been able to prove that those hard problems that we are talking about, really cannot be solved in polynomial time. Then of course we cannot say that they cannot be solved in pseudo-polynomial-time, because one guy might find a polynomial-time algorithm for SAT tomorrow, and that algorithm will solve all problems in \mathcal{NP} in *real* polynomial time.

The best statements we can make are of the form: "Unless \mathcal{P} equals \mathcal{NP} , this is impossible", and those are the kind of statements that we are showing here. Since we know that it is very unlikely that \mathcal{P} will turn out to be equal to \mathcal{NP} – because that would mean that all of those difficult problems can be solved in polynomial time magically somehow – those statements are pretty strong results, and for practical purposes they are strong enough.

IN210 – lecture 10



Alternative approaches to algorithm design and analysis

- **Problem:** Exhaustive search gives typically $\mathcal{O}(n!) \approx \mathcal{O}(n^n)$ -algorithms for \mathcal{NP} -complete problems.
- So we need to get around the **worst case / best solution** paradigm:
 - worst-case \rightarrow average-case analysis
 - best solution \rightarrow approximation
 - best solution \rightarrow randomized algorithms

Autumn 1999

8 of 22

We have now seen one way to try to crack down \mathcal{NP} -completeness in its original form. We have a whole other set of alternative approaches, which are based on looking at the foundation of this whole theory that we have made here and saying: "Wait a minute, this really doesn't do us much good in practice!"

The weak point that is easy to attack in the theory we have constructed, is the fact that it is based on the worst-case & best-solution approach. What is this worst-case & best-solution approach? We know that there are exponential many possible instances. We say that there is a "combinatorial explosion". You look at a graph and it is an innocent kind of simple object. But if you are saying: Let's see how many ways there are to arrange the vertices, how many possible Hamiltonian paths there are in this graph. Then you see that with n vertices there are $n!$ possible orderings of vertices.

Basically $n!$ is something like n^n , and we know that if we are looking at some number like 2^{200} , this number is bigger than the number of molecules in the whole universe – not only planet earth, not only our solar system, but the whole big universe.

So 2^{200} is an astronomically huge number, and 100^{100} also. But one hundred is a small instance – it is not a big graph. We have graphs with thousands of vertices that naturally arise as instances of problems. So the point is that we are looking for the very best solution among these astronomically many possible solutions! This is kind of psychologically wrong – it is greedy. Why do you want the very best solution out of 100^{100} possible solutions? Let's just find a good solution and live with it. This is basically how we do it in normal life.

And we are also looking at how the algorithm performs in the worst case – how it performs for the most annoying input. The number of graphs that can arise on n vertices is also an astronomically large number. So the fact that there are some very bad cases doesn't tell us really much about the difficulty of the problem.

We want to relax the requirement that our whole approach should be based on the worst-case, which may never occur in practice, and say: "Let's look at the average, let's look at the typical cases. That's more natural!" That gives rise to average-case analysis. It is a whole approach to algorithm design and to analyzing algorithms and problems.

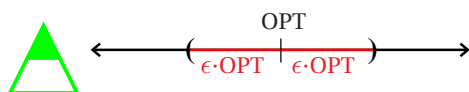
We also want to look at the solution and say: "We don't necessarily want the very best solution, let's find something that we can live by. That's good enough!" This leads to two kinds of approaches. One is approximation, and another one is probabilistic or randomized algorithms.

We are going to look at all three of those alternative approaches. We begin with approximation.

IN210 – lecture 10



Approximation

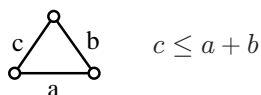


Def. 3 Let L be an optimization problem. We say that algorithm M is a **polynomial-time ϵ -approximation algorithm** for L if M runs in polynomial time and there is a constant $\epsilon \geq 0$ such that M is guaranteed to produce, for all instances of L , a solution whose cost is within an ϵ -neighborhood from the optimum.

Note 1: Formally this means that the **relative error** $\frac{|L_M(n) - \text{OPT}|}{\text{OPT}}$ must be less than or equal to the constant ϵ .

Note 2: We are still looking at the worst case, but we don't require the very best solution any more.

Example: TSP with triangle inequality has a polynomial-time approximation algorithm.



Autumn 1999

9 of 22

Obviously approximation doesn't fit very well with decision problems, because 'Yes' and 'No' cannot easily be approximated (although *fuzzy logic* is trying to do exactly that). So decision problems are not something that we look at in approximation, instead we look at optimization problems.

In optimization the algorithm is supposed not only to say 'Yes' or 'No', but actually produce a solution, and solutions they have costs which reflect their value. There is always the best possible solution, which is either the largest or the smallest possible cost. So we are talking about maximization or minimization problems.

So optimum, OPT for short, is either the biggest possible value or the smallest possible value that can arise as a result. In either case we can look at an interval around this optimum, which is of $\epsilon \cdot \text{OPT}$ width. The figure on the foil illustrates this. You can think of this epsilon as being k percent, say 10 percent. So ϵ is something like a small constant, i.e. 10 percent would give $\epsilon = 0.1$.

The question that we are asking here is: Can we find an algorithm that doesn't necessarily give us the optimal solution, but some "reasonable good" solution? There maybe just one out of astronomically many solutions that actually has that optimality, but maybe there are lots of good solutions. So can the algorithm find a solution that is within 10 percent from the optimum? That is a very natural question.

We now define what a *polynomial-time ϵ -approximation algorithm* is: Given some positive constant epsilon, given some bound, if we have an algorithm that can certainly find a solution that is always within the epsilon bound from the

optimum in polynomial time – ‘always’ meaning for every instance – then we have a polynomial-time approximation algorithm for the problem.

Another way of saying the same is that the **relative error** $\frac{|t_M(n) - \text{OPT}|}{\text{OPT}}$ must be less than or equal to the constant ϵ .

This definition here is different from the definition in your G&J textbook. In G&J it is not required that epsilon is a fixed constant, a well-defined constant. It is enough the relative error, called R_A , is some epsilon. That is not a standard definition, it is not very good. So epsilon needs to be a constant in this class, constant meaning fixed for all problem instances.

Notice that we are still talking about the worst case, because the algorithm has to find an ϵ -approximation to the optimum for all possible instances. But we are not requiring the best possible solution anymore. So worst case is still there, but we don’t require the best possible solution. Instead we say: Give us an approximation to the optimum.

And if we can live with that, then some problems can be solved efficiently. For example the TSP with the triangle inequality has a polynomial-time approximation algorithm which I am going to show now.

Let me first explain what the triangle inequality is. Triangle inequality is the kind of inequality that points in plane satisfy, and triangles also. If you have a triangle with the three edges having costs a , b and c then you know that if this is a nice triangle in a nice Cartesian plane, then for any pair of edges the sum of length of those two edges must be greater than or equal to the length of the third length. This is the triangle inequality.

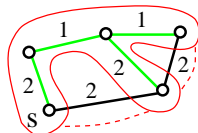
The sum of two edges must be larger than the third edge because this whole thing would otherwise not be a triangle. But if you are talking about the cities in a normal map, then there can be a kind of bumpy mountain road going left and right. But who cares about those guys? If you are looking at nice, honest roads that are not too dirty and too dusty – so that your traveling sales person can use them without getting his car messed up and his suit torn into pieces – then you can assume rather safely that the triangle equality holds in any reasonable map. And if that is the case, then it turns out that TSP has a natural polynomial-time approximation algorithm, which I am going to describe now.

IN210 – lecture 10

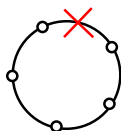
**Algorithm TSP- \triangle :**

Phase I: Find a minimum spanning tree.

Phase II: Use the tree to create a tour.



The cost of the produced solution can not be more than $2 \cdot \text{OPT}$, otherwise the OPT tour (minus one edge) would be a more minimal spanning tree itself. Hence $\epsilon = 1$.



Opt. tour

So assume that this graph on the foil is an instance of an optimization-TSP. We are trying to find the shortest tour. There are some edges missing here, but you can think of these edges as being infinite or being large numbers. So normally you have some kind of edges between every pair of cities in a distance matrix, but since those edges can be infinite large (but they must be finite numbers), there are no loss of generality here.

I will explain the approximation algorithm using the example instance on the foil. The algorithm has two phases. In phase 1 we find the MINIMUM SPANNING TREE in this network, a network being a graph with edges that has distances or weights. I am showing a minimum spanning tree here in green, and you know from our last class that there is an $n \log n$ algorithm for finding the minimal spanning tree.

So the algorithm for solving MST is efficient, and then in phase 2 we use this minimum spanning tree to construct an optimal tour. That is the interesting part, and this is how it works: Think of this minimum spanning tree as being a tour. It is not a tour really, but think of it as being a tour. And it would be a kind of a tour where we start from vertex s , and then we go along the edges, visiting this tree in a depth first order. So we do just simply a depth-first enumeration of the tree. We go around the tree on one side, and then we go around the same edges on the other side and come back to s . I have marked this tour in red.

The problem with the tour is that it is not really a Hamiltonian tour, because we are going along each edge in the spanning tree two times. But the good thing about it is that we know its cost. The cost of this red pseudo-tour that I

have made, is 2 times the cost of the minimum spanning tree, and I claim that that cost cannot be more than 2 times the optimal solution.

To prove this claim I have to show that the cost of the optimal solution cannot possibly be smaller than the cost of the minimum spanning tree. That is easy to show because an optimal solution to TSP is a cycle in the graph, and if this cycle would cost less than the minimum spanning tree, we could then obtain a more minimal spanning tree by just leaving out one of the edges in the cycle. So the optimum solution must cost at least as much as the minimum spanning tree.

This means that if we have a solution that cost no more than this red pseudo-tour, then this solution will cost not more than 2 times the cost of the minimum spanning tree and also not more than then 2 times the optimum. So it is an approximation with $\epsilon = 1$.

Now we construct such a solution from our red pseudo-tour. The solution follows the red line for a while until the red line starts doing weird things like visiting a vertex it has already been at. So when your red pseudo-tour revisits a vertex and then goes to a new vertex, our new solution would go directly to that new vertex. And by virtue of the triangle inequality this new red-dotted path fragment is not any longer than the old red path fragment.

If our red path does the same weird thing again – going back to a vertex that has already been visited – then we do the same trick once more: We go directly to the next vertex on the red line. And so on and so force. Because of the triangle inequality, we don't make the solution worse by taking these shortcuts.

So these two phases put together give us a TSP-solution which is within $\epsilon = 1$ from the optimum. And it works in polynomial time because phase 1 works in time $\mathcal{O}(n \log n)$ while phase 2 works in linear time. So the whole thing is an extremely efficient algorithm for approximating TSP on graphs where the triangle inequality holds.

IN210 – lecture 10



Theorem 3 TSP has no polynomial-time ϵ -approximation algorithm for any ϵ unless $\mathcal{P}=\mathcal{NP}$.

Proof:

Idea: Given ϵ , make a reduction from HAMILTONICITY which has only **one** solution within the ϵ -neighborhood from OPT, namely the optimal solution itself.

	\propto		a	b	c	d
			$2+\epsilon n$	1	$2+\epsilon n$	1
		a	1	$2+\epsilon n$	1	$2+\epsilon n$
		b	$2+\epsilon n$	1	$2+\epsilon n$	1
		c	1	$2+\epsilon n$	1	$2+\epsilon n$
		d	$2+\epsilon n$	1	$2+\epsilon n$	1

$$K = n(= 4)$$

The **error** resulting from picking a non-edge is: $\text{Approx. solutin} - \text{OPT} = (n - 1 + 2 + \epsilon n) - n = (1 + \epsilon)n > \epsilon n$

Hence a polynomial-time ϵ -approximation algorithm for TSP combined with the above reduction would solve HAMILTONICITY in polynomial time.

Now we will look at the other side of the coin. We are going to show that unless $\mathcal{P}=\mathcal{NP}$ the original TSP, TSP without triangle inequality, has no polynomial-time approximation algorithm for any epsilon. In other words, I am going to show that if there is a polynomial-time approximation algorithm for TSP for some epsilon, then immediately we can solve all the \mathcal{NP} -complete problems in polynomial-time, meaning that $\mathcal{P}=\mathcal{NP}$.

The idea is again that we use the standard reduction from HAMILTONICITY, but when we do the reduction we choose the numbers in the TSP-instance such that there will be one and only one solution in the epsilon interval around the optimum. That one and only one is of course the optimal solution, and finding an approximate solution within epsilon, would mean finding the exact solution. It is as simple as that.

In the standard reduction we begin with an instance of HAMILTONICITY, like on the foil, and we are creating a TSP distance matrix. Because we are not obliged to fulfil the triangle inequality constraint, we can choose any numbers we like. For edges we choose 1's as before, but for non-edges we choose some huge number. So that if ever we end up choosing one non-edge in the tour, automatically the overall solution is going to fall out of this epsilon interval around the optimum – the optimum being n if the graph is Hamiltonian.

It turns out to be enough to translate non-edges into $2 + \epsilon n$. So that if we take $n - 1$ edges and 1 non-edge, the tour will cost $(n - 1) + (2 + \epsilon n)$. The error, which means how much we deviate from the optimal cost n , is then $(1 + \epsilon)n$, which is always bigger than ϵn .

So even the smallest possible error, which happens if we choose one non-edge in our Hamiltonian graph, gives an approximate solution that is not within an epsilon-interval from the optimum. This means that any algorithm that approximate TSP within epsilon, actually finds the optimal TSP-tour in this reduced instance, and it solves the HAMILTONICITY problem by virtue of this reduction. And by solving HAMILTONICITY it solves all other \mathcal{NP} -problems because all problems in \mathcal{NP} are reducible in polynomial time to HAMILTONICITY by virtue of HAMILTONICITY being \mathcal{NP} -complete.

IN210 – lecture 10

**Example: VERTEX COVER**

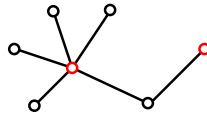
- **Heuristics** are a common way of dealing with intractable (optimization) problems in practice.
- Heuristics differ from algorithms in that they have no performance guarantees, i.e. they don't always find the (best) solution.

A greedy heuristic for VERTEX COVER-opt.:

Heuristic VC-H1:

Repeat until all edges are covered:

1. Cover highest-degree vertex v ;
2. Remove v (with edges) from graph;



Theorem 4 *The heuristic VC-H1 is not an ϵ -approximation algorithm for VERTEX COVER-opt. for any fixed ϵ .*

Autumn 1999

12 of 22

We will now go through another example, namely VERTEX COVER. The optimization version of VC asks for the smallest set of vertices covering all the edges. We will first see an approximation algorithm based on a *heuristics*.

Heuristics are a common way of dealing with intractable problems in practice, and especially optimization problems. A heuristic is often based on a rule of thumb and is widely used in the field of artificial intelligence (AI) to model human behavior and reasoning. Heuristics differ from algorithms in that they have no performance guarantees. A heuristic for an optimization problem does not always find the best solution.

On the foil there is a greedy heuristic for VC, which we will call VC-H1: "Cover the highest-degree vertex v and remove v and its edges from the graph. Repeat this until all edges are covered."

What is the idea? Remember that we are trying to put guards on intersections so that all streets are surveyed with as few guards as possible. Look at the example on the foil: Wouldn't it be the most natural thing in the world to put a guard on this red intersection because there are so many streets coming out of here? And once you have that then already all these streets are covered, and then you need only one more here on the other red vertex. So this is obviously the optimum.

So you would think that this kind of approach should work very well: You pick the highest-degree vertex, put a guard on that vertex and then remove the vertex from the graph together with the incident edges, and then look at the rest. In our example the rest is already only one edge, so you put one guard here

and you have a nice solution.

The question is: How well does this greedy heuristic really work for solving the optimization version of VERTEX COVER – for finding the minimum number of guards needed? And you would expect that it works pretty well.

If you want to answer this question, then there are two approaches to follow: One is trying to prove that the heuristics work. Another is trying to prove that it doesn't work. We have seen lots of "positive" proofs in this class already, but how do we prove that something doesn't hold? It is in fact much easier than proving that something holds in all possible cases. To disprove something, it is enough to show one 'counterexample': You present a kind of a bad instance of a problem and show that your heuristic or algorithm does poorly on that bad instance.

I am going to show that this completely natural heuristic can do very poorly, by cooking up a bad instance of the problem – in jargon it is called an 'counterexample'. This bad instance will in fact be a pattern which proves that VC-H1 is not an ϵ -approximation algorithm for optimization-VC for *any* fixed epsilon. This means that no matter how big you choose the constant ϵ , I will be able to construct an instance, based on this pattern, such that VC-H1 fails to find a solution within epsilon from the optimum.

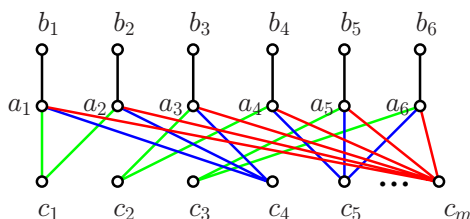
IN210 – lecture 10

**Proof:**

Show a **counterexample**, i.e. cook up an instance where the heuristic performs badly.

Counterexample:

- A graph with nodes a_1, \dots, a_n and b_1, \dots, b_n .
- Node b_i is only connected to node a_i .
- A bunch of c -nodes connected to a -nodes in the following way:
 - Node c_1 is connected to a_1 and a_2 . Node c_2 is connected to a_3 and a_4 , etc.
 - Node $c_{n/2+1}$ is connected to a_1, a_2 and a_3 .
 - Node $c_{n/2+2}$ is connected to a_4, a_5 and a_6 , etc.
 - ...
 - Node c_{m-1} is connected to a_1, a_2, \dots, a_{n-1} .
 - Node c_m is connected to all a -nodes.



Autumn 1999

13 of 22

So I am cooking up a bad instance of VC where the heuristic performs badly, and this is how I do it: I have a graph with a bunch of a_i vertices and a bunch of b_i vertices – n of each. a_1 is connected to b_1 , a_2 to b_2 and so on, as shown on the foil. I have also a huge number of c_i vertices which are connected to the a -vertices in a way I will describe soon.

We can see immediately that the optimal solution is to cover all the a -vertices, because they are connected to all the b - and c -vertices. So the optimal solution has cost n .

I am going to fool the heuristic by connecting the c -vertices to the a -vertices in such a way that the heuristic will cover all the c -vertices before the a -vertices, because in each step it will be one of the c -vertices which has the highest degree. So the algorithm will do lot of unnecessary work before covering the n a -vertices (or b -vertices), which have to be covered in the end anyway.

This is how it works: I connect the first c -vertex with the first two a -vertices, then the second c -vertex with the second two a -vertices, then the third with c -vertex the next two, and so on until all a -vertices have been connected to a c -vertex.

I continue by connecting vertex $c_{n/2+1}$ with the three first a -vertices, $c_{n/2+2}$ with the next three a -vertices, and so on until all a -vertices have been connected once more. Then I connected the next c -vertex with the first four a -vertices, and so on – the pattern should be visible now. The last c -vertex, c_m , will be connected to all n a -vertices.

(This is a blank page)

IN210 – lecture 10



- The optimal solution OPT requires n guards (on all a -nodes).
- VC-H1 first covers all the c -nodes (starting with c_m) before covering the a -nodes.
- The number of c -nodes are of order $n \log n$.
- Relative error for VC-H1 on this instance:

$$\frac{|VC-H1| - |OPT|}{|OPT|} = \frac{(n \log n + n) - n}{n} = \frac{n \log n}{n} = \log n \neq \epsilon$$

- The relative error **grows as a function of n** .

Heuristic VC-H2:

Repeat until all edges are covered:

1. Pick an edge e ;
2. Cover and remove both endpoints of e .

- Since at least one endpoint of every edge must be covered, $|VC-H2| \leq 2 \cdot |OPT|$.
- So VC-H2 is a polynomial-time ϵ -approximation algorithm for VC with $\epsilon = 1$.
- Surprisingly, this “stupid-looking” algorithm is the best (worst case) approximation algorithm known for VERTEX COVER-opt.

Autumn 1999

14 of 22

So all c -vertices will have high degrees, and c_m the highest of them all. So VC-H1 is going to pick c_m first and remove it from the graph. It is quite easy to see that c_{m-1} is now the highest-degree vertex, because that is how we constructed the edges between the c - and a -vertices. And then comes c_{m-2} , etc. all the way down to c_1 . So all the c -vertices will be chosen before any a -vertex.

How many c -vertices do I have? It turns out that I have approximately $n \log n$ of them when you do the counting properly. So instead of picking just n a -vertices, I pick $n \log n$ c -vertices and then n a -vertices. So my heuristic is going to end up with a solution which is of cost $\mathcal{O}(n \log n)$ instead of n which is the optimum. And the ratio between these two solutions is ‘actual cost minus optimal cost divided by optimal cost’. This is called the *relative error*.

We see that on this instance the relative error is $\mathcal{O}(\log n)$. And this is not a constant. It is actually something that grows with n . – it is a function of n . So this heuristic does in fact very poorly in the worst case.

I will now show you another heuristic which have a better worst-case performance – and mind you, we are talking about worst-case here. Heuristic VC-H2 says: “Pick an edge. Cover and remove both of its endpoints! Repeat until all edges have been covered.”

This sounds completely absurd. Why should we place guards on both ends of the edge when – as far as that edge is concerned – you need a guard only on one side. That is surely not going to be an optimal solution. The point is that this gives us a simple performance measure: The edge e we have picked, the fact that e is an edge in the graph implies that any solution to the VERTEX COVER

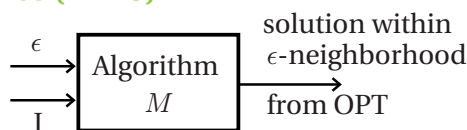
problem must have one of e 's endpoints covered. Here in VC-H2 we are covering both endpoints. So we are introducing only a factor 2, a constant factor.

The solution you get in this way is only at most by a constant factor 2 worse than the best possible solution. In other words: VC-H2 is an approximation algorithm to VC with $\epsilon = 1$.

This absurd heuristic is actually better in the worst case than the completely natural VC-H1. And VC-H2 is actually the best worst-case polynomial-time approximation algorithm known for VC. This should give you an indication that although worst-case performance is easy to analyze, it doesn't always model real-world complexity very well.

IN210 – lecture 10

Polynomial-time approximation schemes (PTAS)



Running time of M is $\mathcal{O}(P_\epsilon(|I|))$ where $P_\epsilon(n)$ is a polynomial in n and also a function of ϵ .

Def. 4 M is a **polynomial-time approximation scheme (PTAS)** for optimization problem L if given an instance I of L and value $\epsilon > 0$ as input

1. M produces a solution whose cost is within an ϵ -neighborhood from the optimum (OPT) and
2. M runs in time which is bounded by a polynomial (depending on ϵ) in $|I|$.

M is a **fully polynomial-time approximation scheme (FPTAS)** if it runs in time bounded by a polynomial in $|I|$ and $1/\epsilon$.

Example: 0-1 KNAPSACK-optimization has a FPTAS.

Autumn 1999

15 of 22

So some problems have polynomial-time approximation algorithms for certain epsilons, other don't unless $\mathcal{P}=\mathcal{NP}$. The question now is: How big is this epsilon? And for some problems there are natural limitations to how much you can approximate the problem, but other problems can be approximated arbitrarily precisely. 'Arbitrarily precisely' meaning the following: You give me an epsilon and I give a solution that is within epsilon from the optimum, no matter how small your epsilon is.

You say: "I want a solution that is within 1 percent from the optimum." And I say: "OK, I give you that solution." So if I am an algorithm capable of producing such solutions efficiently, meaning in polynomial time in the size of the instance, then I call myself a *polynomial-time approximation scheme* (PTAS).

Let me tell you just before I dive into technicalities that this is the best way you can approximate problems. These PTAS they are like really very sleek solutions to optimization problems and to this whole issue of intractability.

Here is the idea behind a PTAS: Think of this box on the foil being an algorithm, and then the algorithm will take two pieces of information as input: One is the original instance I , the other is this epsilon. You, the customer, are saying: "Give me a solution which is within epsilon, say, 10 percent from the optimum."

And the algorithm is then going to produce that kind of solution that is within epsilon from the optimum, in time which is polynomial in the size of the instance. But obviously that polynomial is itself depending on epsilon (i.e. has epsilon in it). So the smaller the epsilon – the more precision you want – the more time this algorithm is going to take. But if you say: "For the rest of my

life I want to give you solutions that is within 10 percent from the optimum." Then you fix epsilon to 10 percent and the whole thing is going to be polynomial. It is going to give you approximate solutions within epsilon in time which is polynomial in the size of the input.

So these schemes they are really like families of algorithms – one algorithm for each epsilon. Here is a formal definition of a PTAS: M is a polynomial-time approximation scheme (PTAS) for optimization problem L if given an instance I of L and a positive value epsilon as input: One, M produces a solution whose cost is within an epsilon-neighborhood from the optimum. Two: M runs in time which is bounded by a polynomial in $|I|$. That polynomial is allowed to be depending on epsilon.

If in addition to all this M happens to run in time which is bounded by a polynomial in $|I|$ and $1/\epsilon$, then we say that M is a *fully polynomial-time approximation scheme*, FPTAS. So we are not only saying that the algorithm runs in time which is polynomial in I , but it is also a polynomial in 1 over epsilon. It is a very strict condition on how your polynomial may depend on epsilon.

We are now going to see, as an example, a FPTAS for the optimization version of 0-1 KNAPSACK.

IN210 – lecture 10

**0-1 KNAPSACK-optimization**

Instance: $2n + 1$ integers: Weights w_1, \dots, w_n and costs c_1, \dots, c_n and maximum weight K .

Question: Maximize the total cost

$$\text{subject to } \sum_{j=1}^n c_j x_j$$

$$\sum_{j=1}^n w_j x_j \leq K \text{ and } x_j = 0, 1$$

Image: We want to maximize the total value of the items we put into our knapsack, but the knapsack cannot have total weight more than K and we are only allowed to bring one copy of each item.

Note: Without loss of generality, we shall assume that all individual weights w_j are $\leq K$.

0-1 KNAPSACK-opt. can be solved in pseudo-polynomial time by dynamic programming. **Idea:** Going through all the items one by one, maintain an (ordered) set M of pairs (S, C) where S is a subset of the items (represented by their indexes) seen so far, such that S is the “lightest” subset having total cost equal C .

Autumn 1999

16 of 22

We have just seen the decision version of the 0-1 KNAPSACK. An instance to the optimization version contains n weights w_i and a maximum knapsack capacity K . This is just like in the decision version. But in addition to that each item is now given a value or cost c_j . So for example a compass is of huge value and is very light, while a television set is heavy and not of much value up in the mountains.

We want to maximize the total value of the items we put into the knapsack, but the knapsack cannot have total weight more than K and we are only allowed to bring one copy of each item. I have written the same question in a more linear-programming kind of style on the foil.

With loss of generality, we shall assume that alle individual weights are less than or equal to K . That is a very reasonable assumption – we are not interested in items which cannot even fit into an empty knapsack.

It turns out that the optimization version of 0-1 KNAPSACK can be solved in pseudo-polynomial time by dynamic programming, in almost exactly the same as the decision version could. The algorithm is just a variant of what we have seen before.

The idea is to go through all items one by one, maintaining an (ordered) set M of pairs (S, C) where S is the lightest set among the subsets of items seen so far having total cost C . So we again build up partial solutions, extending and improving them as we process more and more items.

(This is a blank page)

IN210 – lecture 10

**Algorithm DP-OPT**

1. Let $M_0 := \{(\emptyset, 0)\}$.
2. For $j = 1, 2, \dots, n$ do steps (a)–(c):
 - (a) Let $M_j := M_{j-1}$.
 - (b) For each elem. (S, C) of M_{j-1} :
 - If $\sum_{i \in S} w_i + w_j \leq K$, then add $(S \cup \{j\}, C + c_j)$ to M_j .
 - (c) Examine M_j for pairs of elements (S, C) and (S', C) with the same 2nd component. For each such pair, delete (S', C) if $\sum_{i \in S'} w_i \geq \sum_{i \in S} w_i$ and delete (S, C) otherwise.
3. The optimal solution is S where (S, C) is the element of M_n having the largest second component.

- The running time of DP-OPT is $\mathcal{O}(n^2 C_m \log(n C_m W_m))$ where C_m and W_m are the largest cost and weight, respectively.

Autumn 1999

17 of 22

Here I have given the algorithm in pseudo-code. We are looping through all items and for each item i we do the following:

First we make a new copy of the set M . As explained earlier this is to prevent that we put the same item into the knapsack twice.

Then we go through each element (S, C) in M_{i-1} and check whether we get a new partial solution by adding element i to S , meaning that the total weight of the elements in S plus element i is less than K . If so, then we add the new element, consisting of S plus i with the corresponding total cost, to M_i .

In order to avoid having an exponential large set M , we have to do some cleaning up. The key observation is that if we have two elements with the same second component – meaning with the same cost – in the set M then we can delete the one which has the bigger weight. Because we don't want to exceed total weight K , and if some set with the same total cost already has a bigger weight after considered the same elements, then putting in new elements will not help.

This cleaning-up we do as the third step in the loop, and we can do it efficiently if we keep the set M ordered on the elements' second component, namely their total weight C .

After looping through all n items, we can conclude that the optimal solution is S where (S, C) is the element of M_n having the largest second component.

So in effect what we are doing is that we have a big sack M , and we keep putting in all partial solutions which have weight less than or equal than our limit K . And when we find two solutions which have the same cost, we simply

throw away the one that has the bigger weight, because that one is uninteresting for us. In the end we find the solution inside the bag having the biggest possible total cost, and we say that is our best solution.

How much time does this algorithm take? We have an outer loop which will be repeated for all n items – that's a factor n . We are keeping at most $\sum c_j \leq n \cdot C_m$ elements in M by virtue of this throwing away business, where C_m is the biggest cost among all items. For each of those $n \cdot C_m$ elements we compute their sum of weights in 2b and 2c, but we don't need to recompute it for every iteration of the loop. We can store the result somewhere. Computing a weight sum takes $\mathcal{O}(n \log W_m)$ time, where W_m is the largest weight, because each element in M consists of at most n items. In 2b and 2c we need to be able to insert, delete and search in the set M , which can be done in time $\mathcal{O}(\log(nC_m))$.

A little bit of calculation will show that our solution is $\mathcal{O}(n^2 C_m \log(nC_m W_m))$.

IN210 – lecture 10



Example: Consider the following instance of 0-1 KNAPSACK-opt.

j	1	2	3	4
w_j	1	1	3	2
c_j	6	11	17	3

$K = 5$

Running the DP-OPT algorithm results in the following sets:

$$\begin{aligned}
 M_0 &= \{(\emptyset, 0)\} \\
 M_1 &= \{(\emptyset, 0), (\{1\}, 6)\} \\
 M_2 &= \{(\emptyset, 0), (\{1\}, 6), (\{2\}, 11), (\{1, 2\}, 17)\} \\
 M_3 &= \{(\emptyset, 0), (\{1\}, 6), (\{2\}, 11), (\{1, 2\}, 17), \\
 &\quad (\{1, 3\}, 23), (\{2, 3\}, 29), (\{1, 2, 3\}, 34)\} \\
 M_4 &= \{(\emptyset, 0), (\{4\}, 3), (\{1\}, 6), (\{1, 4\}, 9), \\
 &\quad (\{2\}, 11), (\{2, 4\}, 14), (\{1, 2\}, 17), (\{1, 2, 4\}, 20), \\
 &\quad (\{1, 3\}, 23), (\{2, 3\}, 29), (\{1, 2, 3\}, 34)\}
 \end{aligned}$$

Hence the optimal subset is $\{1, 2, 3\}$ with $\sum_{j \in S} c_j = 34$.

Here I am showing how the DP-OPT algorithm actually runs on a certain problem instance. Notice that we just keep the item indexes in the set S , not the items themselves. So $(\{1, 2\}, 17)$ means that items 1 and 2 together have a cost of 17. You should study this example carefully in order to understand the algorithm.

IN210 – lecture 10



The FPTAS for 0-1 KNAPSACK-optimization combines the DP-OPT algorithm with rounding-off of input values:

j	1	2	3	4	5	6	7	
w_j	4	1	2	3	2	1	2	$K = 10$
c_j	299	73	159	221	137	89	157	

The optimal solution $S = \{1, 2, 3, 6, 7\}$ gives $\sum_{j \in S} c_j = 777$.

j	1	2	3	4	5	6	7	
w_j	4	1	2	3	2	1	2	$K = 10$
\bar{c}_j	290	70	150	220	130	80	150	

The best solution, given the truncation of the last digit in all costs, is $S' = \{1, 3, 4, 6\}$ with $\sum_{j \in S'} c_j = 740$.

Autumn 1999

19 of 22

The next step is to turn DP-OPT into an approximation algorithm APPROX-DP-OPT. APPROX-DP-OPT will essentially consist of two phases: Phase I will be rounding off the input values, and phase II will be running the algorithm that we have just seen, DP-OPT, on this modified instance.

These two phases together give us a FPTAS, the kind of miracle polynomial-time algorithm that for any given epsilon can produce solutions that are within epsilon from the optimum.

So before we go into details, let me just illustrate how these two phases work. On the foil is an instance of 0-1 KNAPSACK with 7 elements and K equal 10. The question is: Can we combine elements such that the total weight is no more than 10, and so that the elements combined have the maximum possible cost or value.

The best solution found by dynamic programming or branch and bound is elements 1, 2, 3, 6 and 7. The sum of their costs is 777. If we truncate the last digit of each element's cost, meaning approximating the costs by replacing the last digit with 0, then we have a different solution – namely elements 1, 3, 4 and 6, with total cost 740. This is about 5 percent from the optimal solution for the non-truncated costs. It is a good, decent solution.

IN210 – lecture 10

**Algorithm APPROX-DP-OPT**

- Given an instance I of 0-1 KNAPSACK-opt and a number t , truncate (round off downward) t digits of each cost c_j in I .
- Run the DP-OPT algorithm on this truncated instance.
- Give the answer as an approximation of the optimal solution for I .

Idea:

- Truncating t digits of all costs, reduces the number of possible “cost sums” by a factor exponential in t . This implies that **the running time drops exponentially**.
- **Truncating error** relative to reduction in instance size is “**exponentially small**”:

$$C_m = 53501 \overbrace{87959}^{\text{half of length}} \\ \text{but only } 10^{-5} \text{ of} \\ \text{precision}$$

Autumn 1999

20 of 22

We now give a rough pseudo-code of the APPROX-DP-OPT algorithm: Given an instance I of 0-1 KNAPSACK and a number t , it will first truncate (round off downward) t digits of each cost c_j in I . Then it will run the DP-OPT algorithm on this truncated instance. Finally it will return the answer from DP-OPT as an approximation of the optimal solution for I .

In practice, already with a little bit of truncation this works very well, as illustrated by the example on the previous foil. The question now is: “How can this be a way of getting around intractability?” Let me tell you how: Think of the pseudo-polynomial-time algorithm DP-OPT which basically works in time polynomial in n , C_m and some log-factor – where C_m is the largest cost in the input. The reason why DP-OPT is not a real polynomial algorithm is the size of the set M , which can be as big as nC_m . This size is exponential in the length of the input because C_m is coded in binary.

What happens when we truncate t digits? Eliminating t digits will actually reduce the number of possible total cost sums by a factor exponential in t . So there will be exponentially fewer possible elements in the set M , because each element in M has a different total cost. The “number value” of C_m will also drop exponentially. This means that the running time drops exponentially in t . But what happens with precision?

We are truncating from below. So who cares about these lower-order digits? Not much of precision is lost. Let’s assume C_m has 10 digits and $t = 5$, meaning that we truncate half of the digits. We then know that a pseudo-polynomial-time algorithm will gain something like 10^5 factor in running time, but in pre-

cision, what we have lost? We have lost only 10^{-5} of precision because we are keeping the higher order digits – throwing away the least significant bits.

I am giving you intuition. This intuition should suggest that in fact truncating is what does a miracle here, combined with a pseudo-polynomial time algorithm, because it gives us great savings in running time with a small loss in precision – exponential saving in running time with "exponentially small" loss of precision.

IN210 – lecture 10



Theorem 5 APPROX-DP-OPT is a FPTAS for 0-1 KNAPSACK-opt.

Proof: Let S and S' be the optimal solution of the original and the truncated instance of 0-1 KNAPSACK-opt., respectively. Let c_j and \bar{c}_j be the original and truncated version of the cost associated with element j . Let t be the number of truncated digits. Then

$$\begin{aligned} \sum_{j \in S} c_j &\stackrel{(1)}{\geq} \sum_{j \in S'} c_j \stackrel{(2)}{\geq} \sum_{j \in S'} \bar{c}_j \stackrel{(3)}{\geq} \sum_{j \in S} \bar{c}_j \\ &\stackrel{(4)}{\geq} \sum_{j \in S} (c_j - 10^t) \stackrel{(5)}{\geq} \sum_{j \in S} c_j - n \cdot 10^t \end{aligned}$$

1. because S is a optimal solution
2. because we round off downward ($\bar{c}_j \leq c_j$ for all j)
3. because S' is a optimal solution for the truncated instance
4. because we truncate t digits
5. because S has at most n elements

This means that they have an upper bound on the **error**:

$$\sum_{j \in S} c_j - \sum_{j \in S'} c_j \leq n \cdot 10^t$$

Autumn 1999

21 of 22

Now I am going to show you the proof that APPROX-DP-OPT is in fact a fully polynomial-time approximation scheme for 0-1 KNAPSACK. This means proving two things: One, that the *error* – meaning how far from the optimum the approximated solution can be – is bounded by the constant epsilon given as part of the input. Two, that the running time of APPROX-DP-OPT is bounded by a polynomial in n and $1/\epsilon$.

The main part of the proof is a series of inequalities giving a bound on the error. Let S and S' be the optimal solution of the original and truncated instance of 0-1 KNAPSACK, respectively. Let c_j and \bar{c}_j be the original and truncated version of the cost associated with element j . Finally, let t be the number of truncated digits.

The validity of the inequality series is explained on the foil. Together they give us a bound on the error, which is the difference between the optimal solution and the truncated solution. This error is less than or equal to $n \cdot 10^t$, where t is the number of truncated digits.

IN210 – lecture 10



- Running time of DP-OPT is $\mathcal{O}(n^2 C_m \log(n C_m W_m))$ where C_m and W_m are the largest cost and weight, respectively.
- Running time of APPROX-DP-OPT is $\mathcal{O}(n^2 C_m \log(n C_m W_m) 10^{-t})$ because by truncating t digits we have reduced the number of possible “cost sums” by a factor 10^t .
- Relative error ϵ is

$$\frac{\sum_{j \in S} c_j - \sum_{j \in S'} c_j}{\sum_{j \in S} c_j} \stackrel{(1)}{\leq} \frac{n \cdot 10^t}{C_m} \triangleq \epsilon$$
 1. because our assumption that each individual weight w_j is $\leq K$ ensures that $\sum_{j \in S} c_j \geq C_m$ (the item with cost C_m always fits into an empty knapsack).
- Given any $\epsilon > 0$, by truncating $t = \lfloor \log_{10} \frac{\epsilon \cdot C_m}{n} \rfloor$ digits APPROX-DP-OPT is an ϵ -approximation algorithm for 0-1 KNAPSACK-opt with running time $\mathcal{O}\left(\frac{n^3 \log(n C_m W_m)}{\epsilon}\right)$.

Autumn 1999

22 of 22

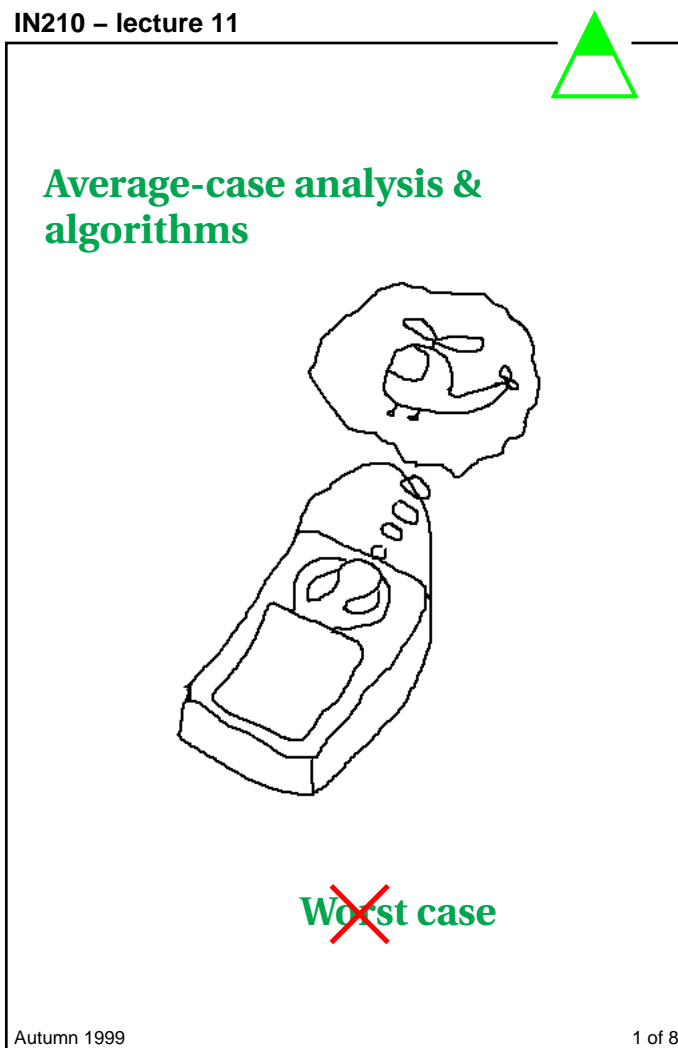
We have seen that the running time of the DP-OPT algorithm is bounded by $\mathcal{O}(n^2 C_m \log(n C_m W_m))$, where C_m and W_m are the biggest cost and weight, respectively. The running time of APPROX-DP-OPT is $\mathcal{O}(n^2 C_m \log(n C_m W_m) 10^{-t})$ because by truncating t digits we have reduced the number of possible “cost sums” – and thereby the maximum number of elements in M – by a factor 10^t .

The relative error is defined as $\frac{\text{optimal solution} - \text{approximated solution}}{\text{optimal solution}}$. It is no more than $n \cdot 10^t$ divided by C_m because we have already calculated an upper bound on “optimal solution - approximated solution”, and our assumption that each individual weight w_j is less than or equal to K ensures that $\sum_{j \in S} c_j \geq C_m$ (the item with cost C_m always fits into an empty knapsack).

We define $(n \cdot 10^t)/C_m$ to be equal to ϵ . Then given any $\epsilon > 0$ as input, by truncating $t = \lfloor \log_{10} \frac{\epsilon \cdot C_m}{n} \rfloor$ digits APPROX-DP-OPT is an ϵ -approximation algorithm for 0-1 KNAPSACK-opt with running time $\mathcal{O}\left(\frac{n^3 \log(n C_m W_m)}{\epsilon}\right)$.

Now we have come to the end. This whole thing is a proof that our approximation algorithm is a fully polynomial-time approximation scheme. The meaning of this is that given an epsilon, the algorithm can give us an approximation that is within epsilon from the optimum. The algorithm does it by choosing this t , which is depending on epsilon, truncating t digits and running the algorithm in time which is polynomial in the size of the input and in $1/\epsilon$.

3.11 Lecture 11



This is lesson eleven, and we have now constructed a whole theory, a kind of a map on which we placed problems according to their complexity or difficulty. And at this point we are looking at alternative approaches to algorithm analysis and design, which are based on some possible weaknesses of the theory that we have constructed. And which allow us to solve problems in practice that according to our theory seem exceedingly difficult.

Often there are problems which in the worst case cannot really be solved efficiently. There are always some problem instances that are too difficult to solve according to the theoreticians. But then there are people who deal with those problems routinely, who solve them and who would say to the theoreticians: "Who cares if your problem is \mathcal{NP} -complete? We have to solve those problems, and we do solve them."

So today we are going to talk about average-case analysis and corresponding algorithms. And I am first going to tell you a little story that is kind of a very strong argument against worst-case analysis, which is what we have done so far.

What you see in the picture on the foil is a computer scientist, a theoretician presumably, lying in bed in the morning as he woke up. He is thinking about his day, about what he is going to do. He is supposed to go to work and teach a class in algorithm theory, and he is thinking about his standard way of coming to work. Normally he would just go into his car, start the car and drive to work.

It takes about 15 minutes to get there.

But he is thinking about the worst case, the worst possible scenario, and he is thinking about whether his standard algorithm, namely driving to work, solves the problem efficiently in the worst case.

Now what is the worst case? You can think of all kinds of disasters happening, like his car breaking down when he tries to start it, the car not starting – there are all kinds of things that can happen. But ignore those, think that his car is working. Then what happens is: Right at the very moment when he is about to drive out of his garage, all the cars from the cities, they just for some reason pile up into his street, and then there is that endless row of cars. He cannot even come out of his driveway, not of speaking about getting to work on time. It takes the whole day to drive to work – at least.

So our computer scientist, still in bed, he decides that his standard solution in fact doesn't work. He either has to buy a helicopter to drive to work, or just give up the whole thing all together because the problem is hopeless.

This illustrates a situation with worst-case analysis. Worst-case analysis is not just pessimistic, it is in a way totally misleading. The good thing about it is that it can be done. It is manageable, theoretically. It is relatively easy to do the analysis. But basing our solutions on the worst case, on the worst possible instance of a problem, it is not a very good idea. It is not very natural and as we have seen, it can be very misleading.

So this is a warning to you to take everything that you hear in your class, especially also in this class, with a grain of salt. Analyze all theories carefully to see what they really mean. So a theory that is based on worst case is in fact useful, but it can also be in very misleading, and because of that we are able to solve in practice some problems that according to the worst-case theory seems intractable or unsolvable.

A more realistic approach to algorithm analysis is the average-case analysis where we try to not just look at the worst case, but we try to look at the typical situations, in a certain sense.

IN210 – lecture 11



- **Problem** = (L, P_r) where P_r is a probability function over the input strings:
 $P_r : \Sigma^* \rightarrow [0, 1]$.
- $\sum_{x \in \Sigma^*} P_r(x) = 1$ (the probabilities must sum up to 1).

- **Average time** of an algorithm:

$$T_A(n) = \sum_{\{x \in \Sigma^* \mid |x|=n\}} T_A(x)P_r(x)$$

- **Key issue:** How to choose P_r so that it is a realistic model of reality.
- **Natural solution:** Assume that all instances of length n are equally probable (uniform distribution).

Autumn 1999

2 of 8

In average-case analysis, in addition to the language L that defines a problem, we introduce a probability function which assigns to every possible string, or to every possible instance of a problem, a probability number between zero and one.

The meaning of this probability is obviously. It's the probability of that particular instance arising as an instance of the problem in practice. So the probability models the practical likelihood of an instance. And of course we want all the probabilities over the instances to sum up to 1, meaning 100%. Because we will get as input either this instance or that instance or that instance or ...

If we happen to have those probabilities, then we can estimate the average time of an algorithm as the sum over all possible instances of length n of the time the algorithm takes on that instance times the probability of the instance arising. You would call this 'expectation' in probability theory. So this is expected time or average time.

The problem in practice is of course how to choose this function P_r so that P_r is more or less realistic. The way this problem is solved routinely, is to say: Since we really don't know anything about the practical instances, let's assume that all instances of length n are equally probable.

(This is a blank page)

IN210 – lecture 11



Random graphs

Uniform probability model (UPM)

- Every graph G has equal probability
- If the number of nodes = n , then

$$P_r(G) = \frac{1}{\#\text{graphs}} = \frac{1}{2^{\binom{n}{2}}}, \text{ where } \binom{n}{2} = \frac{n(n-1)}{2}$$
- UPM is more natural for interpretation

Independent edge probability model (IEPM)

- Every possible edge in a graph G has equal probability p of occurring
- The edges are independent in the sense that for each pair (s, t) of vertices, we make a new toss with the coin to decide whether there will be an edge between s and t .
- For $p = \frac{1}{2}$ IEPM is identical to UPM:

$$P_r(G) = \left(\frac{1}{2}\right)^m \cdot \left(\frac{1}{2}\right)^{\binom{n}{2}-m} = \frac{1}{2^{\binom{n}{2}}}$$

- IEPM is easier to work with

Autumn 1999

3 of 8

So the most natural way to assign probabilities is to assume that all instances of length n have the same probability of occurring. And if our instances are graphs, then this way of assigning probabilities to instances gives rise to a random graph model called the *uniform probability model (UPM)*. In UPM we assume that we have a hat, and in this hat we have all labeled graphs on n vertices – ‘labeled’ meaning that the vertices are distinct. And then we just close our eyes and take out of the hat a graph at random.

What is the probability that a given graph will be picked from the hat? Since we draw graphs on random, then the probability that a given graph is picked is obviously 1 divided by the number of graphs in the hat. And in the hat we have all labeled graphs on n vertices. Let’s count them:

Given n labeled (distinct) vertices, how many possible pairs of vertices are there? There are n ways of choosing the first element in a pair of nodes. And as the second element you can choose everybody else except the first node. That gives us $n-1$ possibilities for the second element. We don’t permit you to choose the first element again, because our graphs don’t have *loops*, meaning edges of the form (x, x) .

So we have $n(n-1)$ ways of choosing two vertices. Now we have to divide this by 2 because it doesn’t matter whether we choose the first one first and the second one second, or the second one first and the first one second, meaning: Edge (x, y) is the same as edge (y, x) because our graph is undirected.

The number of pairs of vertices is then $n(n-1)/2$. This number is also known as $\binom{n}{2}$ – the number of ways to choose 2 elements out of n distinct ones. In a

given graph each such pair of vertices can either be an edge or it can be a non-edge. That gives us 2 choices for each pair of vertices, and all together we can construct $2 \cdot 2 \cdot 2 \cdots 2 = 2^{\binom{n}{2}}$ different graphs.

So there are $2^{\binom{n}{2}}$ possible labeled graphs on n vertices, and each one of them has the probability $1/2^{\binom{n}{2}}$.

The uniform probability model is a very natural model. The problem with UPM is that it is very difficult to work with. Why? Because if it turns out that if my random graph has an edge here, then the fact that this being an edge influences the probability of this other pair of vertices being an edge and this other pair and this other pair and so on. It turns out there is a lot of difficulties in analyzing the running time of an algorithm on this kind of graph model.

A much easier model to use in algorithm analysis is the so-called *independent edge probability model (IEPM)*, where each pair of vertices is an edge with probability p , some probability, and where the probability of this being an edge and that being an edge, they are independent probabilities.

To create graphs in this model you can think that you consider each pair of vertices and toss a coin to decide whether that pair should be an edge or not. And if the coin turns up head, which happens with probability p , then you make that an edge, else you make it a non-edge. You do that for every pair of vertices, and in the end you have your graph g .

If you want to calculate the probability of the outcome of this experiment being that particular graph g , then you just have to multiply all the individual probabilities – $\binom{n}{2}$ of them all together. This is legal because every coin toss is an independent event. They don't interfere with each other, per definition.

So if your graph g happens to have m edges and $\binom{n}{2} - m$ non-edges, then the probability for that graph occurring is $p^m \cdot (1 - p)^{\binom{n}{2} - m}$, where $(1 - p)$ is the probability for a non-edge.

If this coin is a fair coin, meaning that it turns up head with the probability $1/2$, then we have p equals $1/2$. The overall probability for a given graph is then $1/2^{\binom{n}{2}}$, namely exactly the same probability as in the uniform probability model. This is what happens if p is equal to $1/2$. If p is something else, then the probabilities will differ in these two models.

I will not say more about this, but the point is that IEPM is in a certain sense a model of this situation with the hat. It is a way for us to assign probabilities to graphs in a manner which is more conducive to analysis and which still can be interpreted in this very natural way.

IN210 – lecture 11

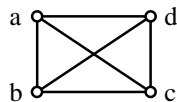
**Example: 3-COLORABILITY**

In 3-COLORABILITY we are given a graph as input and we are asked to decide whether it is possible to color the nodes using 3 different colors in such a way that any two nodes have different colors if there is an edge between them.

Theorem 1 3-COLORABILITY, which is an \mathcal{NP} -complete problem, is solvable in **constant average (expected) time** on the IEPM with $p = 1/2$ by a branch-and-bound algorithm (with exponential worst-case complexity).

Proof:

Strategy (for a rough estimate): Use the indep. edge prob. model. Estimate expected time for finding a proof of non-3-colorability.



K_4 (a clique of size 4) is a proof of non-3-colorability.

Once we have the probabilities, then the question is: What can we do? What are the kinds of results that can come out of this average-case analysis? And here is one sort of shocking example:

We are looking at the 3-COLORABILITY problem, which is an \mathcal{NP} -complete problem. The question of 3-COLORABILITY is: Can we color the vertices of a graph with 3 colors so that both endpoints of any edge have different colors?

You can think of an application of this as being coloring a map. You can easily represent a map by a graph by representing countries with vertices and representing borders between countries as edges. If two countries have a border, then obviously these adjacent countries would need to be colored in different colors. So that is one application of this graph colorability problem.

The question we are asking is: Can we color the map with 3 colors? And the problem is known to be \mathcal{NP} -complete. However, if we look at the average case, then we can prove that 3-COLORABILITY is solvable in constant expected time by a branch-and-bound algorithm, assuming the IEPM with $p = 1/2$.

Now, this is completely shocking because we are talking about an \mathcal{NP} -complete problem, and we know that most probably \mathcal{NP} -complete problems cannot even be solved in polynomial time. But I claim that it can be solved in constant average time.

We know that a branch-and-bound algorithm takes exponential time in the worst case. And constant time is even a lot shorter time than polynomial time or than linear time. In the worst-case analysis you need linear time even to see the whole instance, to read the data. You have to go through the entire input.

Here you are not even going through the entire input: In constant time, say in exactly 197 steps, you figure out the solution no matter how large the instance is. This sounds like a miracle.

This sort of puts all our ideas about complexity in question. Let's see into the miracle a little bit. How does this happen? Intuitively speaking you can say that this happens because with IEPM and $p = 1/2$ almost all graphs are non-3-colorable, and there are lots of witnesses that prove it. So after just checking a constant number of potential witnesses, you are expected to find a witness that proves that your instance is a 'No'-instance.

The proof of this theorem is a research paper, and the actual result is based on analysis on a kind of recurrence relations that allow one to calculate the expected time of the branch-and-bound algorithm more or less exactly. We are not going to do that. I am just going to show you the reason why this works. We are going to do just a very rough estimate.

Think of the branch-and-bound algorithm as either producing a proof of NON-3-COLORABILITY or of coloring the graph in 3 colors. What is the proof of NON-3-COLORABILITY like? Here is one possibility: If my algorithm discovers this subgraph in the input graph, then it can immediately say: "Look, this is not 3-colorable, because this subgraph K_4 is already not 3-colorable. You need 4 colors to color this complete graph on 4 vertices." So as soon as you discover a K_4 in your input, you say: "I give up. This is not 3-colorable. I am done."

IN210 – lecture 11



- The probability of 4 nodes being a K_4 :

$$P_r(K_4) = 2^{-\binom{4}{2}} = 2^{-6} = \frac{1}{128}$$

- Expected no. of 4-vertex sets examined before a K_4 is found:

$$\begin{aligned} \sum_{i=1}^{\infty} i(1-2^{-6})^{i-1}2^{-6} &= 2^{-6} \sum_{i=1}^{\infty} i(1-2^{-6})^{i-1} \\ &\stackrel{*}{=} 2^{-6} \frac{1}{(1-(1-2^{-6}))^2} \\ &= 2^{-6} \frac{1}{(2^{-6})^2} = \frac{2^{12}}{2^6} = 2^6 = 128 \end{aligned}$$

— $(1-2^{-6})^{i-1}2^{-6}$ is the probability that the first K_4 is found after examining exactly i 4-vertex sets.

— (*) is correct due to the following formula ($q = 1-2^{-6}$) from mathematics (MA100):

$$\begin{aligned} \sum_{i=1}^{\infty} iq^{i-1} &= \frac{\delta}{\delta q} \left(\sum_{i=1}^{\infty} q^i \right) = \frac{\delta}{\delta q} \left(\frac{q}{1-q} \right) \\ &= \frac{1}{(1-q)^2} \end{aligned}$$

Autumn 1999

5 of 8

In a random graph based on the independent edge probability model and with $p = 1/2$, then the probability of four given vertices being a complete graph is $1/2^{\binom{4}{2}}$. Because there are $\binom{4}{2}$ possible graphs on 4 vertices and only one of them is a K_4 . This probability is equal to $2^{-6} = 1/128$ which is a constant probability.

So if I look at 4 particular vertices, the probability that they are a K_4 is $1/128$. Now the question is: What is the expected number of 4-vertex sets that we need to examine before we can hope to find one configuration like this?

The expected number of trials is $\sum_{i=1}^{\infty} i(1-2^{-6})^{i-1}2^{-6}$. What does this sum say really? i is the number of 4-vertex sets examined. It might seem a bit strange to take the sum from i to infinity because given an n we have only a finite number of possible 4-vertex sets. But we will see that the expectation is going to be a constant, even when we take the sum to infinity. So this will only strengthen the proof.

We know from basic probability theory (ST001/100) that in order to compute the exactation we must sum up number of steps used times the probability for using exactly that many steps. So i is the number of steps used – or here, the number of 4-vertex sets examined. And $(1-2^{-6})^{i-1}2^{-6}$ is the probability that the first K_4 is found after examining exactly i 4-vertex sets. So first we have examined $i-1$ sets which were not K_4 's, and the probability that a 4-vertex set is not a K_4 is $1-2^{-6}$. Then we found a K_4 , and that has probability 2^{-6} .

This is quite technical, but there is a reason why I am going through it. I

want to show you a little bit of this probabilistic calculations, and also make a point in the end.

The question now is: How do we estimate this sum? It is not an easy sum to calculate. So we use a trick. First, because 2^{-6} is a constant we can move it out of the sum, and then we have a sum of the form iq^{i-1} , where i ranges from 1 to infinity and q is a constant.

How do we calculate such a sum? We use a little trick known from MA100 that is called the *formal derivation*. Think of this sum here $\sum q^i$, which is a nicer sum because we have got rid of one of the i 's. If we compute the derivative of this sum with respect to q , then we get $\sum iq^{i-1}$.

So let's first sum up $\sum_1^\infty q^i$. This sum we know how to sum up because it is the standard kind of geometric series. The answer is $q/1 - q$. When we compute the derivative of this, we get $1/(1 - q)^2$.

This means that we can basically substitute $2^{-6} \sum_1^\infty (1 - 2^{-6})^{i-1}$ with $2^{-6} \cdot 1/(1 - (1 - 2^{-6}))^2$, and now we have just a constant expression, which can be shown to be equal to 2^6 or 128.

IN210 – lecture 11



Conclusion: Using IEPM with $p = \frac{1}{2}$ we need to check 128 four-vertex sets on average before we find a K_4 .

Note: Random graphs with constant edge probability are very dense (have lots of edges). More realistic models has p as a function of n (the number of vertices), i.e. $p = 1/\sqrt{n}$ or $p = 5/n$.

Autumn 1999

6 of 8

So using the IEPM with $p = 1/2$ we need to check 128 4-vertex sets on average before finding a K_4 . But watch what have happened: We went through this whole calculation, which is pretty mind-boggling, in order to figure out something which should have been obvious in the first place. Why? Look at the probability for a 4-vertex set being a K_4 . The probability of that event happening is $1/128$, it is 1 over something. What does it mean? This means that this happens 1 time out of 128. So the expected number of trials that we need to make in order for this event to happen once, should be 128.

The point is that in probability you can use calculations, and that is pretty messy, but you can also just use common sense and understand what the result should be. Here the result should obviously be 128. Common sense is a very powerful tool in probability.

So the bottom line is that if your branch-and-bound algorithm were not doing anything else but actually looking for counter-examples of 3-COLORABILITY of the form of a K_4 , then after examining 128 4-vertex subgraphs, it is likely to find a counter-example. So the expected running time for an algorithm that was just looking for a K_4 , would be 128 times some constant.

After being dug down in details, let's try to see what I am really saying here. What are the basic insights? To begin with, this class is about theory modeling practice. That's the main thing. So we construct theory as a model of practice. At this point I am trying to sort of teach you a little bit about this business, how everything in life is relative, and how even the nicest possible theory can be misleading. When you look at it as a theory it is such a beautiful thing. You tend

to believe in everything it says. And then you change the way of looking, you look from a certain angle and you can tear the theory into pieces. You can say: Look, how different life can really be from what the theory says.

So our theory says that \mathcal{NP} -complete problems they cannot be solved in polynomial time. You need exponential time unless $\mathcal{P}=\mathcal{NP}$ but we believe that \mathcal{P} is different from \mathcal{NP} . And then we know that branch-and-bound algorithms, they take exponential time.

But I have just shown you an example where an \mathcal{NP} -complete problem is solved in constant time by a branch-and-bound algorithm when we use a criterion for computing time which is in fact more realistic practically speaking, namely average case instead of worst case.

The point is that when you change the way of looking, then that way of looking can be more realistic, and with that more realistic way of looking you may in fact draw conclusions which are just totally different from what you had before.

And then we did a little bit of calculation to show that calculations can be very clever and still have the result that can be guessed right away if you just use common sense. That is another insight.

So what is this branch-and-bound algorithm that is going to 3-color the graph doing really? Instead of trying all possible 3-coloring – by branching – the algorithm is focusing on the bounding part. It is trying find a counter-example. If it finds a counterexample, this algorithm is going to stop and say: "No, this graph is not 3-colorable." And the counterexample which disproves 3-coloring is a K_4 , a complete subgraph on 4 vertices. And we have seen that the algorithm on average needs to check 128 4-vertex subsets before finding a K_4 .

I am now going to criticize this approach. I am going to tell you that this what I have just done is, in a certain sense, misleading. Here is why: A graph in which each pair of vertices is an edge with probability $1/2$ is a very dense graph, dense meaning having many edges. What are graphs? They are relations, say, brothers or sisters or friends. Think that you have n people and n being, say, 4 billions or 5 billions. How many pairs of these people are brothers? Or how many pairs do know each other? Whatever your relation is, it is not that if you pick two random people then the probability that they are brothers is $1/2$. The probability that the two people are brothers is very, very small. The point is that these dense graphs are not really natural problem instances. Natural problem instances are much, much sparser, sparse meaning few edges. And in a sparse graph the probability of any 4 vertices being a K_4 is much lower.

So this graph model where every graph has the same probability independent of n , can actually be very misleading. Because that graph model favors very dense graphs which never arises in practice. If you have 10 countries, than a country could be neighbor with half of the other countries. But if you have 100 countries, how many neighboring countries does a country typically have? Certainly not 50. A country cannot be a neighbor to $1/2$ of the other countries in this case. So $p = C$ where C is a constant not depending on n is in fact an unrealistic assumption.

So we would need to be more careful about our choice of probability function. And for a practical problem the function \mathcal{P} will typically not be a constant, but it will be some kind of function of n . Maybe \mathcal{P} is equal to $5/n$ or something like that, meaning that each country in average is a neighbor with 5 other countries. This would give you a graph where a typical vertex has 5 neighbors. That is more realistic.

IN210 – lecture 11

**0-1 Laws**

as a link between probabilistic and deterministic thinking.

Example: “Almost all” graphs are

- not 3-colorable
- Hamiltonian
- connected
- ...

Def. 1 *A property of graphs or strings or other kind of problem instances is said to have a **zero-one law** if the limit of the probability that a graph/string/problem instance has that property is either 0 or 1 when n tends to infinity ($\lim_{n \rightarrow \infty}$).*

Autumn 1999

7 of 8

And with this kind of p , this average-case analysis would look quite a bit different. However, we can learn something from this example that is very important. That is our next story, and the story is about 0-1 laws. This is one chapter of probabilistic analysis of algorithms, problems or sets in general.

My story will be rather intuitive. Intuitively 0-1 laws are a link between probabilistic and deterministic thinking. And let me tell you a good example of this link. Think about your program running on the computer, and the probability of a meteorite hitting into the computer and into your building during the time the algorithm is running.

You will say: "Who cares? These things don't happen." Well, there is a probability that this will happen. So you may or may not take that probability into account. Typically you don't. You say: "Come on, there is a lot more probable that the electrical circuit will jam. There are all kinds of disasters that can happen."

But typically you don't think in terms of all kinds of disasters. Why? Because intuitively you divide events into low-probability events and high-probability events. So although this life and this world they are fundamentally probabilistic – there is some probability of all kinds of things happening – typically events have either probability 0 or probability 1. Probability 0 meaning that they typically don't happen, probability 1 meaning that they typically do happen. You wait for a train or you wait for a bus. Typically this bus comes – maybe a little late, but typically it comes more or less on time. So you typically expect that certain things will happen and that certain things won't happen.

So it is with graphs. It turns out that most of the properties you think about testing in graphs – such as Hamiltonicity, connectedness, 3-Colorability – if you pick a random graph model, whatever this model is, the probability of your graph having the property is typically either 0 or 1. Technically we say that graphs properties tend to have *0-1 laws*.

So we say that a property of graphs, strings, problem instances or whatever, has a zero-one law if the limit of the probability that a graph/string/problem instance has that property is either 0 or 1 when n tends to infinity ($\lim_{n \rightarrow \infty}$).

If we calculate the probabilities of a graph being Hamiltonian or connected or 3-colorable, then we will see that when \mathcal{P} is very small, e.g. $5/n$, then with probability 1 our graph will not be connected, our graph will not be Hamiltonian and our graph will be 3-colorable, because there are just about no edges.

And then as \mathcal{P} increases, at a certain very distinct point called the *threshold*: Bang, the things change! So up to \mathcal{P} of the threshold for a connectivity, the graph will be disconnected, and then: Bang! The graph will almost certainly be connected.

As an example, it has been calculated that the threshold function for a K_4 is $p = n^{-2/3}$, meaning that if p is less than $n^{-2/3}$ then almost no graph has a K_4 (and therefore almost all graphs are 3-colorable), while if p is greater than $n^{-2/3}$ then almost every graph has a K_4 (and virtually no graphs are 3-colorable).

0-1 laws allow us to construct algorithms which works efficiently on average, because we use this fact that a typical instance is either a positive instance or a negative instance. So in this example that we have just seen, we have seen that with a very large probability a random graph with $p = 1/2$ is not 3-colorable. It is just a matter of finding a proof of NON-3-COLORABILITY, and these proofs they are found in almost no time. There are zillions of them all over the graph, it is just a matter of finding one. So we find one with a large probability in very short time (constant time).

The same kind of idea can be used in all sorts of different ways because of the 0-1 laws.

IN210 – lecture 11

**Example: HAMILTONICITY**

a linear expected-time algorithm for random graphs with $p = 1/2$.

- **Difficulty:** The probability of non-Hamiltonicity is too large to be ignored, e.g. $P_r(\exists \text{ at least 1 isolated vertex}) = 2^{-n}$.
- The algorithm has 3 phases:
 - **Phase 1:** Construct a Hamiltonian path in linear time. Fails with probability $P_1(n)$.
 - **Phase 2:** Find proof of non-Hamiltonicity or construct Hamiltonian path in time $\mathcal{O}(n^2)$. Unsuccessful with probability $P_2(n)$.
 - **Phase 3:** Exhaustive search (dynamic programming) in time $\mathcal{O}(2^{2n})$.
- Expected running time is

$$\leq \mathcal{O}(n) + \mathcal{O}(n^2) P_1(n) + \mathcal{O}(2^{2n}) P_1(n) P_2(n)$$

$$= \mathcal{O}(n) \text{ if } P_1(n) \cdot \mathcal{O}(n^2) = \mathcal{O}(n)$$

$$\text{and } P_1(n) P_2(n) \cdot \mathcal{O}(2^{2n}) = \mathcal{O}(n)$$
- Phase 2 is necessary because $\mathcal{O}(2^{-n}) \cdot \mathcal{O}(2^{2n}) = \mathcal{O}(2^n)$.
- After failing to construct a Hamiltonian path fast in phase 1, we first reduce the probability of the instance being non-Hamiltonian (phase 2), before doing exhaustive search in phase 3.

Autumn 1999

8 of 8

We will now show another example by constructing an algorithm for HAMILTONICITY which runs in linear expected time on a random graph model with $p = 1/2$.

The difficulty here is that we cannot just blindly try to construct a Hamiltonian path, even though we know that most likely there will be one. The probability of the graph being non-Hamiltonian is too large to be ignored. If the graph is non-Hamiltonian and we cannot prove it in some clever way, then we have to do exhaustive search. We have to examine all possibilities.

What is the probability of a graph being non-Hamiltonian? It is not very easy to estimate that, but it is easy to compute a lower bound. Think of a vertex v . What is the probability of v being isolated, meaning not having any edges? There are $n - 1$ possible neighbors to v . Each of these is a non-edge with probability $1/2$. So $2^{-(n-1)}$ is the probability of this vertex being isolated. This is an exponentially small probability.

This probability is a lower bound on the probability of your graph G being non-Hamiltonian, because if you have an isolated vertex then certainly the graph is not Hamiltonian. So the probability of a graph being non-Hamiltonian is certainly greater than 2^{-n} .

It is an exponentially small probability but it is a large probability if with this probability we are going to run the exhaustive search algorithm. Because the exhaustive search algorithm takes more time than 2^n . The naive exhaustive search algorithm would take $\mathcal{O}(n!)$ which is much bigger than $\mathcal{O}(2^n)$. Even a more optimized exhaustive search algorithm uses $\mathcal{O}(2^{2n})$.

So if with probability 2^{-n} we are running an algorithm that takes time 2^{2n} , then this here is much bigger than any polynomial – $2^{-n}2^{2n} = 2^n$ is exponential. So if with probability 2^{-n} we are running an exhaustive search algorithm, then the overall performance of our algorithm is exponential.

There is a published algorithm for HAM, which has linear expected time using the IEPM with constant edge probability. This algorithm has 3 phases:

Phase 1 will construct a Hamiltonian path in a kind of a quick-and-dirty greedy way in linear time, and fail with a certain probability. Call that probability P_1 . So for most graph instances this will be successful, but with some small probability P_1 it will fail.

In case phase 1 is successful, then of course you are done. The Hamiltonian path is constructed, and there is nothing more to do. If phase 1 fails, then phase 2 is activated. Phase 2 will find a proof of NON-HAMILTONICITY or construct a path. Phase 2 takes a little more time – it is a little more careful. It uses $\mathcal{O}(n^2)$ time, so it can look at all pairs of vertices and things like that. What phase 2 will typically do, it will eliminate some odd situations like if you have an isolated vertex. But also if you have a vertex with degree 1, then the graph is non-Hamiltonian.

Although NON-HAMILTONICITY is difficult to check in general, there are certain NON-HAMILTONICITY proofs that are easy to check. And the trick here is to find a kind of criterion for NON-HAMILTONICITY, which happens quite often. So if the graph is non-Hamiltonian, then it will have this kind of non-Hamiltonian property, which is easy to check. It doesn't always work of course, but it works in a lot of instances. So you check for that kind of non-Hamiltonian property, and either you find that your graph has that property or if it doesn't, then it turns out that it is fairly easy to construct a path in quadratic time.

So again this phase 2 will work with a large probability, but with some small probability P_2 it will fail. And in case it fails, then you run phase 3, which is exhaustive search using dynamic programming. This can be done in time $\mathcal{O}(2^{2n})$.

My point is that all of these 3 phases they are necessary. Exhaustive search is necessary for your algorithm to always work. Phase 2 is necessary because you do need to deal with NON-HAM in some way, and resolve it for most of the non-Hamiltonian instances – only phase 3 will take of NON-HAMILTONICITY in the general case. And phase 1 you need for the whole thing to be efficient in most cases – to construct a quick and dirty solution.

Now a few words about the analysis of this algorithm: The expected running time is the expected running time for phase 1, $\mathcal{O}(n)$, plus the contribution of phase 2 and phase 3. The contribution from phase 2 is not quadratic time. It is $\mathcal{O}(n^2)$ times the probability that phase 2 will ever be activated, which is P_1 . And then the contribution of phase 3 is $\mathcal{O}(2^{2n})$ times $P_1 \cdot P_2$, the running times of the exhaustive search algorithm times the probability that both P_1 and P_2 fails.

So the whole algorithm is linear if $P_1 n \cdot \mathcal{O}(n^2)$ and $P_1 n P_2 n \cdot \mathcal{O}(2^{2n})$ is both $\mathcal{O}(n)$.

This algorithm is studied in detail in IN391 which is normally taught in the fall. Here I have just been trying to describe to you what is really involved in solving an \mathcal{NP} -complete problem by an algorithm which has efficient average-behavior and always produces a correct solution. This algorithm that I have described is a kind of a pattern. Those phases are something that you will always have to do in order to produce this kind of solution.

Next time I am going to talk about randomized algorithms and parallel algorithms – two other kinds of analysis and kinds of algorithm design.

3.12 Lecture 12

IN210 – lecture 12



Randomized computing

Machines that can **toss coins** (generate random bits/numbers)

- Worst case paradigm
- ~~Always~~ give the correct (best) solution

Algo:

**10 (265-281),
11 (309-323)**

Autumn 1999

1 of 12

Today we continue with alternative approaches to computation and to algorithms. The idea here is to combat intractability by using alternative approaches to either computation or to algorithm analysis. The reason why we have this possibility is that all the classical theory of algorithms that we have seen is based on the worst case & best solution approach, which is kind of unrealistically pessimistic and demanding – it is kind of greedy. Why do you worry about the worst case if it practically never happens, and why do you want the very best solution if there are really just very few of them? And if there are many of those best solutions, then just pick any one of them. And so on.

So we have seen some very strong arguments against the worst case & best solution approach. Today we are about to see a couple of other approaches to computation and to algorithm design and analysis. The first one being randomized computing, and the second one parallel computing. We will finish this theme next time by looking at another kind of computing device, namely the quantum computer.

What is randomized computing? It uses machines or algorithms that can make decisions based on "coin tosses". I put this in quotes. In reality this means that the machines can generate and use random bits or random numbers, and then make decisions based on those.

Now you wonder right away: Why would a machine use random numbers? Why would a machine toss a coin? Isn't computation something from which we

expect certainty, not randomness? Isn't randomness something that we want to avoid as much as possible from the machines? It is enough that people are random and events in nature. Why would we make a random machine?

That is a good question, and I hope to answer that question right away by showing you an example. Notice that we will still be stuck in the worst-case paradigm, but in randomized computing we will permit that the computer does not always give the correct/best solution.

IN210 – lecture 12



Randomized algorithms



Idea: Toss a coin & simulate non-determinism

Example 1: Proving polynomial non-identities

$$(x + 2)^2 \stackrel{?}{\neq} x^2 + 2xy + y^2$$

$$\stackrel{?}{\neq} x^2 + y^2$$

- What is the “classical” complexity of the problem?
- Fast, randomized algorithm:
 - Guess values for x and y and compute left-hand side (LHS) and right-hand side (RHS) of equation.
 - If $\text{LHS} \neq \text{RHS}$, then we know that the polynomials are different.
 - If $\text{LHS} = \text{RHS}$, then we suspect that the polynomials are identical, but we don't know for sure, so we repeat the experiment with other x and y values.
- Idea works if there are many witnesses.

Autumn 1999

2 of 12

Our strategy is to illustrate each of these alternative approaches by showing a situation where the approach really works well. I am now showing you such a situation for randomized computation.

The problem here is proving POLYNOMIAL NON-IDENTITIES. What does this mean? Given two polynomials as input, we want to decide whether these two polynomials are different or not. Two polynomials are different if there is at least one set of values for the variables such that the two polynomials give different answers. They are identical if they compute the same value for all values of the variables.

So what is involved in testing this kind of thing? This is essentially theorem proving, and we have said already that theorem-proving problems tend to be very hard. I would encourage you to think about the complexity of this problem – what is it? Is this problem undecidable? Is it \mathcal{NP} -hard or \mathcal{NP} -complete? What is it? And how can we in fact figure this out?

I will just give you some hints, and let you think about the rest. We have said that basically theorem-proving problems tend to be undecidable. Because as soon as you can simulate a Turing machine with some kind of language, some kind of expression, some kind of axiom system, then proving theorems would in a certain sense be equivalent to showing that a TM doesn't halt or halt.

That is one side of the story. Another side of the story is that this kind of problem would seem to be in \mathcal{NP} . And let me tell you why: If these two polynomials are not identical, then a NTM can simply guess values of x and y for which the two expressions evaluate to different values, and then say 'No'. So POLYNOMIAL NON-IDENTITIES would seem to be in \mathcal{NP} , while POLYNOMIAL IDENTITIES would seem to be in $\text{Co-}\mathcal{NP}$.

What is the difference between the two situations? Is this a proof that POLY-

NOMIAL NON-IDENTITIES is in \mathcal{NP} ? No, there is a big whole in this argument, and it has to do with the size of the numbers. At the present there is no bounds on how big the numbers can be. So a NTM would apparently need to guess arbitrarily large numbers in order to prove that the polynomials are different. That is not in our scheme of things. Everything has to be polynomial if a problem is going to be in \mathcal{NP} . If these two polynomials are not identical, then there are witnesses, but it seems that they can be arbitrarily large.

In fact, it turns out that the witnesses cannot be arbitrarily large. There are limitations to this. And this has to do with a very interesting area of mathematics and algorithm theory, which is related to geometry of numbers. And I am not going to go into that, but the whole point is that there are ways of calculating the limits to the size of these witnesses.

So I will just mention geometry of numbers. And this geometry of numbers has to do with this theory of linear programming which we have seen before. Looking at numbers in a coordinate system, and just looking how big or small they can be and so on. This is a very interesting area to research in.

But this is just proving that the problem is, say, solvable in polynomial space. Still, even if the witnesses are limited and so on, it seems that in order to disprove this non-identity – meaning to prove that the polynomials are identical – we would need to manipulate these two polynomials by using some kind of transformation, as a mathematician would do.

How does a mathematician prove that these two polynomials are identical? She uses certain rules. She says that $(x + y)^2$ is equal to $(x + y)(x + y)$ and this is further equal to $x(x + y) + y(x + y)$, etc. But there are no rules explaining how to use these rules, so she uses trial and error. And then at some point she either gets the right-hand side or she doesn't. If she do, she says: "Bingo, I succeeded, I proved a theorem. I proved that the polynomials are identical."

So proving or disproving theorems is definitely very hard, and especially proving theorems by an algorithm, by a machine. So it is a hard problem. Let us not worry about the details of its complexity too much.

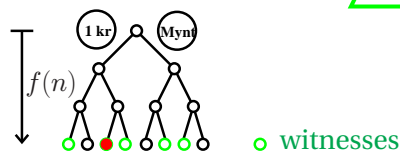
Now I am going to show you that this hard problem is in practice made very easy by the probabilistic approach. Here is why: If these two polynomials turn out to be non-identical, then they may evaluate to the same value for a small number of choices of x and y . But actually most of the choices of x and y will give different values of the right-hand side and of the left-hand side. So the point is that we don't really need to go through all this calculation to prove that the two polynomials are different or identical. It is enough to pick a couple of values for x and y , evaluate those values in the right-hand side and in the left-hand side, and calculate. And if the two values turn out to be different, then we say: "Yes, the two polynomials are not identical. We have found a witness."

Evaluating a polynomial for two small values, that is obviously efficiently computable. It is a very easy problem. What you do is: You evaluate the left-hand side and the right-hand side, compare the two values and 'bang'! If the values are different, then you say that the two polynomials are not identical.

But what if the values are the same, can you then say with certainty that the polynomials are identical? No, because obviously they have to evaluate to the same for all possible numbers, not just for one choice of x and y . So what do you do then? Well, you repeat the procedure. You pick another two numbers, evaluate the left-hand side and the right-hand side again, and compare the results. And again if the results are different, then you say the polynomials are not identical. If the results are the same, what then? Repeat again!

So this can obviously go indefinitely, but every time you repeat, you are more certain that the two polynomials are in fact identical. You can never be completely certain, but we will see in a moment how well this is doing.

IN210 – lecture 12



Let $f(n)$ be a polynomial in n and let the probability of success after $f(n)$ steps/coin tosses be $\geq \frac{1}{2}$. After $f(n)$ steps the algorithm either

- finds a witness and says “Yes, the polynomials are different”, or
- halts without success and says “No, maybe the polynomials are identical”.

This sort of algorithm is called a **Monte Carlo algorithm**.



Note: The probability that the Monte Carlo algorithm succeeds after $f(n)$ steps is **independent of input** (and dependent only on the coin tosses).

- Therefore the algorithm can be repeated on the same data set.
- After 100 repeated trials, the probability of failure is $\leq 2^{-100}$ which is smaller than the probability that a meteorite hits the computer while the program is running!

Autumn 1999

3 of 12

So, this is the algorithm. Now I will formalize a little bit. The algorithm actually uses coin tosses. How does it use coin tosses? It uses coin tosses to generate the values of x and y . You can think of this algorithm as tossing a coin and generating a random number of random bits: 1001101 – call this x – and then 10110, this is y . So these random bits they are randomness in this algorithm.

We want to embed these random algorithms into our existing theory, into our map. That is important because we do want to still use the basic map that we have developed in the first part of the class. We want to explain, by using that map, what we can do with this new approach.

So think of this random algorithm as being something like a canonical NTM that can toss coins. As you might remember, a canonical NTM is a NTM that always has exactly two possible transitions, and which halts after exactly $f(n)$ steps – if it halts. Think that this canonical NTM is solving a problem that is in \mathcal{NP} . It is solving the problem by finding a witness in $f(n)$ time, where $f(n)$ is a polynomial.

I have drawn the computation tree of the canonical NTM on the foil. These green leaves represent the outcomes where the machine has found a witness. Imagine now that there are many 'Yes'-witnesses which can be found after $f(n)$ steps. In our example we have that kind of case because if the two polynomials are non-identical, then they will evaluate to different numbers for a large proportion of choices of x and y .

If we simply use random bits or a coin to decide which way to go in the computation tree, then all of these leaves will have the same probability. And if,

say, half of these leaves represent successful computations that result in finding a witness, then the probability that the corresponding randomized machine will give the right answer for a positive instance is at least $1/2$.

That is roughly the situation we have when proving non-identities. We have the probability of success, which is at least $1/2$. After $f(n)$ steps our probabilistic algorithm will either find a witness and say "Yes, the polynomials are different", or it will halt without success and say "No, maybe the polynomials are identical." Notice that if the machine doesn't find a witness after $f(n)$ steps, then it cannot be certain whether its input is a positive instance or a negative instance, so it says only 'maybe no'. This kind of algorithm is called a *Monte Carlo algorithm*, Monte Carlo being a big gambling place in Italy.

So this is an informal definition of a Monte Carlo algorithm, and you can now understand why I have presented the problem as 'proving non-identities', instead of the more natural 'proving identities'. Because a Monte Carlo algorithm has only positive witnesses – 'Yes'-witnesses. There are no negative witnesses. It cannot say 'No' with certainty. And that is also the case for the POLYNOMIAL NON-IDENTITIES problem. We have lots of 'Yes'-witnesses, meaning x - and y -values which proves that two polynomials are non-identical. But we don't know of any witnesses to the fact that they are not non-identical, or in other words, identical.

The beauty of this probabilistic approach is that it can in a way simulate non-determinism, under the assumption that the number of witnesses has a lower bound – when there are lots of witnesses. And this is the important insight. This is how we can understand what is good about randomized algorithms or randomized computation.

Notice that the probability that the Monte Carlo algorithm succeeds after $f(n)$ steps is *independent of the input*. This probability refers only to the outcome of the coin tosses. Which means that we can repeat the whole procedure for the same input, and independently of the first outcome have again the probability of $1/2$ of success.

And since the two probabilities are independent, they multiply. So if the probability of getting a wrong answer is $1/2$ or less in one iteration of the algorithm, the probability that we will have a wrong answer two times is $1/2 \cdot 1/2$. Three times is $(1/2)^3$. And the probability that after 100 repetitions we will still have a wrong answer, is less than or equal to 2^{-100} .

How big is this probability 2^{-100} ? What sort of number is it? It has been calculated that this probability is smaller than the probability of a meteorite hitting the computer while the algorithm is running. So there are two ways of looking at this. You can say: "Well, this is not certain." Because it is not certain. This algorithm can in fact never give you complete certainty. But you can also say: "Look, for all practical purposes we don't worry about such things as a meteorite hitting the computer while the program is running, or a hardware failure turning a '1' into a '0'. Why would we worry about the probability of failure which is 2^{-100} or less?"

I think this is a very valid sort of argument. So for practical purposes a Monte Carlo algorithm is a very good solution. We are talking about 100 repetitions. 100 is not a big number. We are not talking about anything exponential. 100 is a constant. It is one of those things that we typically neglect when we do algorithm analysis. So after only 100 repetitions you get this kind of error which is for all practical purposes negligible. It is a good solution.

IN210 – lecture 12



Def. 1 A **probabilistic Turing machine (PTM)** is a canonical NTM modified as follows:

- At every step computation the machine chooses one of the two possible transitions uniformly at random.
- The machine has two final states: q_Y and q_N .

Note: An algorithm on a PTM doesn't have to toss coins explicitly at every step, there can be implicit coin tosses with the two possible transitions being to the same state.

Def. 2 A PTM M is said to be a **Monte Carlo algorithm** for language L if there is a polynomial $f(n)$ such that for all inputs x M halts in $f(|x|)$ steps and

- if $x \notin L$ then M halts in q_N .
- if $x \in L$ then $P_r(M \text{ halts in } q_Y) \geq \frac{1}{2}$.

**Notes:**

- The probability $\frac{1}{2}$ can in principle be replaced by any fixed probability $\epsilon > 0$, because of the technique of repeated trials.

Autumn 1999

4 of 12

This is roughly the story of randomized computing. Now we formalize this a little bit, and I just want to show you in a couple of pages how exactly this randomized computation and randomized algorithms are formalized. And also what sort of theory emerges in relation to this. The details are in class IN394. In that graduate class we see some complexity classes. We study whether there are complete problems, what sort of problems are in these classes, and what sort of problems are solved by these approaches. The idea here is just to illustrate the approaches and stimulate your interest.

We have said that a canonical NTM is a NTM that always has exactly two possible transitions, and which halts after exactly $f(n)$ steps – if it halts. It is easy to see how any NTM that is time bounded, can be turned into a canonical TM simply by using some extra states and making sure that if you have 3 or 4 transitions possible from a certain state and a certain input, then you first make a transition to two invented auxiliary states, and then from those two to those real ones. So that you always has exactly two transitions.

This canonical NTM equipped with a coin turns into what is called a *Probabilistic Turing machine* (PTM). A PTM is not non-deterministic anymore, but it is not deterministic either. It is probabilistic in the sense that it can make choices based on random bits.

We can formalize as follows: We can define a PTM as being a canonical NTM modified so that in every step of computation the machine chooses one of the two possible transitions uniformly at random – that means with probability $1/2$. And then the machine has two final states q_Y and q_N . Intuitively this would

mean that it can answer 'Yes' or 'No'.

Notice that an algorithm on a PTM doesn't have to toss coins explicitly at every time step. There can be implicit coin tosses with the two possible transitions being to the same state.

Then we say that a PTM M is a Monte Carlo algorithm for language L if there is polynomial $f(n)$ such that for all inputs x , M halts in $f(|x|)$ steps. And if x is not in L , then M always halts in q_N . If x is in L , then M says 'Yes' (halts in y_Y) with probability at least $1/2$.

Some remarks: The probability $1/2$ can in principle be replaced by any fixed probability $\epsilon > 0$, because by doing repeated trials we can invent a new algorithm which has probability at least $1/2$.

IN210 – lecture 12

**Notes (continued):**

- The probability $\frac{1}{2}$ for choosing the right answer for positive instances must hold in the **worst case**, meaning even for the most tricky input.
- A Monte Carlo algorithm never gives incorrect positive answers, but it can give false NO-answers (“maybe NO”).
- Only languages that are in \mathcal{NP} or in $\text{Co-}\mathcal{NP}$ can have Monte Carlo algorithms.

Def. 3 A pair of Monte Carlo algorithms, one for language L and one for L^c , together compose a **Las Vegas algorithm** for L .

Notes:

- Only problems in $\mathcal{NP} \cap \text{Co-}\mathcal{NP}$ can have Las Vegas algorithms. Example: PRIMALITY.
- After 100 repetitions of a Las Vegas algorithm on the same input, we are “pretty sure” to get either a definite YES-answer (from the L -algorithm) or a definite NO-answer (from the L^c -algorithm).

Autumn 1999

5 of 12

An important point is that this approach to computation is still worst case. The probability $1/2$ for choosing the right answer for positive instances must hold in the worst case, meaning for the most tricky input.

Another point is that only the languages that are in \mathcal{NP} or $\text{Co-}\mathcal{NP}$ can have Monte Carlo algorithms. This is an interesting point. You cannot solve any other sort of problem in polynomial time with this kind of algorithm. This follows directly from the definition.

A Monte Carlo algorithm never gives incorrect positive answers. If it says ‘Yes’ then the instance is for sure a positive instance, because the algorithm has found a witness. But if it says ‘Maybe no’, then it can be lying, because it can happen that it just didn’t find a witness even though the input is a positive instance.

Can we get rid of this asymmetry in the definition of the Monte Carlo algorithm? This asymmetry can be somehow bothersome if you are solving some very important problem by Monte Carlo algorithm. And this happens because this kind of algorithms they are used very much in number-theoretic computing – in finding very large prime numbers or proving that a number is composite, and things like that. We will see next time why these sorts of computations are important, but I will just give you a hint now. They are used in cryptography for creating codes and for encoding messages and sending secret messages, or decoding secret or not so secret messages. And since a lot of money and also security of countries depend on the outcome of these algorithms, then you may actually want certainty. You may don’t want to live with this 2^{-100} kind of prob-

ability. The defense people and banking people are sometimes like that. They really want to be certain.

How can you get certainty from this kind of probabilistic approach without really going all the way to determinism? It turns out to be easy. You put two Monte Carlo algorithms together – one for the language L and another one for L^C – and you run them in parallel. You run one and then you run the other. So what happens? One of them can answer 'Yes' or 'Maybe No', another one can answer 'No' or 'Maybe Yes'.

So you can with probability $1/2$ always get a certain 'Yes' or a certain 'No' answer, and then with some smaller probability you get a 'Maybe' answer. But if you get a maybe answer, you repeat the whole thing. And again with probability $1/2$ you get a certain 'Yes' or a certain 'No'. So in due time, unless you are extremely unlucky, you do get a certain 'Yes' answer or a certain 'No' answer.

This sort of algorithm that in polynomial time produces either a certain 'Yes' or a certain 'No' answer or the answer 'Maybe' with a small constant probability, is called a *Las Vegas algorithm*. And Las Vegas is just about the best approach there is. But unfortunately, and this is the limitation, only problems that are in \mathcal{NP} intersection $\text{Co-}\mathcal{NP}$ can have Las Vegas algorithms. An example of such a question that is in \mathcal{NP} intersection $\text{Co-}\mathcal{NP}$ is primality testing – testing for prime numbers. This is explained in more details in IN394.

IN210 – lecture 12



Randomized complexity classes

\mathcal{RP} (randomized \mathcal{P})

A language belongs to class \mathcal{RP} if it has a Monte Carlo algorithm.

\mathcal{ZPP} (zero error probability \mathcal{P})

A language belongs to class \mathcal{ZPP} if it has a Las Vegas algorithm.

\mathcal{BPP} (bounded probability of error \mathcal{P})

A language L belongs to class \mathcal{BPP} if it is recognized in polynomial time by a PTM M such that

- $x \in L \Rightarrow P_r(M \text{ answers 'YES'}) \geq \frac{2}{3}$
- $x \notin L \Rightarrow P_r(M \text{ answers 'NO'}) \geq \frac{2}{3}$

Autumn 1999

6 of 12

I will just mention some randomized complexity classes. First of all we have \mathcal{RP} – Randomized \mathcal{P} . This is a natural class. We say that a language belongs to the class \mathcal{RP} , randomized \mathcal{P} , if it has a Monte Carlo algorithm.

A language belongs to \mathcal{ZPP} (Zero error Probability \mathcal{P}) if it has a Las Vegas algorithm. It is named zero error probability because ultimately, after enough repetitions, algorithms that put a language into \mathcal{ZPP} give the definite answer 'Yes' or 'No'. So we don't have to live with these errors.

A \mathcal{ZPP} algorithm gives you a definite 'Yes' or 'No' answer with a probability which is exponentially high in the number of repeated trials. On the other hand a \mathcal{RP} algorithm gives you definite 'Yes' answers, but it doesn't give definite 'No' answers, only 'No' answers that is correct with exponentially high probability in the number of repeated trials. So the subtle difference is that with a \mathcal{ZPP} algorithm you will, with a very high probability, know that your answer is the correct answer. While with a \mathcal{RP} algorithm you will get a correct 'No' answer with a very high probability, but you can never be absolutely sure that it is the correct answer.

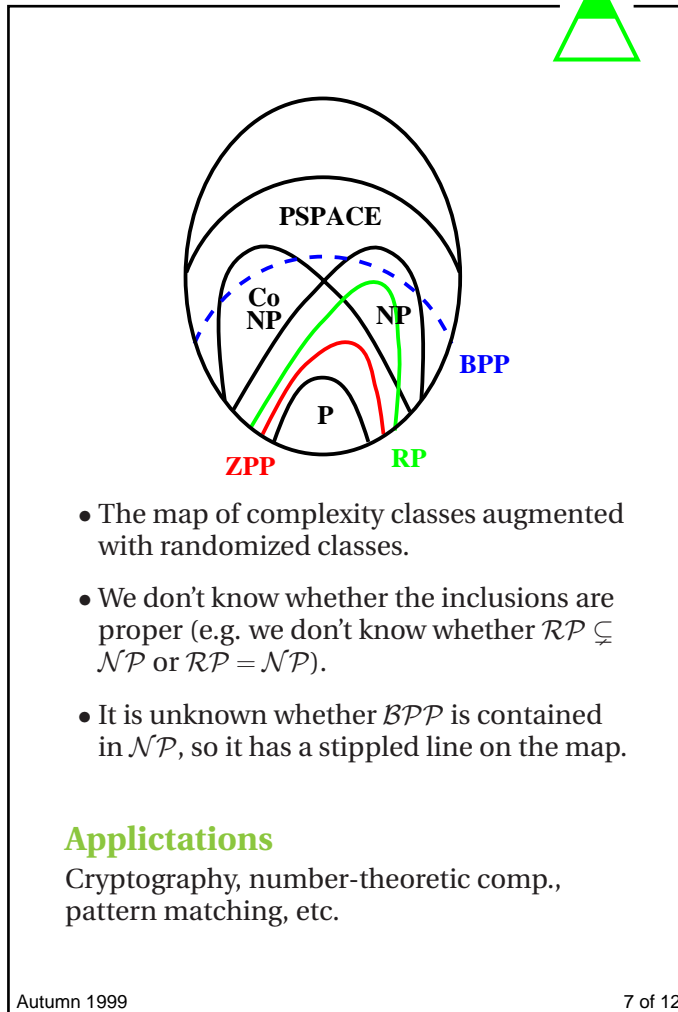
And then there is a third class which is a little bit broader and also very natural: \mathcal{BPP} , or Bounded Probability of error \mathcal{P} . A language L belongs to the class \mathcal{BPP} if it is recognized in polynomial time by a PTM M such that 1) if x is in L then the probability that M answers 'Yes' is greater than or equal $2/3$, and 2) if x is not in L then the probability that M answers 'No' is also greater than or equal $2/3$.

This $2/3$ is an arbitrarily constant. In fact any constant strictly greater than

$1/2$ will do, because then we can repeat the algorithm many times, and the probability of error will drop exponentially.

If you want this constant to be $99/100$ instead of $2/3$ but $99/100$, a constant number of repetitions of an algorithm that has probability $2/3$ is enough. The proof of this fact is a little bit more involved. It involves a certain kind of mathematics which we don't know here. It is not difficult, but we don't really have time to go into that. Again that is taken up in IN394.

IN210 – lecture 12



Here I have drawn our map of complexity classes and augmented it with the three randomized classes. The situation is like this: We know that \mathcal{RP} is definitely contained in \mathcal{NP} , and that \mathcal{ZPP} is definitely contained in $\text{Co-}\mathcal{NP}$ intersection \mathcal{NP} . So there are obvious limits to the complexity of problems that we can solve with these approaches. It is for example impossible to solve a PSPACE-complete problem in polynomial time by using a Monte Carlo algorithm.

We don't know whether the inclusions are proper. For example, we don't know whether \mathcal{RP} is a proper subset of \mathcal{NP} or if \mathcal{RP} happens to be equal to \mathcal{NP} . In other words: We have no answer to the question "Is there a problem in \mathcal{NP} which cannot be solved by a Monte Carlo algorithm?" This implies that we haven't yet found a Monte Carlo algorithm for a \mathcal{NP} -complete problem L , because if we have such an algorithm, then all problems in \mathcal{NP} could be solved by Monte Carlo algorithms, by virtue of L being \mathcal{NP} -complete.

It is also unknown whether \mathcal{BPP} is contained in \mathcal{NP} , so it has a stippled line on the map. But we do know that \mathcal{BPP} is contained in PSPACE, because we can only use a polynomial amount of space in polynomial time.

Finally just a word about applications of this. One very important application of these approaches is cryptography and number-theoretic computing. The reason being that PRIMALITY, testing whether a number is prime, belongs to the intersection of \mathcal{NP} and $\text{Co-}\mathcal{NP}$. PRIMALITY has very important applications in cryptography and all sorts of things. It has been shown that PRIMALITY is in fact in \mathcal{ZPP} .

Similar kinds of ideas can be used for pattern matching. Assume that you

have a big text and you are in a text editor and you are saying: "Search whether the word polyscope is in the text." The text editor can compare character by character, or it can do some kind of randomized test along the lines which I have shown you. And those randomized tests they turn out to be much more efficient. So if you have this kind of problem where you are searching for a string within a larger string – checking whether this string is a substring of the other string, which is something that text editors do all the time – then you would be advised to use the randomized approach because it is much more efficient.

IN210 – lecture 12



Example: Combining randomizing and average-case analysis

A “random walk” algorithm for HAMILTONICITY with good average-case performance:

- **Idea:** Try to construct a Hamiltonian path by adding nodes at random:
 - Pick one of the neighbors, x , of the LAST-node at random.
 - If x is not in the path, add x as the new LAST-node.
 - If x is already in the path, change the path in the following way:



- The algorithm is very simple, but the analysis is not.
- Algorithm is based upon **coupon collector** pattern:
 - How many coupons does it take on average to collect n different ones?

$$\mathcal{O}(n), \mathcal{O}(n \log n), \mathcal{O}(n^2), \mathcal{O}(2^n)$$

Autumn 1999

8 of 12

As an example of the probabilistic approach to algorithm design, I am now going to briefly illustrate an efficient algorithm for HAMILTONICITY. The algorithm uses a probabilistic technique called *random walk* to achieve a good average-case performance.

HAMILTONICITY is an \mathcal{NP} -complete problem and therefore intractable in the worst-case paradigm, but last time we have seen a three-phase algorithm for HAMILTONICITY which runs in linear expected time on a random graph with edge probability $1/2$. That algorithm always produced an answer, even when the graph didn't have a Hamiltonian path.

The random-walk algorithm on the contrary will only produce positive answers but it will do it fast – meaning that it will find a Hamiltonian path very fast with high probability, but it will never succeed when the graph is non-Hamiltonian. In practice you might want to combine this random-walk algorithm with some other approach, so that you also eventually get an answer if the graph is not Hamiltonian.

How does the random-walk algorithm work? You start from a certain vertex, call it vertex 1, and then we look at its neighbors and choose one neighbor at random by tossing a coin. Say that's vertex number 4. And then from that vertex number 4 we try to extend our Hamiltonian path further by tossing a coin again. And say we get vertex number 17, and so on.

So in the general case we have the last vertex assembled, which I have labeled 'LAST' on the foil, and then from this last vertex we toss a coin, picking one of its neighbors at random, and then that one becomes the next last. If the

new vertex happens to be a complete new vertex, then we are fine – we just extend the path. The problems arise if the neighbor happens to be already in the path.

If we pick the vertex y which happens to be already in the path, then the algorithm does something strange: It is going to make y the new LAST vertex in the path by disconnecting the edge between x and y , make a new edge from x to the old LAST, and then reverse the path from y to the old LAST. So this red marked path becomes the new Hamiltonian subpath, and then from the new LAST vertex we toss a coin, trying to find one of its neighbors.

The reason for changing the last vertex, is that we want to give this new vertex y and its neighbors a change. This simple algorithm will for a large enough p , the edge probability, find a Hamiltonian path in very short time almost certainly. Almost certainly means with probability 1, asymptotically speaking.

The way this algorithm is constructed is based upon the coupon collector pattern. And my idea here is this: Instead of showing you the exact analysis of this algorithm, which maybe somewhat complicated and somewhat confusing to you, I am going to tell you how to think about this kind of random-walk algorithm in general, by using a kind of a pattern – a simple model which just models the basic thinking style that is behind the algorithm and its analysis.

So here is that model: What we are doing in this algorithm, is that we start from a vertex, and we pick another vertex at random and then another vertex at random and another vertex at random. And maybe the next vertex is going to be one that is already picked.

In the beginning of time, when we just have vertex 1 and we pick anyone of its neighbors, then of course we get a new guy. But if we have already just about all of the vertices assembled, then the probability of picking a new guy is rather small, and the probability of picking an old guy is rather big. So that when we have $n - 1$ vertices in our pocket, the probability of picking the n 'th guy is low – it is just about $1/n$.

The question is how many times do we have to run this procedure, how many times do we have to pick a vertex at random in this way, before we have assembled all n of them? And this situation is modeled by the coupon collector pattern where you assume that you are a little kid with an album where you glue pictures. And you get those pictures from little chocolates – this is what chocolate people like to do in order to make kids buy as many chocolates as possible.

So in each chocolate you would find a picture, and you have space for n different pictures in the album. The question is: How many chocolates do you have to buy before you can have all n of them?

Say there are $n = 1000$ different pictures. What should be the expected number of chocolates that you need to buy in order to get all n of those? Is it $\mathcal{O}(n)$, $\mathcal{O}(n \log n)$, $\mathcal{O}(n^2)$ or $\mathcal{O}(2^n)$?

What about the distribution, you might ask. And of course you cannot trust the chocolate sellers, but if they were fair and nice people, then all the chocolates would have the same probability. Now, sometimes they may just keep some few pictures kind of non-existent or very difficult to get a hold of. Because when you complete the album you send it to the factory and you get a box full of chocolates. It is pretty expensive to send people boxes of chocolates. It is much better to make them buy more chocolates because they are just missing 3 pictures, but those 3 pictures they don't exist at all.

But suppose that these people they are fair. If they are fair it turns out that $\mathcal{O}(n \log n)$ is the number of chocolates you have to buy. And I don't want to bother you with the details in this class, but this is what comes out of the analysis. This algorithm and its analysis is studied in more detail in the algorithm

design class – IN391.

So the expected number of steps for this algorithm to construct a path is $\mathcal{O}(n \log n)$. But what about NON-HAMILTONICITY? What happens if the algorithm does not construct the Hamiltonian path, and what happens if the Hamiltonian path does not exist at all? Well, it turns out that this algorithm that I have just described, it cannot really deal with NON-HAMILTONICITY. It can only construct a Hamiltonian path if it exists, or raise its arms and say: "I am sorry, I am just having a bad day. I don't know how to deal with this graph."

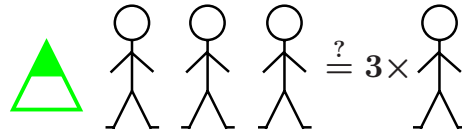
As we have talked about before, NON-HAMILTONICITY is actually quite a bit more difficult than HAMILTONICITY, because it is a co- \mathcal{NP} complete problem and those problems don't seem to have short IDs or proofs of membership. So to prove that a graph is non-Hamiltonian you basically have to try all possible sequences of vertices, which means exhaustive search.

So this algorithm that we have seen, it does not really produce a proof of NON-HAMILTONICITY. All it does is, it either constructs a Hamiltonian path or it fails. But if we assume a constant edge probability, say $p = 1/2$, then the 0-1 law says that any graph almost certainly has a Hamiltonian path. And then this algorithm is successful with a large probability – large probability meaning with probability asymptotically 1. So that's the kind of statement that you can make about this algorithm: We run the algorithm for a polynomial number of steps, and it finds a Hamiltonian path with probability asymptotically 1.

(This is a blank page)

IN210 – lecture 12

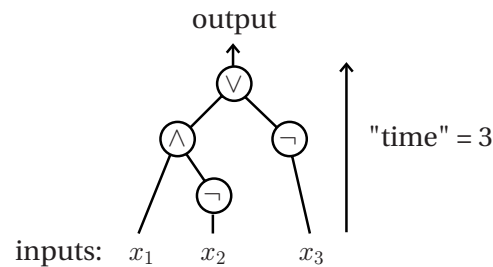
Parallel computing



- some problems can be efficiently parallelized
- some problems seems inherently sequential

Parallel machine models

- Alternating TMs
- Boolean Circuits



— Boolean Circuit complexity: **"time"** (length of longest directed path) and **hardware** (# of gates)

Autumn 1999

9 of 12

There is a very well known book, kind of a classic in software engineering and one of those books that applied people in computer science like to read. It's called "The Mythical Man-Month" and it was written by Frederick Brooks back in 1975. There is a message that comes from that book, and I will introduce parallel computing by using that message. The title of the book says "The Mythical Man-Month", and the point is that the effort involved in a software project was typically measured in man-month, meaning that many people, that many months. So according to that idea if one person would finish a software project in 10 months, then 10 people would finish it in 1 month.

Brooks was saying in the book that this is in fact not at all the case. He was experiencing lots of situations where some number of people would be working on a project. And then you bring more people to the project, trying to speed it up, but in reality the project just slows down by bringing more people. So bringing more people doesn't necessarily mean efficiency.

Parallel computing is the kind of computing where computation happens in parallel in different places, e.g. on different processors or on different machines. This is roughly similar to having a number of people working on a project. The question is: Can we speed up the computation drastically by bringing more hardware, more things, so that the computation happens in different places in parallel? It is a very natural idea and there has been a lot of talk about parallel computing, actually since the year 1960. Since the time where the limitations of sequential computing became apparent to people, they were looking at parallel computing as the solution to all sorts of issues, especially to complexity issues.

According to "The Mythical Man-Month"'s idea there will be two kinds of situations in this world, two kinds of problems: One kind that can be efficiently parallelized, meaning that bringing more people will speed up the solution. Another possibility is that the problem or the situation is inherently sequential, meaning that no matter how many people or how much hardware or how many processors you bring, you still has to do this step by step – it doesn't help to bring more people on the project. You can think of real-life projects that have that quality and you can think of real-life projects that can be efficiently parallelized.

In the world of parallel computing there are obviously parallel machines which run parallel algorithms, but the situation is very different from the world of sequential computing. While in sequential computing the von Neumann-machine is predominant there are all kinds of parallel computers in this world.

So without going into details, it is a tremendous variety, and this provides some obvious problems when we want to come up with good abstractions or with good formal models of parallel machines. In practice three kinds of models are used. One of them is not very widely used. I mention it only because it is a straightforward generalization of the approach we have had so far, which is the Turing machine approach.

So there is a kind of a TM that is used in parallel computing, which is called the *Alternating Turing machine* (ATM). And let me just very briefly mention to you what this is. An ATM would be a generalization along the lines in which a NTM is a generalization of a deterministic TM. A deterministic TM is just the stupid, ordinary deterministic machine which just does a sequence of computations as it is told, without any sort of fancy tricks. A NTM on the other hand does something really fancy. It can check efficiently whether there exists a witness. So it can guess. And this is why a NTM has an easy time with \mathcal{NP} kind of problems, while it cannot do anything with $\text{Co-}\mathcal{NP}$ kind of problems. But if it could answer this kind of questions: "Is it True that for all witnesses something holds?", then it could also deal with $\text{Co-}\mathcal{NP}$ kind of problems. For example a NTM can easily find a witness that proves that two polynomials are non-identical. But it has to answer the question "Is it true that for all values of x and y , the two polynomials evaluates to the same value?" in order to deal with the polynomial identity problem which is in $\text{Co-}\mathcal{NP}$.

An alternating TM is a generalization of a NTM along this approach – alternating between existential and universal quantification. In one step an ATM can act as an ordinary NTM and give an answer whether there exists a witness. Then in the second step it can check whether for all witnesses something holds, and so on.

In a certain sense this generalization of the TM makes it parallel, because this checking whether something holds for all witnesses is something that you could do in parallel on many machines. So this is roughly a kind of a parallel TM model.

The problem with it is that it is highly unrealistic. This is not something that you can actually build. It is very much a mathematical construct, and it is only used in very limited circumstances.

Next we will talk about a very realistic parallel machine model which is called *Boolean Circuits*. This is actually in a way the most realistic machine model in existence, because every digital computer today is by and large a Boolean circuit. A CPU is a circuit – a kind of bunch of wires and gates – which computes a certain function. And a Boolean circuit is just that.

Formally a Boolean circuit is a directed acyclic graph in which the nodes have labels, and those labels are Boolean operators. So here on the foil we see a Boolean circuit that computes a certain function. It has input nodes on the

bottom – x_1 , x_2 and x_3 . And then what happens with x_2 is that it gets negated when passing through the negation gate. And following that we have an AND gate. So in this AND gate we have $x_1 \wedge \neg x_2$ computed. And that result comes into this OR gate together with the negated x_3 . The OR gate is computing the final output function $x_1 \wedge \neg x_2 \vee \neg x_3$.

This is a logical function but as you should know already from other classes (IN147), anything that a computer can do – all computations with numbers and things – can be represented in this way as Boolean expressions. So Boolean circuits can compute anything that any sort of machine can compute.

And there is parallelism. The negation of x_2 and the negation of x_3 can happen in parallel, but the AND operation must wait until the negation of x_2 is finished. And then the OR gate has to wait until but the AND and the negation of x_3 has happened.

We have a natural notion of time as a complexity measure in a Boolean circuit, namely the length of the longest directed path in the graph. In this case that length would be 3, so we can say that this Boolean circuit computes this Boolean function in 3 steps. There is another highly relevant complexity measure in parallel computation, and that is what is roughly called "hardware" or sometimes "work". And this is the man-month measure. Here the hardware is expressed as the number of gates. If we have 4 gates, then we say that hardware equals 4. We have 4 people working on the project.

And then the question would be: What is the shortest time for computing this function, supposing that hardware is cheap? Or if you really care about hardware, the question is: What is the minimum time of computing this function with the hardware we have?

Hardware and time they are obviously related. If you have some limits on hardware, then how much time we can save is also limited. But then also there are interesting questions of the type: If we really don't care about hardware – people are cheap, computers are cheap – what is the shortest time we can solve the problem? That is a kind of question that is studied in parallel computing.

One very important issue here, which I want to mention to you, is the issue whether a Boolean circuit is actually an algorithm or not. And the issue translates into what is called *uniformity*. What is really the problem? The problem is that our standard notion of an algorithm is something that can fit on a piece of paper. Maybe not a single piece of paper, maybe on five pieces of paper, but the main point of an algorithm is that an algorithm is finite. It is a finite description of something – it is a recipe.

You open a book of recipes. Then you can figure out how to make a bread, how to make this or that. In computer science you find recipes for dealing with problems instances of arbitrarily size, but the recipes themselves must be finite – something that can fit into a book. But these circuits they seem to grow with the size of the input, because this is really what these algorithms are like: If you have lots of input variables, then you have lots of these gates.

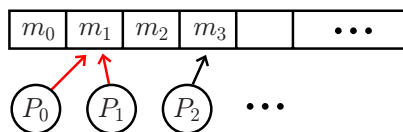
If a circuit can be arbitrarily large, can it then be called an algorithm at all? And we say 'Yes' if the description of the circuit itself is finite! You might say: "Wait a minute. How do you define this? How do you formalize this notion of 'description of the circuit is finite'?" Well, we say that the description of the circuit is finite if the circuit can be generated by an algorithm or a TM that has a certain complexity. This is a very interesting idea – the idea that the notion of a 'finite description' is modeled by TM in a natural way. It is a very, very fundamental notion. You should think about this.

The problem with Boolean circuits in practice is that they are extremely difficult to program. Think of solving the TSP with this sort of device. It is months of work and nobody could understand it. So Boolean circuits are by and large

for proving lower bounds, for proving that something has to use at least that much time. But for actually showing programs they are pretty hopeless and pretty useless.

IN210 – lecture 12

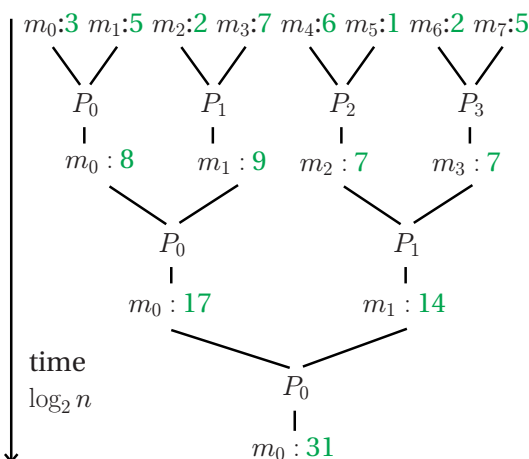
- **Parallell Random Access Machines (PRAMs)**



- Read/Write conflict resolution strategy
- PRAM complexity: **time** (# of steps) and **hardware** (# of processors)

Example: Parallel summation in time

$\mathcal{O}(\log n)$



Result: Boolean Circuit complexity = PRAM complexity.

Autumn 1999

10 of 12

So instead of Boolean Circuits, the *Parallel Random Access Machine* (PRAM) is used as a kind of a high-level-language construct in parallel computing. A PRAM is a straightforward generalization of a RAM. It is a parallelization of a RAM, where a Random Access Machine is the theoretical model that corresponds to our standard von Neumann machine. If the standard RAM has a memory and a processor with a certain instruction set such as LOAD, ADD, SUB and thinks like that, then a PRAM would have the same except that it would have multiple processors that can work in parallel.

So a PRAM has a memory consisting of memory cells, each able to contain some binary number. And it also has some number of processors that can operate on this *shared memory* in parallel. In every step of computation each processor can read an arbitrarily memory cell, perform some computation based on the contents of that cell and its internal registers and state, and then load the result into an arbitrarily memory cell.

Obviously there are some problems here – read and write conflicts can occur. If two processors attempt to write into memory cell M_1 in the same time step, then the question is: What is really written in M_1 ? And this question is handled by the so-called *conflict resolution strategy*. Different conflict resolution policies give rise to different variants of this model. This is not so interesting or important, so I am not going to go into more details than that.

PRAM complexity is naturally measured by the amount of time (number of steps) and hardware (number of processors) used.

I will now show you an example of a situation where this kind of approach

leads to a dramatic speedup in computation time, a dramatic improvement. Think of adding n numbers. We have numbers 3, 5, 2, 7, 6, 1, 2 and 5. These are stored in the first eight memory cells. To add these numbers in parallel you would use four processors. The first processor P_0 is adding the first two numbers and computing an 8, and then storing this 8 maybe into the first memory cell. P_1 would compute 2 plus 7, resulting in a 9, and then put the result into M_1 . And then the third processor will compute 6 plus 1, and store result into M_2 , and finally processor P_3 would add 2 and 5 and store a 7 into M_3 .

So this is two steps of computation – one step for reading the first input element, and one more step for reading the second number and computing and storing the result. In two steps of computation the number of input elements will effectively be reduced by a factor 2.

And then in the third step of computation the number of input elements will again be reduced by a factor 2. M_0 will already have in its internal register the sum it computed in the first two steps, so it would just need to read this 9 stored in M_1 , compute 9 plus 8 which is 17, and store the result maybe in M_0 again. At the same time P_1 would add 7 and 7 together and compute 14. And then all that remains in the fourth step is P_0 adding 17 and 14, obtaining the final result which is something like 31.

Obviously the time that this computation takes is the height of this tree. And this computation tree looks like a binary tree. The height of this tree is in fact the logarithm base 2 of the number of inputs. We know that the fastest possible way of solving in the worst case this or any other reasonable problem sequentially, is n . A sequential TM needs n steps even to read the input. But a parallel algorithm, a PRAM, can solve the problem in logarithmic number of steps – $\mathcal{O}(\log n)$.

So this is an exponential speedup. This is good news. We can speed up some computations dramatically. The question is: What about intractable problems? Can we solve, say, TSP in polynomial time by using parallel computing? And we will come back to this in a second.

Let me just say that these two machine models, as different as they may look, they are in fact shown to be equivalent. We have a result which says that the complexity measured using PRAMs and the complexity measured using Boolean circuits, they evaluate to roughly the same thing. To prove such a result one should prove that a PRAM can simulate a Boolean Circuit efficiently, and vice versa.

It is not surprising that a PRAM can do fast anything that a Boolean circuit can do fast. You just use processors to do this simple Boolean operations, and if you are allowed to have as many processors as there are gates in the Boolean Circuits, then that's easy.

So simulating a Boolean circuit by using a PRAM is easy. The question is: Can we simulate efficiently a PRAM with a Boolean circuit? The answer is 'Yes'. How do we prove this? We use our old friend, the computation matrix. I will not get into this, but the idea is roughly to use the computation matrix, and just about the same idea that we have seen in this class already, which is computing the next row of this matrix from the previous row as input, using logical gates.

IN210 – lecture 12



Limitations to parallel computing

Good news

parallel time \leftrightarrow sequential space

Example: HAMILTONICITY can easily be solved in parallel polynomial time:

- On a graph with n nodes there are at most $n!$ possible Hamiltonian paths.
- Use $n!$ processors and let each of them check 1 possible solution in polynomial time.
- Compute the the OR of the answers in parallel time $\mathcal{O}(\log(n!)) = \mathcal{O}(n)$.

Bad news

Theorem 1 *With polynomial many processors*
parallel poly. time = sequential poly. time

Proof:

- 1 processor can simulate one step of m processors in sequential time
 $t_1(m) = \mathcal{O}(m^k)$, for a constant k
- Let $t_2(n)$ be the polynomial parallel time of the computation. If m is polynomial then
 $t_1(m) \cdot t_2(n) = \text{polynomial}$.

Autumn 1999

11 of 12

The next question and the most interesting question for us, is whether we can solve some intractable problems efficiently in parallel. There is good news and bad news – there is an answer 'Yes' and an answer 'No' here. The answer 'Yes' says that roughly speaking parallel time corresponds to sequential space. There is a theorem which says approximately that. Let me give you an idea of why this is so:

Can we solve HAMILTONICITY in parallel polynomial time? Yes, we can! Say you have a graph. And then in this graph there is some number of orderings of nodes, which are the potential Hamiltonian cycles. We know that we have $n!$ possible orderings of n nodes. Our parallel algorithm will use $n!$ processors, each responsible for checking one of the candidate solutions.

Obviously this work can be done in polynomial time. All a processor needs to check is whether there is an edge between each of the nodes in the ordering, and then an edge between the first and the last node. And if this happens to be so, then the processor will say 'Yes', otherwise he will say 'No'.

Then it is just a matter of bringing the information from all of these $n!$ processors together. And that bringing of information together will amount to computing an OR function of these $n!$ partial results. And since an OR function can be computed in parallel logarithmic time, that will turn this also into a polynomial. So this whole algorithm is parallel polynomial time algorithm for HAMILTONICITY.

This is good news. The good news is that we can solve, by using the same kind of idea, just about any problem in \mathcal{NP} efficiently in parallel.

What is the problem with this? The problem is that we are using too much hardware! We are using $n!$ processors, and that is even more unrealistic than using $n!$ steps of computation. Imagine $n!$ processors – a number of processors that grows exponentially with the size of the input, and which even for moderately small n values exceeds the number of available molecules in the whole universe. That is very unrealistic.

The interesting question is what we can do with polynomially many processors. The bad news is that with polynomially many processors, \mathcal{P} remains robust, meaning that parallel polynomial time is the same as sequential polynomial time. The proof is trivial by simulation:

If I have polynomially many processors, say m of them, then I can simulate what these m processor can do in parallel by using 1 single processor. This is how I do it: My single processor will first run the first processor for a while, then run the second one, then the third one, and so on. This is similar to how a job scheduler in a multi-task operating system operates.

So my single processor can simulate one parallel step of m processors by using $t_1(m)$ sequential time, where $t_1(n) = \mathcal{O}(m^k)$ for some constant k , meaning that t_1 is a polynomial in m . And if this m processors parallel machine works in parallel polynomial time $t_2(n)$, then the single processor would need $t_1(m)f_2(n)$ sequential time. And given that these two guys are polynomials, then the overall sequential time is also polynomial.

So if a parallel machine with a polynomial number of processors solves a problem in polynomial time, then a sequential machine with a single processor can solve the same problem in polynomial time.

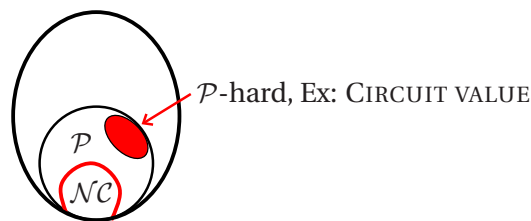
This means that speeding up intractable problems turns out to be hopeless with parallelism, at least from a theoretical viewpoint. In practice even a constant speedup of a HAMILTONICITY-solving program or a chessplaying program might be interesting. So most of theoretical parallel computing focuses on speeding up problems that can be really speeded up, and those are some of the problems in \mathcal{P} .

IN210 – lecture 12



Parallel complexity classes

Def. 4 A language is said to be in class \mathcal{NC} if it is recognized in polylogarithmic, $\mathcal{O}(\log^k(n))$, parallel time with polynomial hardware.



• $\mathcal{P} \stackrel{?}{=} \mathcal{NC}$

Autumn 1999

12 of 12

Most of the theory of parallel computing is focused on the issue of \mathcal{NC} versus \mathcal{P} . By definition a language is said to be in class \mathcal{NC} if it is recognized in polylogarithmic parallel time with polynomial hardware. And here polylogarithmic means function in time which is $\mathcal{O}(\log^k(n))$. This is a straightforward generalization of the idea of efficiently sequentially computable, meaning computable in polynomial time.

I should mention that here we are again talking about uniform models if we are talking about Boolean circuits.

So the class \mathcal{NC} involves those problems that are solvable in sequential polynomial time, and that can be efficiently parallelized. Efficiently means that bringing polynomially more hardware results in a dramatic, exponential, drop in computation time. In contrast to the problems that are in \mathcal{NC} , there are problems which are shown to be \mathcal{P} -hard, or hard for class \mathcal{P} , and those are believed to be not parallelizable efficiently.

The situation here is exactly analogous to the situation between \mathcal{P} and \mathcal{NP} . If one of the \mathcal{P} -complete problems turns out to be in \mathcal{NC} , then all of \mathcal{P} turns out to be in \mathcal{NC} . But whether or not $\mathcal{P}=\mathcal{NC}$, that is again not known.

So to show that a problem is not efficiently parallelizable, amounts to showing that the problem is \mathcal{P} -complete or \mathcal{P} -hard. An example of a \mathcal{P} -hard problem is the Boolean circuit value problem, where we are given a description of a Boolean circuit and the values of the inputs, and the question is to compute the value of the output. This problem seems to be inherently sequential.

This is all I wanted to say about randomized and parallel computing. Next

time, in our last lecture, I will first talk about a new computer model which challenges the Computational Complexity Thesis. Then I will switch to certain big aspects of computation, namely cryptography and language design, which are not directly part of complexity analysis, but where complexity theory really throws a lot of lights on the essential issues.

3.13 Lecture 13

IN210 – lecture 13



Alternative machine models

Computational complexity thesis: All reasonable computer models can simulate one another in polynomial time (i.e. \mathcal{P} is “robust” or “machine independent”).

But the Turing machine is based on a **classical physics model of the universe**, whereas current physical theory asserts that the universe is **quantum physical!**

Question: Can we build more powerful computing devices based on quantum physics?

(Another interesting kind of computing device is the **biological computer** . . .)

Algo:

**11 (323-342),
12 (347-353)**

Autumn 1999

1 of 9

This is our last lecture, so the natural question to start with is: What have we been doing here in this class? Well, we have constructed a map of classes, and we have used this map to organize our world – to organize our thinking about informatics, about computers, about problems, and about their solutions in terms of algorithms.

We first focused on the borderline between those problems that are solvable by algorithms and those that aren't. We learned different techniques – diagonalization and reduction in particular – for proving that a problem is unsolvable, and we got insights about how those unsolvable problems look like. Then we started to focus on the solvable problems and we made a distinction between those problems that are properly solvable – meaning solvable in polynomial time – and those that seem inherently intractable. We identified intractable problems with the class \mathcal{NP} -complete, and we used quite a few lectures to learn how to prove that a new problem is \mathcal{NP} -complete by reducing in polynomial time another \mathcal{NP} -complete problem to our new problem.

So we have constructed a complexity theory which says something about how difficult different classes of problems are to solve. This complexity theory is based on the Turing machine as a model of an algorithm, and it is also based on a certain approach which we called worst case & best solution. And this approach is pessimistic. It can be criticized.

We have been looking in the previous lectures at some alternative ways of

defining and analyzing algorithms, which has led to certain design paradigms which we could analyze and then place on the map too. We have seen approximation algorithms, average-case analysis, randomized computing and parallel computing. We have tried to grasp the possibilities and limitations of those approaches, and especially how suited they are for dealing with those problems that are intractable in the worst-case & best-solution paradigm – the \mathcal{NP} -hard problems.

Today we will start by finishing this part on alternative approaches. We will take a critical look at the backbone of our theory – the Turing machine as a formal model of a computer. This whole complexity theory that we have studied is based on TMs. What if we make a completely different kind of computer that is nothing like a Turing machine? Can we then solve some of those intractable problems efficiently?

And to begin with it seems like that won't really work, because we have the Computational Complexity thesis (CCT), which says that all the *reasonable* computers – all the digital computers that we have today including parallel computers with polynomially many processors – they can simulate one another in polynomial time. In other words we say that \mathcal{P} is robust or machine independent.

But you must remember that CCT is only a thesis, it is not a theorem which can be proven. So the fact that we haven't been able to construct a significant more powerful computing device yet, doesn't mean that the Turing machine is the best we can get. Maybe it just reflects our lack of knowledge and skills.

So the question is a valid one: What about making some completely different kind of machine that is nothing like the classical deterministic TM? Can we solve some of the intractable problems, some of the \mathcal{NP} -hard or \mathcal{NP} -complete problems, on that kind of machine? And what should such a machine be like?

A lot of people have been working along these lines, and some ideas have emerged, such as the quantum computers, and then there are also biological computers and all kinds of other computers. And every once in a while one finds an issue of a scientific journal or a computer science journal that is dedicated to one such special kind of computation.

We will now take a look at the quantum computer. We know that the TM is based on the classical physics model of the universe, which is described by Newtonian mechanics. But today we don't believe that the world is a large mechanism. Instead we believe that the universe is quantum physical. So the natural question is whether we can build more powerful computing devices based on quantum physics instead of classical physics. And we will see that the answer is a 'Yes' and a 'No' – as usual.

IN210 – lecture 13



Quantum Computers

or **outsmarting complexity**

- According to quantum physics a particle (electron, proton, etc) can be in several different quantum states **at the same time**.
- A quantum computer can follow several different paths in the computation tree at the same time, and therefore somehow act as a NTM.
- Several quantum machine models have been proposed, e.g. a **quantum Turing machine**.
- In 1994 Peter W. Shor showed a **polynomial time** quantum algorithm for FACTORING and DISCRETE LOG, two problems that seem to be difficult on a classical TM, and whose intractability modern cryptography relies upon.

Autumn 1999

2 of 9

What is a quantum computer? I cannot really tell you in detail how it works, because it involves some interesting physics, but I can tell you a little story about some principles behind it. The story is about one of the most interesting things that have happened in our time. It is about what has been called the *paradigm change in science*

In quantum physics people have discovered that small particles actually defy our normal logic. What is "the normal logic"? It is the true/false logic that all branches of science have been using, including computer science. But that logic actually was formulated by the ancient Greek Aristoteles and it is a very old thing. The logic for example says that a statement A and its negation cannot be simultaneously true.

That logic was assumed to be first God-given, and then somehow to be natural or to be obvious without doubt. So in various ways this logic was established or postulated, but always it was assumed to be right. Until people proved experimentally that small particles – those guys that are invisible by eye, and that all those clever people in the past didn't know about – that they actually just simply defy the classical logic.

An electron can in fact be in two states simultaneously. It can go through two holes simultaneously, and how this can happen, nobody can quite understand. There is an interesting paper written by Heisenberg, one of the famous physicists, which is called "Physics and philosophy". It is a kind of a monograph on quantum theory and on consequences of it – just completely practical cultural consequences.

In that book Heisenberg is talking about an interesting situation of physicists talking to one another, and they are talking about those electrons and small particles. The problem is that they cannot even use the normal language, be-

cause what they are observing cannot really be described in the normal language, because our language is based on a certain logic. And that logic just doesn't apply there.

So the language doesn't work. Instead they invent their own language, a kind of descriptive language which more or less works. One thing works and that is mathematics. Because mathematics is a design kind of language, so you can design exactly the kind of mathematics – not of course this old logic – but you design a logic that works, the equations, and you can compute the consequences. But you cannot really describe those consequences in the normal language, and mentally you cannot grasp them.

There are these two things. One is what we can grasp mentally and what our normal language can describe. The other is experimental physics and its mathematical models. And those two worlds, they are two disjunct worlds.

Why is this so interesting? It is interesting for many reasons, but one of the reasons is that it was the death of metaphysical thinking. For centuries we had been believing that we could with our minds grasp the reality out there, and that our logic is *the* logic. Now it turns out that what is out there is much deeper and much more complex and much more full of surprises, then what our logic can grasp and anticipate, and then what our language can express.

So we are left doubting, but we are also living in a world which is full of unbelievable possibilities. I am currently working in an emerging field called *information design*, and information design actually begins from there. I will not say more now, but maybe some of you will take our new class in information design, which is normally taught in the spring, and then we can talk more about these things.

There are possibilities opened by this new kind of logic and physics for a new kind of computer. And very roughly speaking this new kind of computer – *the quantum computer* – exploits this possibility of an electron being in two states or going through two holes at the same time. So you can think of actually some physical device which explores not one possibility at a time, but two possibilities. And then again two possibilities, and so on. So something that can more or less do the non-deterministic computation in a real sense.

It is difficult to follow these images with our mind, but the point that I am trying to make here is that nature has all sorts of wonderful possibilities for us. One of them is a completely different kind of computation, which uses a different kind of logic, and which can solve intractable problems in polynomial time.

Several quantum machine models have been proposed. One of them is the *quantum Turing machine (QTM)*. The QTM is almost like a normal TM, except that in each time step it is allowed to do one extra operation on one single bit.

Feynman, a famous physicist, pointed out in 1982 that the classical TM seems to require exponential time to simulate a step of a quantum computer. But how powerful are these new quantum computers? In 1994 Peter W. Shor gave remarkable polynomial-time quantum algorithms for two of the most famous problems in computer science, namely FACTORING and DISCRETE LOG. What is FACTORING? Think of having a large number which is a product of two prime numbers. FACTORING is then finding those two primes.

Shor's algorithms have very big practical consequences. FACTORING and DISCRETE LOG are computational hard problems on a classical computer, meaning that we haven't yet found polynomial algorithms for solving them. And several famous cryptosystems widely used today are based on the assumptive hardness of those problems. So solving those problems efficiently means the possibility of breaking "all" the codes in the world. We will come back to this in a short moment.

IN210 – lecture 13



- E. Bernstein and U. Vazirani have recently showed that a certain problem — the recursive Fourier sampling problem — can be solved in polynomial time on a quantum Turing machine, but requires superpolynomial time on a classical TM unless $\mathcal{P}=\mathcal{NP}$.
- This was the first evidence ever contradicting the Computational Complexity Thesis!
- It has recently been proven that the class \mathcal{NP} cannot be solved on a quantum Turing machine in time $o(2^{n/2})$ unless $\mathcal{P}=\mathcal{NP}$.
- To this date (1998) the “largest” quantum computer actually build has **2** bits, but there is much research going on.
- Many excellent articles on quantum computing and complexity can be found in *SIAM Journal on Computing* Vol. 26, No. 5, pp. 1409-1557.

Autumn 1999

3 of 9

A polynomial algorithm for FACTORING is of course impressive, but it doesn't necessarily contradict the Computational Complexity Thesis even if $P \neq \mathcal{NP}$, because FACTORING is not shown to be an \mathcal{NP} -complete problem. So maybe the quantum computer isn't so powerful after all?

Two scientists E. Bernstein and U. Vazirani have recently showed that a certain problem, called the *recursive Fourier sampling problem*, can be solved in polynomial time on QTM. The interesting part here is that this problem has been shown to be impossible to solve on a classical TM using polynomial time, unless $\mathcal{P}=\mathcal{NP}$. This is the first evidence ever contradicting the Computational Complexity Thesis!

Does this mean that all our theory, we can just throw it away because it is based on the classical TM? No, even if you have the quantum computer, you can still use our basic map in order to say what sort of problems the quantum computer can solve efficiently. And this has been done. People have proven that it is very unlikely that the \mathcal{NP} -complete problems can be solved by quantum computers efficiently. It has been proven that the class \mathcal{NP} cannot be solved on a QTM in time $\Omega(2^{n/2})$ unless $\mathcal{P}=\mathcal{NP}$. This little-O notation means that the class \mathcal{NP} cannot be solved in time which is asymptotically faster than exponential time. In other words, there is at least one problem in \mathcal{NP} that a QTM needs exponential time to solve, unless $\mathcal{P}=\mathcal{NP}$. So we cannot really hope to solve any \mathcal{NP} -complete problems in polynomial worst-case time, even on a QTM.

Another basic observation is that QTM doesn't move the border between the solvable problems and the unsolvable ones. Because a classical TM can simu-

late a QTM if it is allowed to use exponential time. So the quantum computers cannot solve any problem which a normal TM cannot. They can only compute faster. And they cannot even solve \mathcal{NP} -complete problems efficiently unless $\mathcal{P}=\mathcal{NP}$. However they can solve some problems faster. That is the bottom line.

Your next question is maybe: Where can I buy this new quantum computer? The answer is that at the present, you cannot – no matter how much money you have. Because to this date (1998) the largest quantum computer actually built computes only *two* bits. So the quantum computer is still mainly a theoretical notion. But there is much research going on, and most scientists believe that a functional quantum computer will soon be built.

So this is roughly the story about quantum computers, and there are also all kinds of other ideas, i.e. biological computers. It might sound strange, but people are really working on these things. They are looking into genes and little organisms multiplying in their test tubes, and they want to use them for computation. There are whole issues of journals devoted to such ideas.

Why are people so attracted by *any* ideas which can result in a more powerful computer? It turns out that solving intractable problems can have very, very interesting consequences. It is not just about telling the traveling sales person which way to go – how to save gasoline. There are some much deeper problems that are related to complexity. And that we will talk about next.

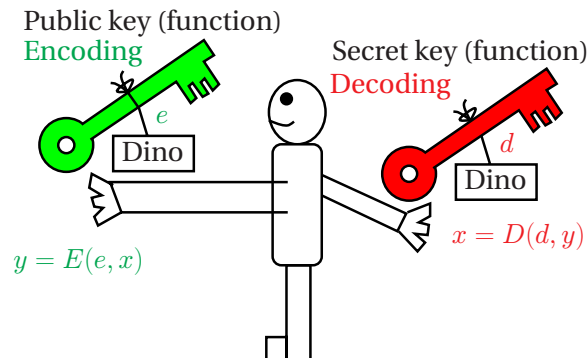
IN210 – lecture 13



Cryptography

or **cultivating complexity**

- Security & legal issues limit the use of computers.
- A foundation stone: **Public Key Cryptosystem**.



- $D(d, E(e, x)) = E(e, D(d, x)) = x$
- The system depends upon the existence of **one-way functions** — functions that are easy to compute, but difficult to invert.

Autumn 1999

4 of 9

I will now briefly cover two large research fields where we use complexity theory to understand the issues. The first field is cryptography. Cryptography introduces an interesting twist on complexity, because there complexity is actually desirable. You eliminate complexity and cryptography is in deep trouble. So whereas until now we were interested in combating complexity in all sorts of ways – eliminating complexity, simplifying things – now we are interested in finding properly complex problems, because without them there is no cryptography as we know it now.

What is the issue? The issue is practically extremely important, very big. You would like to use computers today for all sorts of things. We are linked with networks. People have their home computers and hand-held digital time managers. And many banks today, instead of having a piece of paper with your signature on it saying: "Please transfer money from my account A to my account B, or from my account to this other guy's account", allows you to just log on with your home computer and do all these transactions, see how much money you have, transfer money from one account to another. And if you are an organization, then that goes infinitely faster than sending those little guys on bicycles carrying the bills and papers and documents in their bags.

The reason why we may still need the guys on bicycles and why we may still need to send letters with signatures from Oslo to Vancouver and to Hong Kong, is the issue of security and certain legal issues. And the point is that if I transfer all my money from my account to this other guy's account, and then later I come into the bank and say: "Look, I have never done that. I don't know

what you did with my money." Then, who is really legally accountable for this? Who is responsible? It is difficult to prove that I actually did this unless I came in person and showed my ID, and so on and so forth.

So there are these issues which are obvious, and which severely limit the use of computers now. And if we can somehow get around those issues, if we can resolve them, then we can use computers in all sorts of wonderful ways and get rid of those boys bicycling around with bags full of papers. We can get rid of storing those papers, or signing legal documents and having two witnesses, etc. All can be done by just simply pressing the buttons on the computer.

This raises the issue of cryptography. Cryptography is a very old business. In the olden days before the computers it mainly had to do with one army sending messages to different parts of the army or to the general. Or it had to do with Russians or Germans having military personal around the world, and they wanted to send instructions to those people about what to do – all sorts of spying things. And then people tried to read those messages. They tried to break the codes. So there is a lot of science involved and a lot of things involved around creating those secret codes and breaking those secret codes.

With computers around there is even more interest in creating cryptographic systems – cryptosystems for short – and also there has been a lot of progress on creating cryptosystems because computers can do things which are difficult for humans. And a foundation stone for working in different ways with cryptography has been laid by the so-called *Public Key Cryptosystem* (PKC).

Let me tell roughly what this is. There are two keys involved. Those two keys they are really functions that you apply to the text, or to a sequence of bits, in order to transform the text. You want to transform the text in such a way that the characters are complete nonsense to anybody looking at them, unless the person can decode that message and turn the nonsense into sense.

So basically you can think of any cryptosystem as being two functions, one encoding function, $E(x)$, and one decoding function, $D(x)$. The two functions being inverses of each other, so that when you apply one and then the other, you get the original message. And it is nice if those two are commutable in the sense that you can apply one first and then the second, or the second first and then the first, always getting the same result, the original message.

What is interesting about the PKC is that one of the two keys, one of the two functions, is made public. So that you, an organization or a computer user or a bank, you have two keys associated with yourself. You have two keys of your own – one being a public encoding key, another one being your secret decoding key. And then what you do is you send your public key with your name on it to everybody around. You say: "Look, this is my key, and if you want to send me a message, please you this key to encode the message."

The point is that only I have my secret key which is the inverse of this encoding function. Only I am able to read that message. So although everybody has the encoder – everybody can encode messages and send them to me – only I can read them, because only I have the decoding function.

So you see how this solves zillions of problems. One of the problems is for example: You are an army in a war. Then you need some way of encoding your messages so that only the people within the army can read them. The question is: How do you distribute the keys which encodes and decodes the information? These keys or codes they have to change every once in a while. How do you send the code to people? If you are sending the secret code, you may as well send the message! Some way or the other the code must be distributed. With PKC that problem is solved. You just broadcast your public key, because it is completely public, everybody knows it.

Still, even though this public key is known, its inverse is not known. So what

is this whole thing dependent on? Just look at the situation. It is dependent on the existence of functions which have the following property: They can be computed efficiently, because obviously this encoding it has to be done in polynomial time. Unless you can encode text efficiently, it is completely useless. You cannot use zillions of years making your message unreadable to others. It has to be done fast.

So this encoding functions has to be polynomial-time computable. At the same time this is public information, so if I take this little piece, if I can invert the function, then I have this secret text right away, and I can break the code. So it is essential that this function is efficiently computable, but not efficiently reversible – the inverse of this function should not be computable from the function itself!

So the safety of these public key cryptosystems depends on the existence of the so-called one-way functions, and those are functions that are easy to compute, but difficult to invert. I will talk more about actually the mathematics behind this, which is interesting, in the IN394 class.

(This is a blank page)

IN210 – lecture 13

**Example**

The RSA (Rivest, Shamir, Adleman) cryptosystem (1978)

encoding: $y = x^e \pmod{pq}$, p and q large primes

decoding: $x = y^d \pmod{pq}$

Note: The scheme can be broken (and x computed from (y, pq, e) if pq can be factored (i.e. if p and q can be computed from their product).

Autumn 1999

5 of 9

I will now just mention one cryptosystem which is based on this scheme, and in fact it is the most famous and probably most widely used. It is called RSA by the people who designed it: Rivest, Shamir and Adleman. Here is roughly what this encoding function looks like: x is the message, y is the encoded text. Encoding is done by raising x to the power e , and then computing modulo pq of that, where pq is a large number which is a product of two primes. This e is actually an integer which is relatively prime to $\phi(pq) = pq - p - q + 1$, where ϕ is something called the Euler function.

The encoding key is made public, so e and the product pq are public information. The encoded message y can also be regarded as public information because it is mailed using a non-secure channel. But the number d , which is some kind of reverse of e (in the ring modulo $\phi(pq)$), and the prime numbers p and q , they are secret. The point is that if you have the prime numbers p and q and the encoded message y then you can use e to compute the secret d , and then decode the original message x from the encoded message y .

So the whole point here of this story, of which we don't specify the details, is that if p and q are given explicitly, then d – the secret number – can be computed and the whole code can be broken. So all this banking and national security business actually depends on our inability to figure out p and q , given this public available product p times q . If we can factor p and q out of pq , then we have broken the code.

So now we understand the implications of Shor's polynomial-time quantum computer algorithm for FACTORING. Because FACTORING is believed to be diffi-

cult to solve on a classical computer, but if somebody builds a quantum computer, then that machine can break the codes because it can factor numbers efficiently.

You might say: "Wait a minute. You just said that up to now the biggest quantum computer actually build works on only two bits. So that is no threat to cryptosystems." And you are of course right, but the interesting question is: "If somebody actually managed to build such a computer, would he announce this publically or not?" And this is no joke, because cryptography is an area which is completely full of secrets.

I have been in a conference where an Israelian guy called Shamir was supposed to present a research paper. Shamir is one of those guys who made the PKC, and also many other things. Well, Shamir couldn't, present his paper because the Israelian government forbade him to present his result. It was about probabilistic computation, cryptography and things, so they thought it was an issue of national security. So in instead of presenting the result, he was actually giving a 20 minutes presentation of this absurd situation where he was running back and forth to this building and that building, trying to get the approval for his results to be published. The outcome was 'No'. So he couldn't say anything except tell these anecdotes about the rigidity of those people.

IN210 – lecture 13



Cryptographic protocols

Example: Secret letters with **digital signatures**.

- Two persons Alice and Bob with their public (e_A, e_B) and secret (d_A, d_B) keys.
- Alice computes the letter consisting of message x (in plain text) and signature $D(d_A, x)$ (using her secret key), and encodes the whole thing using Bob's public key.
- Bob decodes the letter using his secret key (the message x is then readable to him) and then computes (encodes) the signature $E(e_A, D(d_A, x))$ using Alice's public key.
- If the result is equal to x , he knows that Alice is the sender.

Autumn 1999

6 of 9

Now I am going to finish this very brief introduction into cryptography by showing you how this PKC can be used to implement a digital signature. The meaning of this is: I can not only send an encrypted message, but I can also sign it so that you, who reads it, know that I have send the message. This is of course based on the assumption that this whole thing works, that nobody can break the code.

I will show you how a cryptographic protocol for digital signatures works. What is a cryptographic protocol? It is a kind of a procedure that two persons or two programs can use over an insecure communication channel in order to communicate safely and accomplish certain tasks. And those tasks they are of all kinds, and for each task there are certain protocols. This gives rise to all sorts of interesting problems. We have now a whole domain of problems which are solved by protocols.

One of the problems is the problem of creating a safe digital signature so that when I send you a message, I can sign it. The message is then secretly encoded so that only you can decode and read it, but somehow you must also be able to see that only I – and nobody else – sent that message.

How do we do that? In all of these schemes both Alice and Bob have two keys each. They have one encoding key each, e_A and e_B , which are public, and one decoding key each, d_A and d_B , which are secret. The encoding key is used in the encoding function and the decoding key is used in the decoding function – the two being the inverses of each other. And also the two functions commuting – meaning that you can apply them in any order.

So Alice is sending a message x , and she computes the function $D(d_A, x)$, where D is the decoding function and this d_A is her secret key. This looks kind of stupid to decode a message which is not encoded, but it is allowed because the encoding and decoding function commute. So she computes the decoding function of x and sends it together with a plain copy of x . So we now have a message $S_A(x)$ which consists of two strings – one is the message x and the other is the *decoded* version of the message x . This decoded version of message x is going to be her digital signature.

Alice then of course encodes this composite message $S_A(x)$ by using Bob's public key, and sends the whole thing to Bob. Bob can now use his secret key and decode this message $S_A(x)$ which consists of two parts. So he can read the message x , and then he can look at the digital signature that Alice sent, which is this part $D(d_A, x)$ here.

How can he check that this is Alice's signature? He applies Alice's encoding key e_A – which is public available. So he encodes $D(d_A, x)$ by using Alice's public key. In effect he computes $E(e_A, D(d_A, x))$ using Alice's public key. And if the result of this inversion is now equal to x , he knows that Alice is the sender. Nobody else could be the sender.

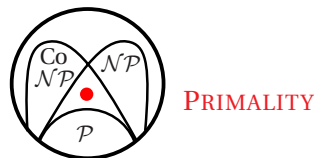
Why could nobody else be the sender? Because nobody else can do the Alice kind of decoding. Because the Alice kind of decoding requires the person having her secret key d_A , which is a large integer with certain properties.

IN210 – lecture 13



- PKCs are based on one-way functions which are easy to compute, but difficult to invert.
- RSA uses essentially PRIMALITY as the easy function and FACTORING as the supposedly difficult function.
- PRIMALITY can be shown to belong to $\mathcal{NP} \cap \text{Co-}\mathcal{NP}$.
- It is also proven that PRIMALITY belongs to \mathcal{ZPP} , meaning that it can be solved by a Las Vegas algorithm.
- There exists no polynomial-time algorithm for FACTORING on a classical TM, but FACTORING can be solved efficiently on a quantum TM.

Note: If $\mathcal{P} = \mathcal{NP}$ then any public key cryptosystem can be broken.



Autumn 1999

7 of 9

I will now just sort of summarize some of the issues we have talked about today – to see how this whole fits together. What do we know about the complexity of this whole PKC thing? I have said that PKCs depends on one-way functions, functions that are easy to compute, but difficult to invert. So it must be easy to encode the message, but difficult to decode it without knowing the secret key.

We have seen that in RSA the important part of the public key is this large number pq , which is a product of two large prime numbers p and q . p and q typically have 150 binary digits or more. Making this key must be easy, so we need to be able to find large primes efficiently. On the other hand, this scheme can be broken if somebody can figure out the primes p and q given their product pq . So factoring a large number must be difficult – FACTORING being the problem of finding the prime factors which a non-prime number is made up of.

FACTORING seems to be a hard problem on a classical computer – no efficient algorithm is found yet. But then again, Shor's polynomial-time FACTORING algorithm on a quantum computer indicates that this PKC is broken the day the first quantum computer is built. But up to now no quantum computer is built (we believe!), so PKC is safe for the moment being.

What about constructing the keys – how difficult is that? To make the keys we definitely need to be able to test efficiently whether a number is prime or not. The problem to decide if a number is a prime is called PRIMALITY.

PRIMALITY has been proven to be in the intersection between \mathcal{NP} and $\text{Co-}\mathcal{NP}$, meaning that both NON-PRIMALITY and PRIMALITY have short ID's. A good witness for a number being non-prime is of course the factors which the com-

posite number is made up of. So non-prime is obviously in \mathcal{NP} . It is less obviously that there exists a short certificate to the fact that a number is prime. How do we verify that a number n is prime? We can of course check for all numbers from 2 to \sqrt{n} that those numbers are not factors of n , but that would require exponential time in the length of n . It is not obvious that a short certificate exist, but people have shown that there indeed exists such a short witness.

PRIMALITY being in the intersection between \mathcal{NP} and $\text{Co-}\mathcal{NP}$ is good news. Although no polynomial-time worst-case & best-solution algorithm has been found for PRIMALITY, we do have fast randomized algorithms. PRIMALITY can actually be solved by a Las Vegas algorithm – this kind of miracle probabilistic algorithm which with an exponentially (in the number of trials) large probability either answers, "Yes, the number is prime" or "No, the number is not a prime".

So given a large number, we can use a Las Vegas algorithm to test efficiently whether a number is prime or not. But how many primes are there out in the world of integers? If I am a user of RSA then I say: "Here I am. Give me a secret code." and then you need an algorithm that actually produces lots of these keys, which are large prime numbers.

By using a little bit of computation you can find out that in any first n numbers, about $n/\log_e n$ of them are prime, where $e = 2.71828\dots$. This implies that the density of primes around any number n is $1/\log_e n$. What does this mean? Well, suppose that you want to find a prime which has m decimal digits. This prime-density result means that if you test $\log_e 10^m = \mathcal{O}(m)$ random decimal numbers of length m , then you are expected to find one which is prime! So it turns out to be sufficiently many of these prime numbers so that if you have a Las Vegas algorithm for testing PRIMALITY, then with a large probability you can actually find one of primes numbers in polynomial time – because there are lots of them.

The situation with the algorithm theory at this point is such that we can create these cryptosystems because we have the tools for creating large primes, but not for breaking them (factoring them).

However, nothing is really proven. Because it turns out that if $\mathcal{P}=\mathcal{NP}$, then this RSA scheme can be broken, meaning we can decode the secret message x in polynomial time. How? By using our old friend, the non-deterministic Turing machine. Given the public key p, q and e and the encoded message y , the NTM can then just guess the right message x and verify in polynomial time that x is indeed the secret message by computing $x^e \pmod{pq}$ and check that this equals y .

Most people believe that \mathcal{P} is different from \mathcal{NP} , but nobody has been able to prove it. So suppose I sit down today, it is my lucky day, and I solve HAMILTONICITY in polynomial time. You can say: "Who cares about HAMILTONICITY?" But the way this theory is structured, you know immediately that the consequence of this is that I can also do FACTORING in polynomial time because all these things are reducible to one another. If I solve HAMILTONICITY in polynomial time, then I break all these codes in polynomial time – no problem! So the consequences are in fact tremendous, even though HAMILTONICITY may not seem like such an interesting problem – who cares whether you can go around the graph in this way or not? That is food for thought – the consequences of complexity.

So this gives us an interesting view on complexity, where complexity is actually completely crucial in this world. A lot of important things in this world actually depend on certain problems, certain functions, being complex. That concludes the second part of today's lecture.

IN210 – lecture 13

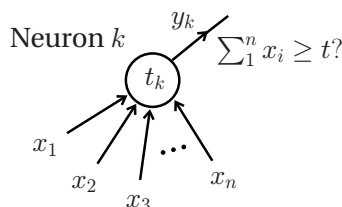


Expressive/computational power of machines & languages

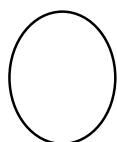
or **expressing** complexity

Sample results

- **Modeling** (Mc Culloh, Pitts, ca. 1950): Neural networks are Turing equivalent.



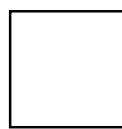
- **Logic** (Expressive power of first-order logic): First-order graph properties are in \mathcal{P} .
— First-order logic: $\forall x \exists y (\neg x \wedge F \vee y) \Rightarrow T$



problems
FLs



algorithms
TMs



properties,
theories
logic

Autumn 1999

8 of 9

I will now introduce a very large theme which we can call "the expressive power of machines and languages". This theme studied quite a bit in IN394. I will just touch it briefly here – I will mention issues and results. This issue has to do with studying all sorts of things simply by using the tools and results of complexity theory – by using our map and by using some of the insights from the map and maybe some of the techniques that we have seen.

And the first thing I mention here is a result which was made about 1950, which was a kind of a very interesting result with all sorts of consequences. The result was made by two people. One was a neurologist, another one was a kind of a computer scientist, as much as anybody could be a computer scientist in 1950. That was like just the beginning the field. The field did not really exist at that point. And the first computers were just about being made.

But these two people came together, and they were interested in studying how the human brain functions: Can we understand our thinking? Can we really understand, in some scientific way, how our brain functions? How our thinking functions?

And if we can then all kinds of things are possible. You can imagine creating a kind of a computer which functions in a similar way as our mind. And one thing is obvious again, and that is that we – the humans – we solve all sorts of unbelievably hard problems almost without even thinking. For example recognizing scenes, recognizing things. When you come into this room, you immediately know what is far and what is close. And to do computation with these problems are extremely difficult.

So there are all kinds of things that our brain does that are computationally difficult. On this most primitive level we can learn how to solve problems just by studying the brain, but the brain is obviously an interesting kind of thing to study in itself, because there are more than computational problems that are in it. So how these things actually functions is a most interesting question.

One of these two guys, the neural scientist, was quoted saying: "In neurology the problems are usually so complex that we cannot even formulate them". What does this mean? It is the basic problem of turning something into science. And in this case that "something" is a domain which is extremely important, namely understanding our neural system, understanding our brain.

So what these two characters did, they actually created a model of the neural system, based on a model of the neuron – the basic cell. And the issue there is taking something which is biochemically, physically and in every possible way unbelievably complex, and turn it into an manageable abstract model which captures the reality in a good way.

So they created an abstract model of the brain. They said that the essence of a neuron is this little figure on the foil. It has an associated threshold with it, and then there are these inputs, which are basically 0 or 1. These inputs are nerves. And a bunch of nerves come into a certain place, from other cells, and each of these nerves they can be either excited or non-excited, they can either have the value 0 or 1. And if sufficiently many of these are excited, then this neuron will also get excited and produce a 1 here as the result y_k . So if more than the threshold number of inputs is in state 1, then this neuron becomes set and produces a 1. And this 1 is carried further to the other neurons, to which this y_k is an input. So you can imagine a neural network as being a kind of a mesh of these neurons, one being input to another.

This is both a model of computation and a model of our brain or of our neural system. The first thing the two scientists did was that they proved that these neural networks can simulate a Turing machine. What is the meaning of that? We know that a TM is actually as powerful as any computer. So there is a very, very basic thing involved in proving that something is Turing equivalent, meaning that is as powerful as a Turing machine.

We know that we – the human beings – we are Turing equivalent. We can run any program, we can compute any algorithm by using our mind. So if a model of a neural system is not as powerful as a Turing machine, that means immediately that it is not a good model, because we know that we, humans, are at least as powerful as Turing machines. Conversely, if one can prove that a model of the brain can solve any problem that a Turing machine can, that means that any solvable problem is solvable by the model, and in a way that the model is powerful enough.

A very interesting question is of course whether our brain is more powerful than the Turing machine. Can we solve problems which are unsolvable? That we don't know. But we now that a TM can simulate a neural network, so if the answer is 'Yes', then the neural network is not a powerful enough model of the brain.

The point that I am making here is that by using the standard complexity tools like the Turing machine, you can study models of things – not only programs, not only solutions to processing problems, but even modeling situations – which are completely outside anything which looks like computer science. Actually these neural networks have been used to design algorithms. They have been used to solve intractable problems efficiently, but in a certain approximate sense. It is a whole area of research.

This interaction between computer science and neurology goes both ways. Neurology has learned something from computer science – this result here.

And people have learned from neurology how to make efficient algorithms for solving certain kind of problems, for example learning algorithms and pattern recognition.

We turn the page and look at another issue, which is the expressive power of logic. The issue is roughly this: In this world of theory, in addition to two big things which we have studied – problems and algorithms – there is a third thing which has to do with stating properties or stating theories. And if the first world of problems has been modeled by formal languages, and the second world of algorithms by Turing machines, then this third world of properties and theories has been modeled by formal logic.

And the most widely used logic that people study, is the first-order logic. It has quantifiers and then expressions of the Boolean kind, involving connectors AND, OR, NEGATION and the constants 'TRUE' and 'FALSE'. So these kinds of things are called expressions in the first-order logic. They express properties so that if, say, the Boolean variables x and y happens to be vertices in a graph, then we can use first-order logic to specify properties of graphs. So that first-order logic becomes a language for expressing properties of things.

The question is: How powerful is this language? What sort of properties can we express in that language? And then one result is that the first-order graph properties are in \mathcal{P} . What are graphs? Graphs are relationships. People being brothers or sisters, people knowing each other. In databases people use relational databases for storing all sorts of information about how things are related. A company and the employees of a company, they are graphs. Graphs are kind of binary relations. So you can think of graphs being models for relations in this world.

So the first-order logic is a way of specifying properties of relations in this world. It is a formal model of those basic things. Then it becomes a very basic issue to understand how powerful is this language for expressing properties. What kinds of properties can it express? The answer is that the graph properties that are expressible in first-order logic are properly contained in \mathcal{P} , 'properly' meaning that you cannot even specify all the polynomial-time computable properties in the first-order logic.

Again it has all sorts of consequences. One of the consequences is that if you can specify some property in first-order logic, then that property can be computed in polynomial time.

(This is a blank page)

IN210 – lecture 13



- **PL design** (Expressive power of programming languages):
 - Simula is Turing equivalent (**applicative PL**)
 - Prolog? (**declarative PL**)
- **Query language design** (Expressive power of database query languages):
 - *Datalog* queries are polynomial-time computabel
- **Grammars, compiler design, etc.**
- **Relationship between logic & complexity, (query) language design**

Fagin (1976):

$$\mathcal{NP}_{\text{graph}} = \text{existential second-order logic} \\ (\exists R \phi(G, R))$$

$$\mathcal{P}_{\text{graph}} = \text{first-order} + \text{while} + \text{successor} \\ = \text{first-order} + \text{fixpoint} + \text{successor} \\ = \text{Horn existential second-order} + \text{succ.}$$

Autumn 1999

9 of 9

A related issue is programming language (PL) design where we have a certain programming language and we are interested in understanding what sort of computations, what sort of algorithms, can be expressed in programming language. And the most basic kind of result there is proving that a PL is Turing equivalent. And what this means is that it can express all algorithms.

And we know that all existing reasonable general-purpose PL, such as Simula, Pascal, C, Java and so on, are easily proven Turing equivalent – meaning that they are equivalent to each other, or that they are as powerful as a language can be. They are proven Turing equivalent by simply showing that they can simulate a Turing machine. That is infinitely easier than proving that Simula can do everything that Pascal can do and everything that C++ can do, and so on and so forth. So all you have to do is to simulate a TM. A TM is such a simple thing that you are finished in 15 minutes of time proving that the PL that you have just designed and want to publish, can simulate the TM. You have proven that it is as powerful in principle as it can be.

A more interesting set of issues is this: Given so-called declarative PLs, how powerful are they? A declarative PL doesn't really tell you *how* to compute things. It only tells you: "This is what I want." So those languages allow us to specify properties without saying how those properties are to be computed.

Prolog is such a language. Artificial intelligence uses those languages quite a bit. The Japanese, when they started their 5th generation computing hype, they based it on Prolog and things like Prolog. Their main concern was that these days most people are using computers, but it is not natural for people to actually program those computers. Instead of saying "This is how I want this computed.", it is more natural to just say: "These are the properties I want, these are my questions." Such declarative questions are naturally defined using logic.

So I define my questions using logic and you, the computer, should give me the result. I don't care how you do it.

The main problem is that if I have such a declarative PL or declarative query language, then I can possibly ask questions which are in principal computable, but intractable. Then we are in deep trouble, because how can a computer provide answers to such intractable questions? So one would like to have declarative languages which have limited power. They should not really be Turing equivalent. They should allow us to specify queries which can be solved efficiently, but not those other ones which cannot.

So this is a big issue in the query language design, and the issue is not small, because a database can be a very huge thing. Using time which is exponential in the size of the database, is just an unbelievably large time. Nobody has that kind of time. So it is important to design query languages whose questions can be solved in time which is polynomial in the size of the database. And you have results like that the queries expressible in Datalog – a well-known query language based on limited kind of prolog – are polynomial-time computable.

The final issue is the relationship between logic and complexity, which is a very fruitful domain of research. It allows one to understand not only certain issues in logic, but also to have a new way of thinking about complexity.

There we want to define kinds of logic which capture exactly complexity classes. So the first result of that kind is due to Fagin around 1976. He proved that in the domain of graphs, \mathcal{NP} is equal to existential second-order logic. The meaning this is that all \mathcal{NP} graph properties, and exactly those, can be specified by using this special kind of logic which is called existential second-order.

The details are in class IN394, here I will just give you an idea what it is about. Existential second-order logic is a second-order logic which has a single existential second-order quantifier and a first-order sentence ϕ . This second-order relation R can be something like a Hamiltonian path. So $\exists R\phi(G, R)$ is saying: There exists a Hamiltonian path such that ϕ is true. And ϕ is a first-order sentence relating the graph G and the path R .

So this is a very natural way of expressing \mathcal{NP} -properties. Fagin has proven that all \mathcal{NP} [graph] properties, and exactly those, can be expressed in this way. Such results give us a possibility to design query languages which correspond exactly to complexity classes. And that has been done for class \mathcal{P} . That is obviously of practical interest, because \mathcal{P} is what is efficiently computable.

We understand now how to design database query languages by extending the first-order logic with successor function and certain loop constructs like while or fixpoint, or by restricting the existential second-order logic to only Horn-clauses instead arbitrarily first-order sentences and adding a successor function. In either way one obtain a logic which captures \mathcal{P} – which allows us to ask questions that are polynomial-time computable, and only those.

So this is one set of issues. Another set of issues is that this allows us actually embed the entire complexity theory into logic. Because if a logician can prove to you that Horn existential second-order plus successor, which is a kind of logic, can define all properties which the existential second-order logic can do, then I have proven to you that $\mathcal{P}=\mathcal{NP}$! And vice versa. If I prove to you that these two logics are distinct, I have proven that \mathcal{P} and \mathcal{NP} are distinct. As you can guess, this has not yet been proven. But the interesting point is that this is now completely a problem in logic, but it captures the $\mathcal{P}=\mathcal{NP}$? question from the theory of computing.

This has been sketchy and sort of just introducing the issues, but I hope that you get a sense that these issues, they are very broad and interesting, both intellectually and practically. And the techniques that are used here are shown in IN394. So I am not leaving you without any hope.

Kapittel 4

Oppgaver, ledetråder og løsninger

Dette kapittelet er delt inn i tre deler: oppgaver, ledetråder (hint) og løsningsforslag. Anbefalt framgangsmåte er at du først leser oppgaveteksten og prøver å løse oppgaven på egenhånd. Hvis du mot formodning skulle stå helt fast, kikker du på ledetråden til oppgaven. Når du har løst oppgaven/gitt opp (stryk det som ikke passer), kan du slå opp i løsningsforslaget. Alt dette foregår sjølsagt i god tid før gruppetimen slik at du kan bruke denne til å sette gruppelærer fast med ekle spørsmål :-)

Oppgaver med engelsk tekst er lagd av Dino Karabeg, mens “norske” oppgaver er forfattet av Stein Krogdahl og Ellen Munthe-Kaas i 1989 og 1990. Løsningsforslagene til de norske oppgavene ble i sin tid skrevet av Stein Krogdahl.

Dette kapittelet bærer dessverre preg av fortsatt å være under konstruksjon. Det mangler en del ledetråder og løsningsforslag, og en del av de gamle oppgavene er ikke helt oppdatert mhp. terminologi. Utover høsten kommer vi til å legge ut (nye) løsningsforslag til en del oppgaver på kursets nettsider:

<http://www.ifi.uio.no/~in210/>

Gamle eksamensoppgaver finnes på nettet, se peker fra IN210-kursiden.

4.1 Problems

Problem 1 (Diagonalization)

Show that there are more real numbers than natural numbers. (Important notions: Proof by contradiction, diagonalization.)

Problem 2 (Big-O notation)

Is $10n + 16n^3 = O(n^2)$? (Important notions: Approximation, big-O notation.)

Problem 3 (Coding of instances)

Numbers can be written in a variety of alphabets. Compare the approximate (big-O) lengths of natural number codes in decimal, binary and unary alphabet. (Important notions: Coding, string lengths, exponential vs. linear difference.)

Problem 4 (Hamiltonicity)

Show how HAMILTONICITY can be represented as a formal language over the alphabet $\{0, 1\}$. Discuss the relationship between the length of the representation and the number of nodes and edges in the input graph. (Important notions: strings as a formalization of problem instances, formal languages as a formalization of problems. Relationship between the code lengths and the number of elements in the input.)

Problem 5 (Turing machines)

a) Construct a Turing machine which recognizes the language over the alphabet $\{0, 1\}$ which consists of the strings of one or more 0's only. What is the time complexity of your Turing machine? (Important notions: Turing machine as a formalization of "algorithm" and "solution". The language recognized by a Turing machine. The complexity of a Turing machine.)

b) Show how the above construction can be modified into a machine which recognizes the language $L = \{1^k 0^k \mid k = 0, 1, 2, \dots\}$ (words in L consist of k zeros followed by k ones). How has the time complexity of the machine changed?

Problem 6 (Universal TM)

There is an obvious difference between a Turing machine and an ordinary computer: A Turing machine executes a single algorithm, while an ordinary computer takes in both an algorithm (program) and input data and runs the program on the input data. Show that there is a Turing machine called Universal Turing machine (UTM) which takes a TM code M and a string I as input and then does the same as the machine M would when started on input I . (Important notion: UTM)

Problem 7 (Reductions)

The proof that every decidable language is acceptable involves a reduction. Explain in detail what that reduction consists of. Can the reduction itself be computed by an algorithm? Describe how to do such a reduction by using a Universal Turing machine, and without using a UTM. (Important notions: Reduction)

Problem 8 (Xerox TM)

Describe the XTM (a Xerox TM) which, on input w , outputs $w\#w$. Notice that XTM computes a function. What is the complexity of XTM? (Important notions: TMs that compute functions and the related complexity.)

Problem 9 (Proving undecidability)

Let

$$L_1 = \{M \mid M \text{ writes a \$ for every input}\}$$

$$L_2 = \{M \mid M \text{ writes a \$ for input '010'}\}$$

$$L_3 = \{M \mid \text{There is no input } y \text{ such that } M \text{ writes a \$ for input } y\}$$

Show that L_1 , L_2 and L_3 are undecidable.

Problem 10 (TM as a bunch of templates)

Construct the templates for the Turing machine M given in class, which accepts only the string '010', and show how to construct the corresponding computation matrix for input '010'.

Problem 11 (Tiling problem)

Show how to modify the above construction into an instance of the tiling problem. Argue that the tiling problem is not solvable by showing a reduction from the HALTING problem.

Problem 12 (TM as a general grammar)

Show how to modify the construction from problem 10 into a general grammar. Show how the grammar generates the string corresponding to the halting configuration of the Turing machine. Argue that the question of whether a general grammar generates a given string w is undecidable. Explain why general grammars are not suitable for defining programming languages.

Problem 13 (Theorem-proving)

How would you show that theoremhood in first-order logic is undecidable? Use the undecidability of first-order logic theoremhood to reflect about the question "Can machines think?". If no algorithm can decide whether something is a first-order logic theorem, then no algorithm can prove theorems in first-order logic. Why is this so?

Problem 14 (Hamiltonicity)

Describe an algorithm for HAMILTONICITY. What is the complexity of your algorithm? (Important concept: Exponential-time algorithms.)

Problem 15 (Class \mathcal{NP})

Prove that HAMILTONICITY is in \mathcal{NP} by showing a non-deterministic algorithm for HAMILTONICITY. What is the complexity of that algorithm? (Important concept: \mathcal{NP} .)

Problem 16 (Shortest path in a graph)

Prove that the problem of finding the shortest path in a graph is in \mathcal{P} . (Important concept: Polynomial-time algorithms.)

Problem 17 (Reductions)

Prove that if the problem of finding the longest simple path in a graph can be solved in polynomial time, then so can HAMILTONICITY. (Important concept: Studying complexity of problems by using reductions.)

Problem 18 (Non-Hamiltonicity)

We do not know whether NON-HAMILTONICITY is in class \mathcal{NP} . What is the difficulty? (Important insight: What do problems that are beyond \mathcal{NP} look like?)

Problem 19 (Traveling Salesperson's problem)

Prove that the TSP (TRAVELING SALESPERSON'S PROBLEM) optimization problem is at least as difficult as HAMILTONICITY in the following sense: If the TSP can be solved in polynomial time so can HAMILTONICITY. (Important concepts: Reduction. Representing optimization problems by decision problems.)

Problem 20 (Cook-Levin's Theorem)

Show how to make the SAT-instance which corresponds to (encodes) the TM from problem 5a.

Problem 21

Det å gå fra et optimaliseringsproblem til det tilsvarende desisjons-problemet kan i utgangspunktet synes å være en dramatisk forenkling av problemstillingen. Som eksempel kan man tenke på Handelsreisendes problem (TSP, der vi i stedet for å spørre etter den minimale turen spør om det finnes en tur kortere enn et oppgitt tall.

Vi skal overbevise oss om at forenklingen likevel ikke er så drastisk. Anta at alle avstandene i Handelsreisendes problem er heltall, og vis at dersom vi har en prosedyre for å løse desisjonsproblemet, så kan vi også finne *lengden av den korteste turen* ved et polynomisk antall kall på denne prosedyren. (Noe liknende gjelder det å finne *selve den korteste turen*, men dette er litt mer tuklete.

Problem 22

På slutten av diskusjonen omkring Hamiltonian Path problemet (Appendiks, side 199 i G&J) står det at for rettede grafer kan dette problemet løses i polynomisk tid, dersom grafen ikke har (rettede) løkker. Skisser en slik algoritme, og gi en grov øvre grense for tidsbruk.

Problem 23

Overbevis dere om at Lemma 3.1 (side 54 i G&J) er riktig og at hver av problemene VERTEX COVER (side 46 og 190 i G&J), INDEPENDENT SET (side 194 i G&J) og CLIQUE (side 47 og 194 i G&J) er \mathcal{NP} -komplette hvis og bare hvis de to andre er det. Forsøk å finne 'dagligdags' problemer som naturlig kan formuleres som en av disse (eller som deres optimaliseringsvarianter).

Problem 24

Vis at SUBGRAPH ISOMORPHISM (side 64 i G&J) er \mathcal{NP} -komplett dersom CLIQUE er det (altså CLIQUE \propto SUB.ISOM.). Kan du finne andre problemer som her kunne ha erstattet CLIQUE?

Problem 25

Omarbeid følgende instans av SAT (her skrevet ut som logiske uttrykk) til en instans av 3SAT etter skjemaene gitt på forelesning og på side 48 og 49 i G&J:

$$(a \vee \neg b), (\neg a), (\neg a \vee b \vee \neg c \vee d \vee e \vee \neg f)$$

Problem 26

Omarbeid følgende instanser av 3SAT til instanser av VERTEX COVER, og sjekk at ja/nei-egenskapen bevares:

$$(a \vee \neg b \vee c), (\neg a \vee b \vee \neg d), (b \vee \neg c \vee d)$$

og

$$(a \vee \neg b \vee c), (\neg a \vee b \vee \neg c)$$

Problem 27

Se på problemene PARTITION og SUBSET SUM (G&J, side 223). Vis PARTITION \propto SUBSET SUM. Sjekk at du med den restriksjonen du bruker kan få fram enhver instans av PARTITION.

Vis også SUBSET SUM \propto PARTITION. For det siste tilfellet: Vis hvordan SUBSET SUM med size-verdiene (12, 25, 17, 5, 22) og $B = 37$ (ønsket subset-sum) blir omarbeidet til en instans av PARTITION.

Problem 28

a) Under problemet VERTEX COVER (side 190 i G&J) står det at det tilsvarende EDGE COVER problemet (plukk kanter slik at for hver node er minst én av kante-
ne plukket) kan løses i polynomisk tid. Algoritmen bygger på at vi har en algoritme for MAXIMUM MATCHING, som finner det maksimale antall kanter man kan velge dersom bare én kant kan velges mot hver node. (Dette er en morsom algoritme som har vist seg å løse mange problematiske \mathcal{P} -problemer (se f.eks. på 'matching' i indeksen bak i G&J). Vis at om vi har en polynomisk algoritme for MAXIMUM MATCHING så kan vi lett løse EDGE COVER problemet i polynomisk tid.

b) Se på fargeleggingsproblemet (GRAPH K -COLORABILITY, side 191 i G&J), og vis at for $K = 2$ er det i \mathcal{P} .

c) Konstater at EXACT COVER BY 2-SETS er svært nært beslektet med MAXIMUM MATCHING nevnt i (a).

Problem 29

Oppgave 3, 4 og 5 på side 75 i G&J. Disse skulle nå kunne taes forholdsvis raskt.

Problem 30

Vi skal se på oppgave 11 på side 76 i G&J, som omhandler SET SPLITTING problemet.

a) Kan du tenke deg en praktisk situasjon der dette problemet oppstår?

b) Problemet er opplagt i \mathcal{NP} , og vår første oppgave er å vise at det er \mathcal{NP} -komplett. Dette kan lettest gjøres ved å transformere fra problemet NOT-ALL-EQUAL 3SAT (NAE3SAT) (side 259 i G&J), som vi altså antar er \mathcal{NP} -komplett. Beskriv denne transformasjonen, og begrunn at den oppfyller de nødvendige krav.

c) Vi ser at hintet i oppgaveteksten i G&J går på å transformere fra 3SAT. Vis og begrunn en slik transformasjon.

d) Begrens SET SPLITTING til de instanser der mengdene i kolleksjonen C maksimalt har 2 elementer. Vis at dette problemet er i \mathcal{P} .

e) Bruk et av punktene over til å begrunne at om vi begrenser oss til de instanser av SET SPLITTING der mengdene har maksimalt 3 elementer, så er dette problemet \mathcal{NP} -komplett.

Problem 31

Vi har tre mengder $W = \{a_1, a_2, a_3\}$, $X = \{b_1, b_2, b_3\}$ og $Y = \{c_1, c_2, c_3\}$. Vi har gitt 100 tripler fra $W \times X \times Y$, hvorav de 4 første er: (a_1, b_2, c_3) , (a_1, b_3, c_2) , (a_2, b_2, c_1) og (a_3, b_1, c_3) .

Dette er en instans av 3DM. Vis hvordan den vil bli transformert til en instans av SUBSET SUM, etter metoden forklart på forelesningen, som er den samme som er forklart i G&J på side 60 - 62 (fram til 'The final step ...', der man transformerer videre til PARTITION). Vi kan passelig bruke ti-tall-systemet i stedet for to-tall systemet som er brukt i G&J.

Problem 32

De mest typiske ‘tallproblemer’ er de der instansen rett og slett består av ett (eller et par) heltall, med et passelig spørsmål til. Det ‘tradisjonelle’ størrelsesmålet for instanser av slike problemer er å bruke antall siffer i tallet (tallene), og vi kan da spørre om det finnes polynomiske algoritmer i forhold til dette målet. Alternativt kan vi si at størrelsen av instansen er selve tallverdien av tallet (tallene), og algoritmer som er polynomiske i forhold til dette målet kalles ‘pseudopolynomiske’ (se G&J, kap 4.2 og forelesningen uke 10). Tenk på noen algoritmer for følgende problemer, og avgjør om de er polynomiske eller pseudopolynomiske (eller ingen av delene).

- Ett tall: Avgjør om tallet er et primtall.
- To tall: Summer de to tallene.
- To tall: Avgjør om det første tallet går opp i det andre.
- To tall: Finn største felles divisor for de to tallene. Her kan man bl.a. tenke på Euklids algoritme, og det er ikke uten videre klart hvor fort den går(!?).
- Ett tall T : Finn det største heltallet x slik at $x^5/7 + x^3$ ikke overstiger T .
- Ett tall T : Finn det største heltallet x slik at $\log_2 x$ ikke overstiger T .

Problem 33

Vi skal se litt nærmere på RYGGSEKK-problemet (KNAPSACK, side 65 og 247 i G&J). Vi angir n instans av problemet ved $B, K, ((s_1, v_1), (s_2, v_2), \dots, (s_n, v_n))$, der B er en øvre grense for s -summen, og K er en nedre grense for v -summen i en løsning.

- Utred kort sammenhengen mellom dette problemet, og det å pakke påske-sekken.
- Se på hver av de nedenfor gitte restriksjoner av problemet, og se om du kan finne polynomiske algoritmer som løser dem (slik at de restrikterte problemene altså er i \mathcal{P}). Noen av punktene kan ha sammenheng med senere spørsmål.
 - Alle s -verdiene er like.
 - Alle v -verdiene er like.
 - Alle v -verdiene er enten 1 eller 2.
 - Alle v -verdiene og s -verdiene er (heltall) mellom 1 og 10.

På side 65 i G&J står det forklart hvordan man viser at RYGGSEKK-problemet er \mathcal{NP} -komplett ved å restrikttere det til PARTITION. Dermed er det ikke mye håp om å finne en polynomisk algoritme, men det kan være håp om å finne en ‘pseudopolynomisk algoritme’ (altså en som er polynomisk dersom vi i ‘problemstørrelsen’ lar tallene i problemet inngå med sin *tallverdi*, i stedet for med sitt antall siffer, som er det tradisjonelle).

Vi skal vise at en slik pseudopolynomisk algoritme finnes, og som inspirasjon skal vi bruke metoden angitt på side 90 i G&J, med ‘dynamisk programmering’. Vi skal bruke en tre-dimensjonal tabell der $t(i, j, k)$ angir (ved TRUE/FALSE) om man fra mengden $((s_1, v_1), (s_2, v_2), \dots, (s_i, v_i))$ kan gjøre et utplukk der s -summen er nøyaktig lik j og v -summen er nøyaktig lik k .

- Over hvilke intervaller for i, j og k vil du forsøke å fylle ut denne tabellen?

Hvordan må tabellen ende opp for at svaret skal være 'ja'?

- d) Hvordan vil du fylle ut det første laget i tabellen, for $i = 1$?
- e) Vis hvordan du kan fylle ut lag $i + 1$ i tabellen når lag i er fylt ut, og skriv det hele ut som en programskisse.
- f) Påvis at algoritmen er pseudo-polynomisk, ved å angi en øvre grense for tidsforbruket ved O -notasjon.

Problem 34

Vi skal se på forskjellige varianter av SUBSET SUM, og oppgaven er å forsøke å bestemme hvilke av variantene som kan løses i polynomisk tid, hvilke som er \mathcal{NP} -komplette men kan løses i pseudopolynomisk tid, og hvilke som er \mathcal{NP} -komplette i sterk forstand (\mathcal{NPC} i sterk forstand betyr at problemet ikke kan løses av en pseudo-polynomisk algoritme hvis $\mathcal{P} \neq \mathcal{NP}$). Vi tenker oss i utgangspunktet at vi har gitt en sekvens av heltall (s_1, s_2, \dots, s_n) samt et tall B .

- a) Vi spør om det finnes et utplukk av s -verdier som gir sum B , og der avstanden mellom laveste og høyeste indeks på s -ene i utplukket ikke er større enn $n/2$.
- b) Tallene s_1, \dots, s_n utgjør også nodene i en oppgitt graf, og vi spør om det finnes et utplukk av tallene som gir sum B og som utgjør en nodeoverdekning i den gitte grafen.
- c) Vi spør om det finnes et utplukk som har sum B og som utgjør en sammenhengende subsekvens i den oppgitte sekvens av s -verdier. (Hva skjer om vi her tillater at de utplukkede tallene kan få utgjøre opp til *tre* sammenhengende subsekvenser?)

Problem 35

Bruk den generelle metoden som er eksemplifisert ved Traveling Salesman på side 116 og 117 i G&J og som er gjennomgått på forelesning til å vise at problemene gitt under er \mathcal{NP} -lette. Man kan kanskje bruke (b) til å repetere hele gangen i beviset, og ellers bare skissere det essensielle. Problem (d) er litt spesielt.

- a) Gitt en graf G . Finn en Hamiltonsk løkke om en slik finnes.
- b) Gitt en graf G . Finn en 'klikk' med så mange noder som mulig.
- c) Gitt en KNAPSACK-instans, men se bort fra verdien K (nedre grensen på verdi-summen). Finn et 'lovlig' utplukk (der s -summen altså ikke overgår B) som har maksimal v -sum.
- d) Gitt en graf G , med positive heltallige vekter på nodene. Finn en uavhengig nodemengde som har så mange noder som mulig, men som innenfor dette har så *liten* vekt-sum som mulig.

Problem 36

Vurder følgende uttalelser (Noen, men ikke alle, er greie):

(A): "Jeg har løst SAT i polynomisk tid, og derfor kan jeg løse ethvert \mathcal{NP} -hardt problem i polynomisk tid."

(B): "Mitt problem X , som jeg har vist er \mathcal{NP} -hardt, har jeg klart å løse i polynomisk tid. Derfor er $\mathcal{P} = \mathcal{NP}$."

(C): “Jeg kan vise at mitt problem X er i \mathcal{NP} , og at $\text{SAT}_{\infty T} X$. Derfor er X \mathcal{NP} -komplett.”

(D): “Det er mange desisjonsproblemer som kan løses i polynomisk tid, men der et løsningsforslag ikke kan *bevitnes* i polynomisk tid (og de ligger dermed utenfor \mathcal{NP})”.

(E): “Et problem kan ikke samtidig være både \mathcal{NP} -hardt og \mathcal{NP} -lett”.

(F): “Alle desisjonsproblemer som er \mathcal{NP} -lette ligger i \mathcal{NP} ”.

Problem 37

Vi skal se litt på tilnæringsalgoritmen for VERTEX COVER som omtales øverst på side 134 i G&J og som i praksis er identisk med algoritme VC-H2 angitt i forelesningen i uke 10. Den bygger på først å velge en “matching” (altså et utplukk av kanter som ikke har noen felles endenode), og denne skal velges ved å starte med en tilfeldig kant, og så legge på nye kanter inntil det ikke er mulig å legge på fler. En slik matching kaller boka en “maximal matching”, som vi kan oversette med en “ikke-utvidbar matching” (dette i motsetning til en “maximum matching”, som er en matching med så *mange* kanter som mulig).

a) Vis at alle endenodene av en ikke-utvidbar matching må utgjøre en nodeoverdekning (et “Vertex Cover”).

b) Vis at ingen nodeoverdekning kan ha færre noder enn antall kanter i en (vilkårlig) matching.

c) Som en tilnæringsalgoritme A for å finne et Vertex Cover med så få noder som mulig bruker vi følgende: (1) Velg en eller annen ikke-utvidbar matching. (2) Bruk endenodene på denne som nodeoverdekning.

Vis at dette er en ϵ -tilnæringsalgoritme med $\epsilon = 2$ (G&J sin notasjon: $R_A = 2$). Gi også et eksempel på at algoritmen “treffer” akkurat denne epsilonverdien – med andre ord skal du angi en instans der denne algoritmen gir en løsning som er akkurat dobbelt så stor som den optimale (OPT). Vis også at uansett hvor stor OPT-verdi grafen har, så kan du angi en slik instanse der A finner en løsning som er akkurat det dobbelt av OPT. (G&J sin notasjon: $R_A^\infty = 2$).

d) Vi legger til et “strykesteg” i algoritmen angitt i (c): (3) Let etter en “overflødig” node (altså en node som er slik at dersom den fjernes, utgjør de gjenværende nodene fremdeles er en nodeoverdekning), og fjern denne. Gjenta dette til prosessen stopper av seg selv. Kan du si noe om R_A og R_A^∞ for denne varianten?

e) For å få denne algoritmen så god som mulig er det fint om vi kan klare å velge en ikke-utvidbar matching med så *få* kanter som mulig. Forsøk å avgjøre om det er mulig i polynomisk tid å finne en slik matching i en generell graf.

Problem 38

Vi skal se på et eksempel på “skalering” (truncating/rounding off), altså den teknikken som er benyttet for Ryggsekk-problemet på side 136 i G&J og på forelesningen i uke 10. Problemet vi skal arbeide med er Multiprocessor Scheduling (side 65 og 238 i G&J), og vi skal begrense oss til $m = 2$, (to prosessorer) og se på det som et minimaliseringsproblem: Gitt n prosesser med (positive heltallige) tider (t_1, t_2, \dots, t_n) , minimaliser den tid det tar før alle prosessene er utført.

Dette kan sees som en variant av PARTITION, der vi skal minimalisere størrelsen av “den største delen”. Dette problemet kan løses med standard dynamisk programmering (som på side 91 i G&J) i antall steg $O(nT)$, der T er summen av alle t_i -ene (det klarer seg med tabell opp til $T/2$).

Vi skal se på en tilnæringsalgoritme A_K , der vi først skalerer ned t_i med en faktor K , slik at vi får et nytt problem I' med $t'_i = \lfloor t_i/K \rfloor$. Dette skal vi så løse nøyaktig med dynamisk programmering, og benytte den oppdelingen (vi kan kalle de to prosess-mengdene for U'_1 og U'_2) som kommer ut av dette som en tilnærming til den *best mulige* oppdelingen for det opprinnelige problemet I (den beste oppdelingen kan vi si er U_1 og U_2).

Vi kan anta at nummereringen er gjort slik at $t(U_1) \leq t(U_2)$ og $t(U'_1) \leq t(U'_2)$ (der $t(U)$ angir t -summen over prosessene i U).

- a) Forklar at $t(U_2) \leq t(U'_2)$.
- b) Sett opp et uttrykk for $R_{A_K}(I)$, der A_K er algoritmen angitt over.
- c) Anta at vi klarer å vise at $t(U'_2) - t(U_2) \leq a \cdot n \cdot K$, for en passelig konstant a . Vis hvordan vi da, for enhver ønsket nøyaktighet $1/k$ kan velge K slik at denne blir oppnådd (altså $R_A \leq 1 + 1/k$), og slik at det hele utgjør et fullstendig polynomisk tilnærmingsskjema.

Problem 39

Consider the problem of collecting all of the n existing coupons and the "standard algorithm" which consists of buying chocolates (each containing one coupon, chosen at random) until the goal is accomplished.

- a) Calculate the expected running time of your algorithm.
- b) Turn your algorithm into an algorithm which makes a pre-defined number of attempts $M(n)$ (i.e. one buys a pre-defined number of chocolates) so that the probability of success is at least $1/2$. How large does $M(n)$ have to be? Prove your answer.

Problem 40

Describe an algorithm for finding a triangle subgraph in a given input graph with n nodes.

- a) What is the worst-case complexity of your algorithm?
- b) What is the average-case complexity of your algorithm (assume that for each pair of vertices the probability that the pair is an edge is equal to $1/2$)? How would you change your algorithm so that its average-case performance is improved?

Problem 41

A skeptic would say that everything in life is uncertain. In the probabilistic language we would say that every event has a certain "finite" probability of occurring (probability greater than zero and smaller than one). Fortunately (for optimists) it turns out that the probabilities of events tend to be either very large (close to one) or very small (close to zero). In the probabilistic language statements of this kind are called zero-one laws. Zero-one laws mean that we can very often in practice safely disregard the probabilistic nature of things and make claims or promises with relative certainty. The following is an illustration of that point.

Think of tossing a coin one hundred times and getting one hundred heads in a row. It is immediately obvious that this is very little probable. Getting one hundred heads is a low-probability event, as anyone will agree. But if I tell you that you have your entire life-time to toss the coin – would you count on getting

one hundred heads (or tails) in a row at least once? Well, the intuition at this point betrays us. Let us therefore resort to calculation. Estimate (roughly) the probability of getting one hundred heads or tails in a row if all of the human kind were tossing coins once every second since the beginning of known history (say, for 10 000 years). Conclude the following: Who knows what sorts of possibilities are hidden in nature as low-probability events!

Problem 42 (Monte Carlo algorithms)

Evaluate the following algorithm for COMPOSITENESS:

Input: natural number N

Guess a natural number K , ($1 \leq K \leq \sqrt{N}$) by using a random number generator.

if N is divisible by K **then** answer 'Yes'

else answer 'probably No'.

Is this a Monte Carlo algorithm? Justify your answer.

Problem 43 (Heuristics)

Heuristics are a common way of dealing with optimization problems in practice. They differ from algorithms in that they have no performance guarantees (they don't always find the best solution). Construct a greedy heuristic for finding the shortest path between two nodes in a network. (A network is a graph whose edges have positive integer 'lengths'.) Show that your heuristic is not an algorithm. Is your heuristic a polynomial-time approximation algorithm? (By our definition, a poly-time approx. alg. must always give solutions that are within epsilon from the optimum, for a fixed constant epsilon.)

Problem 44

Show a dynamic programming algorithm for the shortest path between two nodes in a network (cf. Dijkstra's algorithm). Show a dynamic prog. alg. for the longest path between two given nodes in a network. What are the time complexities of your algorithms? What are the storage requirements?

Problem 45

Sketch a branch-and-bound algorithm for the TSP. What is the time complexity of your algorithm? What are the storage requirements?

4.2 Hints

Hint to problem 1

Try to use a counting argument: Towards a contradiction assume that there are as many integers as real numbers. Then there must be possible to make some kind of 1-1 mapping from integers to real numbers. Try to make such a mapping by constructing a big, infinite matrix with integers as row labels and real numbers as rows (meaning that all entries in a row taken together represents a real number). Use the diagonalization technique shown in class to prove that there exists at least one real number which is not in the matrix. This is a contradiction, so you can then conclude that there is more reals than integers in this world.

Hint to problem 2

What does $t(n) = \mathcal{O}(n^2)$ mean? Informally it means that $t(n)$ doesn't grow faster than n^2 -polynomials when n is "large enough". Formally it means that there

exists positive constants c and n_0 such that $t(n) \leq cn^2$ for all $n \geq n_0$. You must either come up with constants c and n_0 such that $10n + 16n^3 \leq cn^2$ for all $n \geq n_0$, or show that such constants cannot possibly exist.

Hint to problem 3

Code some numbers in decimal, $\{0, 1, \dots, 9\}$, binary, $\{0, 1\}$, and unary, $\{1\}$, alphabet. Try to see a pattern – a relationship between the length of the codings in the three alphabets.

Hint to problem 4

What does it mean "To represent HAMILTONICITY as a language over the alphabet $\{0, 1\}$?" It means to encode each possible instance of HAMILTONICITY as a string of zeros and ones. The codes which encode Hamiltonian graphs are the words in the formal language which corresponds to the HAMILTONICITY problem.

What are the possible instances of HAMILTONICITY? They are graphs. We need to represent graphs as strings of zeros and ones.

What is a graph? A graph consists of two sets, V (vertices) and E (edges), where E consists of pairs of elements from V .

How does one represent a set by using zeros and ones? The natural way is to use binary numbers from 1 to n to represent each element in the set (where n is the number of elements in the set). But what about separators such as commas? Obviously, one needs more than two symbols (if the code should be "reasonable" i.e. not have an exponential length). But one could easily construct the additional symbols by using multiple (two) characters for each symbol.

Notice that the encoding rules are not part of the input. It is assumed that the Turing machine understands how the graph is encoded.

Variations: How can one represent the TSP (TRAVELING SALESPERSON'S PROBLEM, see the textbook for definition) or SORTING or MATCHING?

Hint to problem 5

a) What is a Turing machine? A TM is by definition a quadruple: an input alphabet Σ , a tape alphabet Γ , a set of states Q and a transition function δ . We need to come up with those four things, then.

Those four things are what we end up with, but not what we begin with. It is better to begin with high-level ideas, as in structured programming. How should the machine behave? It should halt writing a 'Y' on the tape if and only if the input is a string of zeros, otherwise it should halt writing a 'N'. Here is a possible pseudocode solution:

```

Scan the first character;
if the character is a '0' then erase the '0' by a blank and move the head to the right
else <blank out the rest of the input, write a 'N' and halt>;
while the scanned character is non-blank do
begin
  if the character is a '0' then remain in the same state, erase the '0'
    by a blank and move the head to the right
  else if the character is a '1' then <erase the rest
    of the input, write a 'N' and halt>;
end
Write a 'Y' and enter state  $h$ ;

```

The input alphabet is $\{0, 1\}$, while the tape alphabet could be $\{0, 1, b, Y, N\}$. Spell out the remaining details of the machine.

b) The machine has to move its head back and forth, eliminating one '1' for each '0'. Notice that, although the back-and-forth movement is extensive (which is characteristic for a Turing machine), the back-and-forth movement increases the complexity of an algorithm "only" by a polynomial factor. That is why a polynomial algorithm on an ordinary machine translates into a polynomial algorithm on a TM.

To make the machine simpler, you don't have to code the cleaning-up part where the machine blanks out the tape before writing a 'Y' or a 'N'. You can just write a 'Y' or 'N' and halt.

Variation: Construct a TM which recognizes palindromes (words which read the same left-to-right as right-to-left such as "abracacarba") and answer the same questions.

Hint to problem 6

Assume that the UTM has three tapes, one for the input and output, another one for storing the input machine code M (its rules), and the third one for book keeping, i.e. for keeping track of the state of the simulated machine etc. . The UTM computes each step of the machine M by looking at the input scanned by M and the state of M , and then finding the corresponding entry of the transition function on tape 2. Note 1: A TM machine with three tapes can be simulated by a machine that has only one tape. Note 2: The UTM must exist according to the Church-Turing Thesis. But since a thesis is itself not proven or provable, we can not use it in formal proofs.

Hint to problem 7

Assume that L is decidable and prove that L is also acceptable. The assumption means, by definition, that there is a TM M which decides L . What needs to be proven is, by definition, that there is a TM M' which accepts L . Thus the reduction amounts to a proof that M' exists provided that M exists. One may prove that by showing how M can be modified to obtain M' , or by simulating M by using the UTM and then doing some obvious additional work.

Hint to problem 8

Copy character by character. Use "marking" (by a special character) to denote the last copied character. Notice once again that moving the head back and forth has only a polynomial cost.

Hint to problem 9

The proof given in class works with small modifications. It is important to work through the details carefully, but at the same time see the larger picture—what is the physiognomy of the undecidable problems?

Assume towards a contradiction that L_1 is decidable by a machine M_{L_1} . Show that we can then solve L_H by using M_{L_1} as a kind of subroutine. Conclude that this is a contradiction since L_H is unsolvable, which means that your assumption about L_1 being decidable, must be wrong.

Hint to problem 10

See lecture 4.

Hint to problem 11

See lecture 4.

Hint to problem 12

See lecture 4.

Hint to problem 13

See lecture 4.

Hint to problem 14

Use exhaustive search. The complexity should be exponential.

Hint to problem 15

Guess a cycle in linear time by using non-determinism. Verify in deterministic polynomial time that the guessed path is indeed Hamiltonian.

Hint to problem 16

It is important here to feel the nature of polynomial and "exponential" problems. You may use the standard algorithm ("Dijkstra") or develop one on your own.

Hint to problem 17

Construct a reduction from HAMILTONICITY ("Is there a simple path that visits all the vertices exactly once?") to LONGEST SIMPLE PATH ("Show me the longest simple path"). Intuitively this means that you solve HAMILTONICITY by using the algorithm for finding the longest simple path as a subroutine. Argue that the reduction can be computed in polynomial time. Conclude that if LONGEST SIMPLE PATH is solvable in polynomial time, so is HAMILTONICITY by virtue of the reduction.

Hint to problem 18

Observe that NON-HAMILTONICITY doesn't seem to have any short "certificate" of membership.

Hint to problem 19

Show a reduction from HAMILTONICITY to TSP.

Hint to problem 20

Look at lecture 6 or section 2.6 in G&J.

Hint to problem 21

Use binary search.

Hint to problem 22

Bruk en modifisert versjon av standardalgoritmen for topologisk sortering.

Hint to problem 23

N/A

Hint to problem 24

Hint på side 64 i G&J.

Hint to problem 25

Oversett klausulene hver for seg. Sjekk hvor mange literaler klausulen har: 1, 2, 3 eller mer enn 3. Innfør hjelpevariable etter mønster i læreboka eller forelesning uke 6.

Hint to problem 26

Lag først en “truth-setting” komponent for hver enkelt logiske variabel i 3SAT-instansen. Lag deretter en triangel-subgraf for hver klausul. Koble komponentene sammen etter mønster angitt i læreboka eller forelesning uke 7. Angi til slutt riktig antall vakter i VC-instansen som en funksjon av antall variabler og klausuler i 3SAT-instansen, og vis at ja/nei-egenskapene bevares.

Hint to problem 27

For SUBSET SUM \times PARTITION: Se side 62 i G&J eller forelesningen uke 7.

Hint to problem 28

b) Forsøk med en enkel traverseringsalgoritme.

Hint to problem 29

N/A

Hint to problem 30

c) Denne reduksjonen blir litt mer komplisert enn i (b), men prøv med et ekstra ‘false-element’ i mengden.

d) Vis at det da kan sees som (eller ‘reduseres til’) et problem i \mathcal{P} som vi har vært borte i tidligere.

Hint to problem 31

N/A

Hint to problem 32

N/A

Hint to problem 33

N/A

Hint to problem 34

N/A

Hint to problem 35

N/A

Hint to problem 36

Påstand (C): Les G&J side 158, litt under midten.

Hint to problem 37

N/A

Hint to problem 38

N/A

Hint to problem 39

a) Break the problem into subproblems. First try to calculate the expected number of buys before getting the i 'th coupon, given that you already have $i - 1$ out of the n different coupons. Call this random variable for $X_i(n)$. To calculate this sum you can essentially use the same approach as shown in lecture 11 (3-COLORABILITY).

The expected running time of the algorithm is then $\sum_{i=1}^n X_i(n)$ because we first need to get the first coupon, then the second, and so on. To calculate this sum, remember that the expectation of a sum of random variables is equal to the sum of the expectations of the individual variables. You might also want to know that the sum of the harmonic series $\sum_{i=1}^n \frac{1}{i}$ is approximately $\mathcal{O}(\ln n)$ where \ln is the natural logarithm.

b) Use the *Markov Bound* which says that $\Pr[X \geq kE(X)] \leq 1/k$, where X is any random variable and $E(X)$ its expectation.

Let X be the number of chocolates we have to buy in order to get all n coupons, and let $E(X)$ be the expected value of X found in a). We want to have the probability of success $\geq 1/2$, so $k = 2$ is a natural choice. Then the Markov Bound says that the probability that we have to buy more than $2 \times E(X)$ chocolates is less than $1/2$.

Hint to problem 40

The natural algorithm draws 3 vertices at random and checks whether they form a triangle subgraph. If they do, then everything is fine. If they don't, the algorithm tries another 3-vertex set.

a) Assume that there are no triangle subgraphs in the input graph. How many 3-vertex sets must the algorithm check before it can say "No, there are no triangle subgraphs in this input graph"?

b) Average-case analysis: What is the probability that three vertices picked at random form a triangle subgraph? Then continue as in the example in lecture 11 (3-COLORABILITY).

A smarter algorithm: How can you pick three vertices so that you know that there are at least two edges among them? If you can do that, then you that there is a probability of $1/2$ for the last pair of vertices being an edge.

Hint to problem 41

Estimate the total number of coin tosses in the human history and divide this number by the probability of getting 100 head in a row.

Hint to problem 42

Argue that COMPOSITENESS is in \mathcal{NP} , but that the number of "naive" certificates is small. What is needed is some way to come up with many certificates. The actual Monte Carlo algorithm for COMPOSITENESS involves some very advanced mathematics.

Hint to problem 43

A greedy algorithm does what looks best at each step, without any global considerations. Given a node s in a network. If s has three outgoing edges with weights 2, 5 and 3 respectively, which of them will a greedy algorithm choose? Give a counter-example which shows that the greedy approach doesn't find the optimal solution. How far from the optimal solution can the "greedy solution" be? Is the relative error limited by a constant or does it grow as function of the input size?

Hint to problem 44

N/A

Hint to problem 45

N/A

4.3 Solutions**Solution to problem 1**

Let \mathbb{N} and \mathbb{R} be the set of natural and real numbers, respectively. We want to prove $|\mathbb{N}| < |\mathbb{R}|$. We use "proof by contradiction":

Assume that $|\mathbb{N}| \geq |\mathbb{R}|$. Then $|\mathbb{N}| = |\mathbb{R}|$, since every natural number is also a real number. It must then be possible to create an (infinite) mapping from elements in \mathbb{N} to every element in \mathbb{R} . The table below illustrates one possible mapping, using binary representation for the real numbers. The table is infinite in both directions. Without loss of generality we can put the real numbers between zero and one at the top.

Int.	Real numbers
1	0. 1 0 1 0 1 0 0 1 ...
2	0. 1 0 0 1 0 1 0 0 ...
3	0. 0 1 1 0 1 1 0 1 ...
4	0. 1 0 1 0 0 1 0 0 ...
5	0. 1 1 0 1 0 0 1 0 ...
⋮	⋮ ⋱

If we invert the bits in the diagonal, we will in the above example get the real number 0.01011... which is different from every real number in the table (it's different from the n 'th number in the n 'th digit of the decimal fraction). This *diagonalization* shows that the table cannot contain all real numbers, and the assumption $|\mathbb{N}| \geq |\mathbb{R}|$ cannot be true. We conclude that $|\mathbb{N}| < |\mathbb{R}|$.

Solution to problem 2

The answer is no. If $10n + 16n^3 = O(n^2)$ then there exists positive constants c and n_0 such that $10n + 16n^3 \leq cn^2$ for all $n \geq n_0$. To prove that such constants cannot be found, we have to find a value m such that $m > n_0$ and $10m + 16m^3 > cm^2$, no matter what n_0 and c are. We can choose $m = cn_0$ (assuming $c > 1$) because $10cn_0 + 16(cn_0)^3 > (cn_0)^3$.

Solution to problem 3

Let D , B and U be the decimal, binary and unary encoding of the same number. If the encoded number is e.g. 14 then $D = 14$, $B = 1110$ and $U = 11111111111111$.

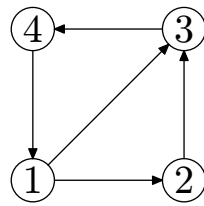


Figure 4.1: A graph

Let $|D| = n$. What is then the length of B and U ?

$$\begin{aligned} |B| &= n \log_2 10 = O(n) \\ |U| &= 10^n = O(10^n) \end{aligned}$$

It does not matter if we encode numbers in any radix greater than 1. The length will be roughly the same. Unary encoding, on the other hand, is not a sensible encoding because we get an exponential increase in length.

Solution to problem 4

Let $G = (V, E)$ be a directed graph. V is the set of *vertices*, and E is the set of *edges*. The graph is Hamiltonian if it has a Hamiltonian cycle i.e. a simple cycle that includes all the vertices of G . To show that *Hamiltonicity* can be represented as a formal language over the alphabet $\{0, 1\}$, we must show that every graph can be encoded as a string in the alphabet $\Sigma = \{0, 1\}$. Then

$$L_H = \{x : \text{The graph represented by } x \text{ has a Hamiltonian cycle}\}$$

Take the graph in figure 4.1 as an example. (The graph is Hamiltonian, but that is not important.) How can we represent that graph?

Idea 1 The simplest way to represent a graph is to list its nodes and edges. The above graph would then be written like:

$$\langle 1, 2, 3, 4 \rangle, \langle (1, 2), (2, 3), (3, 4), (4, 1), (1, 3) \rangle$$

We don't have to represent the parentheses directly, but we have to separate the list of nodes from the list of edges, say by a ';'. If we use a binary representation of numbers, then we need four symbols: '0', '1', ';' and '>';'. These can be represented by 00, 01, 10 and 11 respectively using our two-symbol alphabet.

If there are n edges, every number takes at most $2 \log_2 n$ symbols to represent. There are at most n^2 edges. The length of the representation will then be at most:

$$2n(\log_2 n + 1) + 2n^2(\log_2 n + 1) = O(n^2 \log n) = O(n^3) = O(p(n))$$

The important point is that the length of the representation is polynomial in n . (Question: Could we have used unary representation here and still had a polynomial length?)

$$\Sigma = \{0, 1\}, \Gamma = \{0, 1, b, Y, N\}, Q = \{s, q_1, q_e, h\}$$

state	0	1	b
s	(q_1, b, R)	(q_e, b, R)	$(h, N, -)$
q_1	(q_1, b, R)	(q_e, b, R)	$(h, Y, -)$
q_e	(q_e, b, R)	(q_e, b, R)	$(h, N, -)$

Figure 4.2: $L = \{x \mid x \in 0^+\}$

Idea 2 A graph can be represented as a 2-dimensional matrix. The graph in the example above will then look like:

	1	2	3	4
1	F	F	F	T
2	T	F	F	F
3	T	T	F	F
4	F	F	T	F

This is easily represented as a string in 0, 1 by putting one row after the other and using one bit to represent a boolean value. The length of the representation will be $n^2 = O(n^2) = O(p(n))$.

Solution to problem 5

a) A TM that recognizes all strings in the alphabet $\Sigma = \{0, 1\}$ which consists of the strings of one or more 0's only is presented in figure 4.2. A dash ('-') means that the read/write head doesn't move (if the definition of the TM doesn't allow it, then it can be simulated in two steps by first moving right and then moving left). The time complexity is $O(n)$ where n is the length of the input string.

b) Let $\Gamma = \{0, 1, \$, \#, b\}$. We can now mark out the first "1" with a \$, then the first "0" with a #. We then go back and forth, marking the same number of 1's and 0's. The TM in figure 4.3 does this. The time complexity function is $O(n^2)$.

Solution to problem 6

We use three tapes for the Universal Turing Machine (UTM), one for the input and output, another one for storing the input machine code M , and the third one for book keeping, i.e. for keeping track of the state of the simulated machine (see figure 4.4). This is OK because a machine with three tapes can be simulated by a machine that has only one tape.

To encode the input TM, we need a way to represent the transition function. One simple way is to represent it as a list of 5-tuples on the form (state, read-symbol, new-state, print-symbol, direction). We must also have special encodings of the special states (s and h).

The UTM starts with the encoding of s on tape 3 and the input on tape 1. Each computational step consists of:

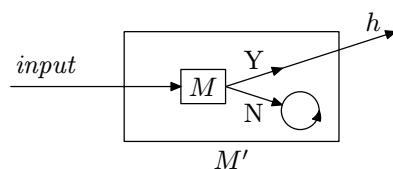


Figure 4.5: Reduction

1. Search on tape 2 for the tuple containing the state of tape 3 and the input symbol of tape 1 as the two first elements,
2. copying the new state to tape 3.
3. copying the print symbol to tape 1 and
4. moving the head of tape 1 according to the encoded direction of tape 2.

The UTM halts when the new state is one of the halt states.

Solution to problem 7

Prove: L is decidable $\Rightarrow L$ is acceptable.

Assume that L is decidable. Then there exists a TM M which decides L . If there is a TM M' which accepts L , then L is acceptable. So we try to construct M' , using M as a “subroutine”. This is called a *reduction*. If we succeed, M' must exist when M exists.

In figure 4.5 we can see that the TM M' is constructed by modifying the halting behaviour of M : If M writes a 'Y' and halts, then M' just halts. If M writes a 'N' and halts, then M enters an infinite and never halts.

This reduction can be performed automatically in (at least) two ways.

Modifying M Assume that M has m states. We can make a TM that takes M as input, and modifies the transition function by replacing entries of the form $(h, N, -)$ by (q_{m+1}, b, R) where q_{m+1} is a new state simulating a loop (for example by always moving to the right, staying in state q_{m+1} , whatever the scanned symbol looks like).

Using the UTM We can use a variant of the UTM called UTM' . UTM' can be constructed by modifying the UTM as described above. UTM' will then behave as M' given M as input.

Solution to problem 8

The TM in figure 4.6 will produce $\$w\#w\$$ on input $\$w\$$. The time complexity function is $O(n^2)$. For simplicity the input alphabet is set to $\{0, 1\}$.

Solution to problem 9

N/A

Solution to problem 10

N/A

$$\Sigma = \{0, 1\}, \Gamma = \{0, 1, \$, \#, b\}, Q = \{s, \dots, q_8, h\}$$

state	0	1	\$	#	b
s	$(s, 0, +1)$	$(s, 1, +1)$	$(q_1, \#, -1)$	—	—
q_1	$(q_1, 0, -1)$	$(q_1, 1, -1)$	$(q_2, \$, +1)$	—	—
q_2	$(q_3, \$, +1)$	$(q_5, \$, +1)$	—	$(q_7, \#, +1)$	—
q_3	$(q_3, 0, +1)$	$(q_3, 1, +1)$	—	$(q_3, \#, +1)$	$(q_4, 1, -1)$
q_4	$(q_4, 0, -1)$	$(q_4, 1, -1)$	$(q_2, 0, +1)$	$(q_4, \#, -1)$	—
q_5	$(q_5, 0, +1)$	$(q_5, 1, +1)$	—	$(q_5, \#, +1)$	$(q_6, 1, -1)$
q_6	$(q_6, 0, -1)$	$(q_6, 1, -1)$	$(q_2, 0, +1)$	$(q_6, \#, -1)$	—
q_7	$(q_7, 0, +1)$	$(q_7, 1, +1)$	—	—	$(q_8, \$, -1)$
q_8	$(q_8, 0, -1)$	$(q_8, 1, -1)$	$(h, \$, 0)$	$(q_8, \#, -1)$	—

Figure 4.6: TM that produces $\$w\#w\$$ on input $\$w\$$

Solution to problem 11

N/A

Solution to problem 12

N/A

Solution to problem 13

N/A

Solution to problem 14

N/A

Solution to problem 15

N/A

Solution to problem 16

N/A

Solution to problem 17

Given an instance to HAMILTONICITY, we use the same instance as input to LONGEST SIMPLE PATH. The HAM question “Is there a Hamiltonian path?” is translated to “Show me the longest simple path”. All we need to do is to check whether the longest simple path found has exactly $n - 1$ edges where n is the number of vertices in the graph. If this is so, then we know that the graph is Hamiltonian, otherwise it isn't.

Why do we know it? Because a Hamiltonian path *is* a simple path of length $n - 1$, and vice versa. All this work can obviously be done in polynomial time, so that if we get the longest simple path in polynomial time, then we can answer in polynomial time whether the graph is Hamiltonian.

Solution to problem 18

N/A

Solution to problem 19

N/A

Solution to problem 20

N/A

Solution to problem 21

La oss konkret snakke om Handelsreisendes problem, men likevel forsøke å tenke på et generelt minimaliseringsproblem. Løsningen for å finne lengden av korteste reisevei har to faser: (1) Finn et intervall å søke i, og (2) gjør selve binærsøkingen.

(1) Vi vil her finne et tall MAX som (a) er minst like stort som den kortest mulige tur (generelt: er i ja-området for det tilsvarende desisjonsproblemet) og (b) Har et antall siffer som er begrenset av et polynom i forhold til input-størrelsen. I vårt tilfelle kan MAX velges som lengden av en tilfeldig valgt tur, ved å ta summen av de n lengste avstander eller (aller grovest, men helt OK) ved å ta lengste avstand ganger antall noder.

(2) Vi vet nå at lengden av den korteste turen må være et heltall mellom null og MAX. Vi kan så suksessivt innsnevre dette området, ved stadig å velge midtpunktet i det gjenværende området, og kalle desisjonsprosedyren med dette som 'spørsmålsverdi'. Vi velger så øvre eller nedre halvdel, ut fra svaret. Etter \log_2 MAX steg (som er ca 3 ganger antall desimale sifre i MAX) vil vi ha bestemt det nøyaktige minimum.

Solution to problem 22

Om vi bare er ute etter å sjekke om det finnes en Ham. Path. i en løkkefri rettet graf, så kan det gjøres ved å lage en 'topologisk sortering', altså en rekkefølge på nodene slik at alle kantene peker fra venstre mot høyre. Dersom det finnes en Ham. Path. i grafen, så kan det bare finnes én slik rekkefølge, og i denne må det da gå kanter fra venstre mot høyre mellom alle nabonoder i rekkefølgen. Man kan godt sjekke dette underveis mens man lager den topologiske sorteringa, og det hele koker da ned til følgende (husk at løkkefrie rettede grafer alltid må ha en node uten inngående kanter):

Vi fjerner én og én node fra grafen, men hele tiden slik at den vi velger for fjerning ikke har inngående kanter (fra gjenværende noder). Så langt er dette bare en standard algoritme for å lage en topologisk sortering, men hver gang før vi fjerner en node sjekker vi i tillegg at det er nøyaktig én node som ikke har inngående kanter (slik at vi bare har ett valg). Om det noen gang er flere, så finnes ingen Ham. Path (Hvorfor ikke?). Dersom vi har en teller over gjenværende inngående kanter i nodene, så kan dette gjøres i tid \mathcal{O} (antall kanter).

Skal vi finne lengste (rettede) vei, enten mellom to gitte noder eller i grafen totalt, så kan vi tenke oss at kantene er 'aktiviteter' med varighet lik lengden, og følge opplegget for å finne korteste gjennomføringstid etc. fra 'IN115-oblig-2'. Om vi frigjør oss fra denne IN115-oppgaven kan det være greiest å tenke i 'biler langs kantene' (som bruker tid lik lengden av kanten), bare med den vri at vi nå sender ut en bil fra en node når *alle* bilene er ankommet noden. Vi simulerer så forløpet i tid, og får da behov for en prioriteskø som inneholder de nodene som er ute og kjører i øyeblikket.

Om vi vil ha den totalt lengste veien setter vi i gang biler fra alle noder som ikke har innkommende kanter, og lengden blir da tidspunktet når alle biler er

kommet til alle noder. Dersom vi har oppgitt 'fra' og 'til' node må vi først skrelle bort alle noder som ikke kan nås fra 'fra'-noden, og deretter sende ut biler fra denne noden. Vi måler så tidspunktet når alle biler er kommet til 'til'-noden.

På grunn av tiden det tar å administrere prioritetskøen, blir tiden her $O(|E| \cdot \log(|E|))$, der $|E|$ er antall kanter (som også er det maksimale antall biler som kan være i gang).

Denne bilutsendelsen er selvfølgelig kritisk avhengig av at det ikke er løkker i grafen, for da blir det aldri spørsmål om bilene har vært to ganger innom samme node eller ikke. Dersom det *er* løkker, så ser det altså ut til å bli håpløst å kontrollere dette problemet (på 'polynomisk' måte).

Solution to problem 23

Selve det å studere forståelsen av og begrunnelsen for Lemma 3.1 overlates til gruppene eller selvstudium. Som forslag til 'dagligdagse' problemer kan vi jo foreslå følgende:

VERTEX COVER: Vi har et gatenettverk, med rette gater mellom kryss. Man skal sette ut vaktposter i kryssene slik at alle gater kan sees, og vi vil klare oss med så få vaktposter som mulig.

INDEPENDENT SET: En bedrift vil bygge et stort nytt lager, men det er en del typer varer som ikke kan stå i samme lager av sikkerhetsgrunner. For å få samlet mest mulig i det store lageret ønsker man å finne den største varemengden der det ikke er noen konflikter. Grafen har her kanter mellom de varer som ikke kan stå i samme lager.

CLIQUE: Det skal settes sammen en hurtigarbeidende komite med k medlemmer, og disse må kjenne hverandre på forhånd for at arbeidet skal komme fort i gang. Grafen blir her 'bekjentskapsgraf'.

Solution to problem 24

(3) på side 64 i G&J:

Det er her bare å la grafen H være en komplett graf med J (= min. antall noder i klikken) noder. Angående tilleggsspørsmålet så går svaret fram av kommentaren til Sub.Isom. på side 202 i G&J: Vi kunne f.eks. brukt Ham.Circuit i stedet, ved å la H være en 'ring med n noder'. Merk her at Sub.Isom. ikke forlanger at kantene i grafen H *nøyaktig* skal tilsvare kantene som går mellom den utvalgte del V av G , men at det skal finnes et *subsett* av kantene mellom V -nodene, slik at vi får en graf som er lik H .

Solution to problem 25

Vi bruker skjemaene direkte, og får:

$$(a \vee \neg b \vee y_1) \wedge (a \vee \neg b \vee \neg y_1) \wedge \\ (\neg a \vee y_2^1 \vee y_2^2) \wedge (\neg a \vee y_2^1 \vee \neg y_2^2) \wedge (\neg a \vee \neg y_2^1 \vee y_2^2) \wedge (\neg a \vee \neg y_2^1 \vee \neg y_2^2) \wedge \\ (\neg a \vee b \vee y_3^1) \wedge (\neg y_3^1 \vee \neg c \vee y_3^2) \wedge (\neg y_3^2 \vee d \vee y_3^3) \wedge (\neg y_3^3 \vee e \vee \neg f)$$

Merk her at man må ta nye ferske hjelpevariable hele tiden, også når vi går over til å omskrive en ny klausul.

Solution to problem 26

Selve grafbeskrivelsene overlates til tavletegning, etter mønster av figuren på side 55 i G&J eller figurene i forelesningsfoilene fra uke 7.

Kommentar 1: Det må innrømmes at oppgavestilleren i et svakt øyeblikk mente at SAT-instansen til høyre ikke var tilfredstillbar. Det *er* den selvfølgelig, og hvertfall så lenge man forlanger tre *forskjellige* variable i hver klausul, så må ikke-tilfredstillbare instanser av 3SAT ha en del klausuler. Som en liten

tilleggsnøtt kan man se på spørsmålet om *hvor mange klausuler* en slik 3SAT-instans minst må ha for å kunne være ikke-tilfredsrikkbar. Svaret står lenger ned.

Kommentar 2: Spørsmålet om man i en 3SAT-instans *forlanger* at det er tre forskjellige variable i hver klausul er litt uavklart i G&J. Det er klart at den intense behandlingen av “case 1” og “case 2” nederst på side 48 nettopp er gjort for å sikre at man får tre forskjellige variable i alle klausulene. Om man ikke forlanger dette kunne man f.eks. rett og slett transformere klausulen (a) til $(a \vee a \vee a)$. Grunnen til at man på side 48 er så nøye med å vise at man *kan* lage klausuler med tre forskjellige variable er antakeligvis at det i enkelte transformasjoner fra 3SAT kan være behagelig eller nødvendig å forutsette at det er tre forskjellige variable. Som nok en liten tilleggsnøtt kan man forsøke å avgjøre om dette gjelder transformasjonen $3SAT \propto VC$ beskrevet på side 55. Forsøk f.eks. å transformere $(a \vee a \vee a) \wedge (\neg a \vee \neg a \vee \neg a)$. Svar nedenfor.

Svar på smånøttene angitt over:

Hvor mange klausuler må en instans av 3SAT (med tre forskjellige variable i hver klausul) ha for å kunne være ikke-tilfredsrikkbar? Svaret er 8, og om vi begrenser oss til det tilfellet der det totalt bare er tre variable i instansen er det lett å se: Det går da nemlig an å lage 8 forskjellige klausuler, ved å variere negasjonen. Om vi har en instans som består av akkurat disse 8 klausulene ser vi lett at den ikke er tilfredsrikkbar. Har vi derimot færre enn 8 klausuler må det være én av de 8 kombinasjonene som “mangler”, og vi kan da velge verdier slik at alle literalene i denne er FALSE. Da vil minst én av literalene i alle de andre være TRUE.

At det bare blir verre om instansen har flere enn tre variable overlater vi til leseren å verifisere.

Ellers går transformasjonen i boka fra 3SAT til VC bra også om vi tillater samme variabel flere ganger i hver klausul.

Solution to problem 27

PARTITION \propto SUBSET SUM:

Denne går med restriksjon, ved å restrikttere SUBSET SUM (senere SS) til de instansene der $B = 1/2 \sum_{a \in A} s(A)$.

Når vi nå eksplisitt blir bedt om å sjekke at vi kan få fram *alle* instanser av PARTITION på denne måten (generelt er jo dette viktig), så kan vi observere at om man er *helt formell* så er dette faktisk ikke riktig (og det samme gjelder (6) og (7) på side 65). F.eks. skulle jo PARTITION-instansen (2, 3, 4) komme fra “SS-instansen” (2, 3, 4) med $B = 4.5$, men i SS-forlanges at B er et heltall.

Nå er det jo bare *opplagt gale* instanser av PARTITION som på denne måten ikke kommer fra noen legal SS-instans, og for å sjekke at dette forholdet ikke skaper noen dypere problemer, kan man gå tilbake til grunnfjellet og sjekke at vi kan lage en *transformasjon* fra PARTITION til SS. Denne bør først summere ‘size’-verdiene, og om dette blir et par-tall kan vi rett fram lage en SS instans med B lik halve summen.

Dersom summen er et odde tall vet vi jo at denne PARTITION-instansen skal ha svar nei, og da kan vi egentlig avbilde denne på en vilkårlig nei-instans av SS, ved f.eks. å la alle slike avbildes til SS-instansen (1, 3), med $B = 2$. Et alternativ er å først multiplisere alle ‘size’-verdiene i PARTITION med 2. Dette bevarer ja/nei, og nå kan vi alltid lett finne en ekte SS-instans etter den vanlige reglen.

Legg merke til at dette ville fungere like bra dersom det var noen opplagte ja-instanser av PARTITION som ikke framkom ved restriksjonen. Ser vi på det som transformasjon kunne vi da bare ha avbildet disse på en fast ja-instans av SS. Merk at uttrykket ‘opplagt nei-instans (eller ja-instans)’ generelt skal tolkes som ‘instanser som kan gjenkjennes og besvares i polynomisk tid’. Arbeidet

med å kjenne dem igjen og besvare dem må jo kunne gjøres som en del av den polynomiske transformasjonen.

SUBSET SUM \propto PARTITION:

Vi angir (som over) size-verdiene som ingår i en instans av PARTITION eller SS rett og slett ved å liste dem opp i en parentes. En instans av SS har dessuten en B -verdi. Anta at vi har gitt SS-instansen (s_1, \dots, s_n) med en gitt B . Studerer man side 62 ser man at man der legger til *to* nye elementer med velvalgte size-verdier. Vi lar derfor $S = s_1 + \dots + s_n$, og transformerer denne SS-instansen til følgende PARTITION-instans: $(s_1, \dots, s_n, S + B, 2S - B)$

At dette kan gjøres i polynomisk tid anses opplagt. Vi må så bevise at ja og nei bevares. Dette kan generelt vises på to hårfint forskjellige måter:

(a) vise at en ja-instans avbildes på en ja-instans og at en nei-instans avbildes på en nei-instans, eller:

(b) vise at en ja-instans avbildes på en ja-instans og at alt som blir avbildet på en ja-instans selv er en ja-instans.

Disse er jo logisk ekvivalente. Vi velger her den siste varianten (og det er vel det vanligste).

Anta at SS-instansen er ja-instans, altså at vi kan gjøre et utplukk U med sum B av verdiene (s_1, \dots, s_n) . Resten av verdiene har da sum $S - B$. Vi ser at instansen $(s_1, \dots, s_n, S + B, 2S - B)$ av PARTITION er nå en ja-instans, for vi kan plukke U og elementet $2S - B$ og vise at disse har samme sum som resten. U og $2S - B$ har summen $2S$, og resten har summen $2S - B + (S - B) = 2S$. Altså OK.

Anta så at PARTITION-instansen $(s_1, \dots, s_n, S + B, 2S - B)$ er en ja-instans. Vi kan da først observere at $S + B$ og $2S - B$ må ligge i hver sin 'partisjon', siden allerede de har sum $3S$, mens resten bare har sum S . I samme partisjon som $2S - B$, må det være et visst utplukk av s -verdier, og vi kaller dette U , og lar summen over dette være x . Summen av de resterende s -verdiene er da $S - x$. Siden summen over de to partisjonene er like må vi ha: $2S - B + x = S + B + (S - x)$ Dette gir umiddelbart $x = B$, og den opprinnelige SS-instansen må derfor være en ja-instans.

Transformasjon av den gitte SS-instansen gir PARTITION-instansen:

(12, 25, 17, 5, 22, 118, 125)

Solution to problem 28

a) Anta at vi har en algoritme som finner en maksimal matching (i betydningen: 'med så mange kanter som mulig') i en generell urettet graf. Vi kan da finne et 'edge cover' (kantoverdekning) med så *få* kanter som mulig som følger: Velg først en maksimal matching for den aktuelle grafen. Her vil hver kant dekke to noder, og ingen dekkes dobbelt. Bedre kan vi altså ikke gjøre det så langt. Man ser så at for å dekke resten av nodene må man bruke en kant på hver, for om man klarte å dekke to nye noder med én kant, så måtte dette representere en utvidelse av den maksimale matchingen, som er umulig. Vi kan lette sjekke at denne algoritmen blir polynomisk, dersom max. matching algoritmen er det.

Det er også slik at *enhver mindre* kantoverdekning måtte innholdt en større matching. Dette kan man se slik: La "overskuddet" til en kantoverdekning være det antall ekstra overdekningskanter (ut over én) som går inn mot hver node, summert over alle noder. (Dette kan også angis som: $2 * |E| - |V|$, der $|E|$ =antall kanter i overdekningen og $|V|$ =antall noder). Dersom den maksimale matchingen vi fant over hadde m kanter, og vi la på k kanter til slutt, så er overskuddet her lik k .

Anta så at vi fant en overdekning med én kant mindre, altså med $m + k - 1$ kanter. Siden vi da har fjernet *to* kantender, må overskuddet til denne være $k - 2$. Dette overskuddet kunne helt sikkert fjernes ved å ta bort $k - 2$ kanter fra

overdekningen, og da ville vi stå igjen med en matching med $m + 1$ noder. Dette er mot forutsetningen, altså finnes ingen mindre kantoverdekning.

Som eksempel å se på kan man f.eks. bruke en stjernegraf med tre armer, og en trekant i enden av hver arm (altså 10 noder).

NB: Vi må forutsette at den grafen vi arbeider med ikke har 'enslige noder'. For slike kan ingen kantoverdekning finnes.

Og til slutt: Når vi på denne måten kan finne en *minimal* kantoverdekning, kan vi også avgjøre ethvert spørsmål av typen "finnes en kantoverdekning med færre enn K kanter?".

b) Dette gjøres enkelt ved f.eks. et vanlig dybde først søk, og vi starter med en tilfeldig farge i startnoden. Når vi går langs en kant fra en node A til en node B , så er det alltid bestemt hvilken farge vi *må* ha i B , nemlig den motsatte av den i A . Dersom B allerede er behandlet (og derved fargelagt) sjekker vi at dette stemmer, og trekker oss tilbake. Ellers fargelegger vi den nå med den riktige fargen, og fortsetter søket. Et vanlig dybde først søk tar tid \mathcal{O} (antall kanter), altså polynomisk.

2-fargbare grafer kalles også ofte 'bipartite' grafer, og de kan også karakteriseres ved at de ikke inneholder løkker med et odde antall noder.

Litt løs filosofering: Det som gjør at vi her klarer å kontrollere 2-fargbarhet synes å være at med bare to farger så er det å vite hvilken farge det *ikke* skal være, også nok til å bestemme hvilken farge det *skal* være. Dette gjelder jo ikke om vi har tre eller flere farger, og dette er altså så alvorlig at problemet da faller over i \mathcal{NPC} . Det er også mange andre problemer der steget fra 2 til 3 gjør at det går fra \mathcal{P} til \mathcal{NPC} , f.eks. Exact Cover By 2-sets (X2C), og Exact Cover By 3-sets (X3C). Man må dog ikke tro at dette er noen generell regel.

c) Exact Cover By 2-sets er identisk med å spørre om det i en gitt graf går an å plukke en matching som dekker alle nodene (en såkalt "perfekt matching"). Man tolker da den totale mengden i X2C som en nodemengde, og lar hvert 2-set være en kant mellom noder.

Solution to problem 29

Nr. 3: Vi restrikerer til de instanser der $K = 1$, og får (alle instanser av) HC.

Nr. 4: Vi kunne også her restrikeret til HC, men siden det er mulig kan vi her, som en variasjon, bruke Clique: Vi legger først merke til at spørsmålet er stilt slik at G_1 og G_2 må ha like mange noder for at det skal være en ja-instans. Vi restrikerer derfor for det første til bare *slike* instanser, og videre til de instanser der G_2 inneholder en komplett graf med $J \leq |V_2|$ noder (og ellers bare har kantløse noder), og der $K = J(J - 1)/2 =$ antall kanter i G_2 . Da får vi Clique-problemet, med spørsmål om G_1 inneholder en klikk av (minst) størrelse J .

Nr. 5: Vi restrikerer til de instanser der $K = 2$ og $J = 2(\frac{1}{2} \sum_{a \in A} s(a))^2$, og får dermed PARTITION-problemet. Denne er litt mer fiklete, så la oss se på den som en transformasjon. Vi kaller vår restriksjon av MIN-SUM-OF-SQUARES-problemet for RESTR. Vi observerer først at om vi har en ja-instans av PARTITION, så er det opplagt at vi får en ja-instans av RESTR, ut fra den samme oppdelingen.

Anta så at vi har en ja-instans av RESTR. For å kunne konkludere med at disse size-verdiene også er en ja-instrans av PARTITION, er det et viktig poeng at når man skal dele et tall (her 'size-summen') i to deler og ta kvadratsummen av delene, så blir denne minst mulig når man deler i *nøyaktig to like deler*, og alle andre oppdelinger gir større kvadratsum (se under). Dermed vet vi at den oppdelingen som gjør at RESTR-instansen er en ja-instans, må dele size-summen i to like deler, ellers ville kvadratsummen vært større enn J -verdien i RESTR. Altså har vi også en ja-instans av PARTITION.

Begrunnelse for påstanden over: Anta at tallet vi skal dele er $2n$. Deler vi det i to like deler og tar kvadratsummen får vi $2n^2$. Bruker vi en annen oppdeling får vi $(n+m)^2 + (n-m)^2 = 2n^2 + 2m^2$, altså større.

Solution to problem 30

a) Vi kan f.eks. tenke oss at en større ekspedisjon må dele seg i to grupper for en periode, f.eks. for å få undersøkt et større område. Hver av gruppene må da ha en viss kompetanse, så som botanisk kunnskap, zoologisk, medisinsk, utsyrsteknisk, lokal terrengkunnskap, kokekunst, etc, etc. Mengdene i kolleksjonen C er da mengdene av de personer som har de forskjellige kunnskaper og kompetanser. For å få noe skikkelig problem ut av det, må vi tenke oss at hver person (eller hvertfall en del av personene) har kompetanse på flere områder. (Om det *ikke* er tilfelle vil problemet lett kunne løses i polynomisk tid. Hvordan??).

b) Som i boka (side 38, linje 6 nedenfra) definerer vi en *literal* til å være en logisk variabel eller en negert logisk variabel. Om vi har n logiske variable kan vi altså lage $2n$ forskjellige literaler over disse, og det er nøyaktig 3 literaler i hver klausul av en (NAE)3SAT-instans.

Transformasjonen fra NAE3SAT til Set Splitting (SS) går som følger: Mengden S i den SS-instansen vi lager lar vi bestå av alle literaler over variablene som forekommer i NAE3SAT-instansen. Kolleksjonen C i SS lar vi bestå av:

(i) Alle 2-literal-mengder av formen $\{u, \neg u\}$, der u er en variabel i NAE3SAT-instansen.

(ii) De 3-literal-mengder som tilsvarer klausulene i NAE3SAT-instansen.

Vi aksepterer her at denne transformasjonen kan gjøres i polynomisk tid (men skal eventuelt i et eksempel senere se på hva som er et rimelig svar om det eksplisitt er spørsmål etter argumentasjon for at en transformasjon er polynomisk)

Vi antar så at vi har en ja-instans av NAE3SAT, og vil vise at den produserte SS-instans er en ja-instans. I mengden S lar da de literaler som har verdien TRUE være den ene mengden S_1 , mens de som er FALSE utgjør mengden S_2 . Vi ser da umiddelbart at i mengdene (i) må ett element være i S_1 og ett i S_2 . Siden hver klausul har minst en variabel lik TRUE og en lik FALSE (ut fra def. av NAE3SAT), ser vi også uten videre at hver av mengdene i (ii) er splittet slik at minst ett element er i S_1 og minst ett er i S_2 .

Anta så at transformasjonen gir en ja-instans for SS. Dermed finnes lovlig splitting av denne, og vi kan velge en tilfeldig av mengdene (f.eks. S_1) og la literalene i denne være TRUE, og resten være FALSE. Vi ser da (fra at (i)-mengdene er lovlig splittet) at dette er en konsistent verdi-setting av variablene, og da (ii)-mengdene er lovlig splittet må også alle klausulene i NAE3SAT-instansen vi kom fra ha tilfredstilt kravet i NAE3SAT.

c) Transformasjonen fra 3SAT til SS går som følger: Mengden S i SS lar vi bestå av alle literaler over variablene i 3SAT-instansen, samt ett ekstra element vi kaller 'F'. Kolleksjonen C i SS lar vi nå bestå av:

(i) Alle mengder av formen $\{u, \neg u\}$, der u er en variabel i 3SAT-instansen (altså som over).

(ii) For hver av klausulene i 3SAT-instansen, en mengde på 4 elementer, nemlig elementet F, samt de tre literalene fra klausulen.

Anta så at vi har en ja-instans av 3SAT. I mengden S lar vi da de literalene som har verdien TRUE være den ene mengden S_1 , mens elementet F, sammen med de literalene som er FALSE lar vi være mengden S_2 . Vi ser som over at mengdene (i) er splittet riktig. Siden hver av mengdene i (ii) har minst en literal som

er TRUE (og derved ligger i S_1) og helt sikkert inneholder elementet F (som ligger i S_2), så er også disse riktig splittet.

Anta så at transformasjonen gir en ja-instans for SS. Dermed finnes lovlig splitting, og vi kaller den delen som inneholder F for S_2 , og den andre S_1 . Vi lar så literalene i S_1 være TRUE, og literalene i S_2 være FALSE. Vi ser da (fra at (i)-mengdene er riktig splittet, som over) at dette er en konsistent verdi-setting av variablene, og da hver av (ii)-mengdene må ha minst ett element i S_1 må også alle klausulene i 3SAT-instansen vi kom fra ha minst en literal som er TRUE. Dermed er 3SAT-instansen vi kom fra en ja-instans.

d) En algoritme for å løse problemet kan først undersøke om det finnes mengder i kolleksjonen C med ett element. I så fall er svaret opplagt 'nei' (!?). Om dette ikke er tilfelle (alle mengder har to elementer) lager man en graf der elementene i mengden S utgjør nodene, og der hver mengde i kolleksjonen C representeres som en kant. SS-spørsmålet kan da omformes til spørsmålet om denne grafen er 2-fargbar (altså om vi kan dele nodene i to grupper $S_1 =$ de røde og $S_2 =$ de blå, slik hver kant har en rød og en blå ende). Dette problemet har vi løst tidligere ved enkel graf-traversering.

e) Under (b) transformerte vi et kjent \mathcal{NP} -komplett problem til en instans av SS der alle mengdene i kolleksjonen C var av størrelse 2 eller 3. Om det restrikterte problem oppgaveteksten beskriver kalles 3SS, har vi altså rett og slett vist $\text{NAE3SAT} \propto 3\text{SS}$.

Solution to problem 31

Vi henviser til beskrivelse og figur i G&J, side 61. I vårt problem er altså $q = 3$, og antallet p av (desimale) sifre som må til for å lagre tall opp til 100 er 3 (altså $p = \lceil \log_{10}(100 + 1) \rceil = 3$). Tallene vi skal angi som 'size'-verdier til SUBSET SUM instansen må altså ha $3pq = 3 \cdot 3 \cdot 3 = 27$ siffer.

Hvert trippel skal dermed omarbeides til en slik size-verdi med 27 siffer (inklusive innledende nuller), og SUBSET SUM instansen for den gitte 3DM-instansen blir:

(a1, b2, c3):	001 000 000	000 001 000	000 000 001
(a1, b3, c2):	001 000 000	000 000 001	000 001 000
(a2, b2, c1):	000 001 000	000 001 000	001 000 000
(a3, b1, c3):	000 000 001	001 000 000	000 000 001
....	...		
Verdien av B:	001 001 001	001 001 001	001 001 001

Solution to problem 32

a) Anta at tallets verdi er n . Her vil selv de mest naive algoritmer være pseudopolynomiske. Feks. kan vi gå gjennom hvert av tallene $2, 3, \dots, n - 1$, og sjekke om de går opp i n . Dette blir $O(n)$ sjekker, og hver av disse kan gjøres i $o(\log n)$ tid (se pkt. (c)).

Faktisk er det slik at selv om man utfører testen på om ' x går opp i n ' ved suksessivt å trekke fra x , og se om man 'treffer null', så vil den fremdeles være pseudopolynomisk. Hver slik test vil jo ikke ta mer enn n subtraksjoner, og totalt vil det hele da bruke ikke mer enn $O(n^2)$ subtraksjoner (som hver går i tid $O(\log n)$).

Det er ingen kjent algoritme som løser dette problemet i beviselig polynomisk tid, men det finnes algoritmer som klarer tall på noen hundre desimale sifre

uten store problemer.

b) Det å summere tall tar bare tid proporsjonalt med antall siffer, så denne blir (ekte) polynomisk.

c) Om vi bruker vår vanlige hånd-divisjons-algoritme (og tester om resten er null) går dette i (ekte) polynomisk tid.

For interesserte: Ser vi litt nøyere på det å utføre " $n : m$ ", der $n > m$ (eller er resten gitt), og tallene har hhv. sn og sm siffer, så vil vi få omtrent $sn - sm$ hovedsteg, og hver av disse vil innebære det å finne neste siffer (maksimalt forsøke med $9 \cdot m, 8 \cdot m, \dots, 1 \cdot m$), samt en subtraksjon. Maksimalt blir dette omtrent $(sn - sm) \cdot 10 \cdot sm$ enkeltsteg, som er $O(sn \cdot sm)$. Tidsforbruket til divisjon er altså grovt regnet kvadratisk i tallengden.

d) La de to tallene være a og b , $a \geq b$. Nesten enhver tenkelig algoritme vil være pseudopolynomisk, f.eks. å se på $b, b - 1, \dots, 2$, og sjekke om de går opp i både a og b . Ser vi på Euklids algoritme så går den slik (vi tenker oss $a = c_1$ og $b = c_2$):

$$a \bmod b = c_3, b \bmod c_3 = c_4, c_3 \bmod c_4 = c_5, \dots$$

Her stopper man når tallene ikke lenger forandrer seg, og har da det ønskede tall. Det viser seg her at c_i alltid er mindre enn halvparten av c_{i-2} , og lengden av sekvensen blir derved ikke større enn omtrent $2 \cdot \log a$. Dermed er algoritmen (ekte) polynomisk.

For interesserte: Beviset for halverings-setningen over er ikke vanskelig. Vi vet at $a \bmod b \leq b - 1$ og $a \bmod b \leq a - b$. Man skiller så mellom de to tilfellene $a - b \leq b - 1$ og $a - b > b - 1$, og i begge tilfellene følger det greit at $a \bmod b \leq (a - 1)/2$. Forsøk selv!

e) Det var her ment $T \geq 0$. Vi ser at for x hel og $x \geq 0$, så er $x^5/7 + x^3 \geq x$, og dermed må den ønskede verdi ligge mellom 0 og T (inklusive). Siden funksjonen er monotont voksende kan verdien finnes ved binær søking, og en slik algoritme blir (ekte) polynomisk.

f) (Også her antas $T \geq 0$). Svaret her er jo rett og slett $x = 2^T$, og problemet her blir at dette tallet i to-tall-systemet har $T + 1$ siffer, og i ti-tall-systemet omtrent tredjeparten så mange. Det har altså $O(T)$ siffer, og hvertfall i to-tall-systemet lar dette tallet seg lett skrive ut i tid $O(T)$ (det er jo bare en 1-er med T nuller etter). Dermed kan problemet løses lett i pseudopolynomisk tid, og om vi *forlanger* at tallet skal skrives ut i med siffer i et passelig tallsystem så kan det opplagt ikke løses i polynomisk tid. Om vi derimot aksepterer en utskrift av typen "svaret er en ener med T nuller etter", så kan det jo typisk løses i polynomisk tid. Problemet må vel dermed klassifiseres sammen med de omtalt i nest siste avsnitt på side 11 i G&J.

Solution to problem 33

a) Man kan tenke seg at hvert element (s_i, v_i) er ting man kunne ønske å ha med seg i sekken, der s_i er vekten og v_i er "verdien" (altså, hvor viktig det er å ha det med). Tallet B vil da representere den maksimale vekten sekken kan ha, mens K er den verdi-sum man minst må ha med. Om man går ut over desisjonsproblemer ville vel dette naturlig være et optimaliseringsproblem der man vil maksimere K .

b) Alle $s_i = s$:

Da er B -kravet rett og slett at vi kan ha med B/s stykker, uten hensyn til hvilke.

Vi plukker så de B/s elementene med størst v -verdi. Om v -summen for disse er minst K er svaret 'ja', ellers er det 'nei'. Dette kan gjøres i polynomisk tid.

Alle $v_i = v$:

Nokså tilsvarende forrige punkt. K -kravet blir da rett og slett at vi må ha med minst $\lceil K/v \rceil$ stykker, uten hensyn til hvilke. Vi tar med de $\lceil K/v \rceil$ elementer med minst s -verdier, og om summen av disse ikke overgår B er svaret 'ja', ellers er svaret 'nei'. Dette kan gjøres i polynomisk tid.

Alle $v_i = 1$ eller 2 :

Det ser ut til at denne kan løses i polynomisk tid, ved å først sortere elementene stigende på 'verdi per størrelse' ('verditetthet'), og så ta med de med størst verditetthet (minst 'størrelse per verdi') inntil vi når verdien $K \pm 1$. Man kan så, om nødvendig, foreta en lokal enkel justering, og dermed vite at vi har oppnådd den ønskede verdi K med så liten total størrelses-sum som mulig. Det er da bare å sammenlikne denne s -summen med B .

Alle s_i og $v_i = 1, 2, \dots, 10$:

(Denne oppgaven ble litt mer komplisert enn oppgavestilleren i utgangspunktet hadde tenkt seg da vi *ikke* har noen begrensninger på B og K . Oppgaven ble imidlertid *desto mer interessant!*

). Vi observerer først at dersom $K > 10n$ så *må* svaret være 'nei', og videre at om $B > 10n$ så vil B -grensen aldri komme inn i bildet, og svaret vil bare avhenge av om v -summen overgår K . Altså, ved først å teste på disse betingelsene kan vi i polynomisk tid besvare spørsmålet dersom vi har et av disse tilfellene.

For de resterende tilfellene er jo B og K begrenset av $10n$, og ut fra svaret på punkt (f) under ser vi at tiden vil være $O(n \cdot 10n \cdot 10n)$, som er $O(n^3)$. Dette er (ekte) polynomisk, da ethvert fornuftig Length-mål må vokse minst lineært med n .

Vi kan imidlertid også se dette siste som et tilfelle av det som er diskutert på side 95 i G&J, mellom 'Observation 4.1' og 'Observation 4.2'. Anta at $Max(I)$ er definert som den største tallverdien i instansen (G&J's standard-definisjon for Max). Vi ser da at i vårt tilfelle har vi (etter at de to lette tilfellene er skilt ut) restrikkert alle tall som inngår i instansen slik at $Max(I) \leq 10n \leq 10 \cdot Length(I)$. Altså er $Max(I)$ polynomisk begrenset i forhold til $Length(I)$, og da gjelder uten videre at det som er pseudopolynomisk også er polynomisk.

Til slutt nok en variant man kan lure på:

Hva skjer om vi begrenser n oppad med en fast grense (f.eks. 100), men tillater at *tallene* vokser så mye de vil. Kan problemet da løses i polynomisk tid? (Svar står på slutten av denne oppgaven)

c) For å ta det siste først: Svaret er 'ja' om det finnes en TRUE i området: $i = n$, $j = 0, \dots, B$ og $k = K, \dots, V$, der $V = v_1 + \dots + v_n$. Dette dekker jo nettopp alle mulige s -summer og v -summer vi kan få fra lovlige utplukk blant alle n elementene.

Så til det første: Selv om vi bare er interessert i området angitt over, må vi (ut fra det som er gjort på side 90 i G&J) være forberedt på måtte beregne oss gjennom alle indekseringer fra 1 for i og fra 0 både for j og k . Dermed må vi totalt regne med å måtte fylle ut tabellen for $i = 1, \dots, n$, $j = 0, \dots, B$ og $k = 0, \dots, V$.

d) Vi antar alle $s_i \leq B$, ellers kan de 'kastes' (kan aldri komme med i noe lovlig utvalg)!

```
t(1, j, k) =      TRUE   for j = k = 0
                TRUE   for j = s1, k = v1
                FALSE  ellers
```

e)

```

t(i, j, k) =      TRUE   dersom  t(i-1, j, k) = TRUE
                TRUE   dersom  j-si }= 0, k-vi }= 0 og
                        t(i-1, j-si, k-vi) = TRUE
                FALSE  ellers

```

Enten kunne vi gjøre slik utplukk før (s_i, v_i) ble lagt til, eller vi kan gjøre det ved på inkludere det nye elementet. Andre muligheter finnes ikke.

Vi kaller operasjonene $D(1, j, k)$ og $E(i, j, k)$, og kan skrive det som følgende programskisse:

```

for j:= 0 to B do
  for k:= 0 to V do  D(1, j, k);

for i:= 2 to n do
  for j:= 1 to B do
    for k:= 1 to V do  E(i, j, k);

for j:= 1 to B do
  for k:= K to V do
    if t(n, j, k) then {svaret er `ja`};
    {ellers er svaret `nei`};

```

Testingen kan også (for noen tilfeller mer effektivt) gjøres ved at vi på slutten av operasjonene $D(1, j, k)$ og $E(i, j, k)$ legger setningen:

```

if k <= K and t(i, j, k) = TRUE then {svaret er `ja`};

```

f) Operasjonene 'D' og 'E' tar tid $O(1)$. Max tid blir derfor $O(nBV)$. Dette er pseudopolynomisk. (Om man vil se det i forhold til en $Max(I)$ -variant som angir det maksimale tallet i instansen, kan man bare sette inn at $V \leq n \cdot Max(I)$.)

Svar til ekstraoppgave gitt under pkt b over

(Hva om antall (s, v) -par n er begrenset av en fast øvre grense?):

Vi skal her merke oss at instansene kan bli så store de bare vil, ved at tallene blir store. Denne er løsbart i polynomisk tid. Vi konstaterer rett og slett at antall utplukk fra mengden er begrenset av det faste tallet 2^n , og at det er en opplagt polynomisk operasjon å sjekke for hver av disse om de oppfyller kravene til s -sum og v -sum. For $n = 100$ må vel imidlertid denne algoritmen sies å ha fortrinnsvis akademisk interesse.

Solution to problem 34

a) Dette er \mathcal{NP} -komplett, men kan løses i pseudopolynomisk tid. Det er \mathcal{NP} -komplett fordi vi kan transformere det rene SUBSET SUM: $(r_1, r_2, \dots, r_k), B'$ til dette ved å transformere til $(r_1, r_2, \dots, r_k, B' + 1, \dots, B' + 1), B'$ (med f.eks. k elementer med verdi $B' + 1$ lagt til. Disse vil aldri kunne være med i noe utplukk med sum B').

Videre kan det løses i pseudopolynomisk tid ved å se hver for seg på utplukk fra de ca. $n/2$ forskjellige mulige intervaller av lengde ca. $n/2$. (Mer nøyaktig: Om n er partall skal man se på nøyaktig $n/2$ forskjellige intervaller, hver med lengde $n/2 + 1$. Om n er odde blir det $(n + 1)/2$ intervaller av lengde $(n + 1)/2$.)

b) Dette er \mathcal{NP} -komplett i sterk forstand, for om vi begrenser alle s -verdiene til bare verdien 1 så er problemet fremdeles \mathcal{NP} -komplett. Da har vi nemlig VERTEX COVER med $K = B$.

PS: Dette problemet kan sees som en 'veiet' variant av et 'allerede' \mathcal{NP} -komplett problem, og slike problemer blir stort sett \mathcal{NP} -komplette i sterk forstand.

c) Dette kan løses i polynomisk tid, da det bare er et polynomisk antall forskjellige intervaller vi kan velge, nærmere bestemt $n(n-1)/2$ stykker (altså begrenset av n^2). Det å sjekke for hver av disse om summen er lik B er opplagt polynomisk. (En gruppelærer foreslo at det måtte finnes en morsom 'slange-algoritme' for å løse dette. Det kan man jo filosofere over.)

Dersom vi tillater inntil tre intervaller, kan problemet fremdeles løses i polynomisk tid. Antall forskjellige måter vi kan velge 3 intervaller vil nemlig opplagt ikke overstige n^6 , i og med at intervallene er gitt ved sine 6 endepunkter. Vi ser derved at om vi forlanger, for en fast M , at tallene skal velges slik at de danner inntil M sammenhengende intervaller fra den gitte sekvensen, så kan problemet løses i polynomisk tid.

Solution to problem 35

Den "generelle" metoden i boka for å vise at et søkeproblem X er \mathcal{NP} -lett (selv om de ikke innfører denne betegnelsen før etter at metoden er vist) går ut på først å velge et fornuftig "mellomproblem" XE ("extension"-problem), som på en passelig måte spør om en et forslag til "halvferdig løsning" kan utvides til en fullverdig løsning. For å kunne vise \mathcal{NP} -letthet må dette problemet være i \mathcal{NP} , og det er som regel svært enkelt å få til. Man må imidlertid passe seg for varianter der ja-svar *ikke* uten videre kan bevitnes, som f.eks. "Kan denne by-sekvensen utvides til en *minimal* tur?". (Hvordan bevitne at en tur er *minimal*?) I stedet kan man bruke "Kan denne by-sekvensen utvides til en tur som ikke er lengere enn B ?". Her kan jo et ja-svar klart bevitnes.

For å kunne bruke denne metoden til å "løse" optimaliseringsproblemer må man derfor først skaffe f.eks. lengden av korteste tur, og får derfor to faser (som begge bruker en tenkt 'orakelprosedyre' som løser XE i polynomisk tid): (1) Finn den optimale verdi som kan oppnås (ved binærsøkning) (2) Bygg opp en løsning som har denne verdien. For Traveling Salesman blir dette i boka gjennomgått ved at skrittene gjøres i rekkefølgen (1) og deretter (2), mens vi på forelesningen tok (2) først. Vi har også sett på fase (1) i oppgave 2.1 tidligere.

Merk altså at om vi har vist at $X \propto_T XE$ og at XE er i \mathcal{NP} , så vet vi at X er \mathcal{NP} -lett. Det er f.eks. ikke nødvendig å vise at XE er \mathcal{NP} -komplett.

a) Dette problemet (HCS) er ikke et optimaliserings-problem, og vi trenger derfor bare én fase. Spørsmålet i 'HCE' vil være "kan vi utvide den gitte node-sekvensen S til en Hamiltonsk løkke (i den gitte grafen)?".

For å vise $HCS \propto_T HCE$ kan man starte prosedyren for HCS med et kall på HCE, med en sekvens med en tilfeldig node (eller med tom nodesekvens), og derved få svar på om det finnes en Hamiltonsk løkke. Om slik finnes kan man bygge den opp ved gjentatte kall, nøyaktig som for Trav. Salesm.

b) Her kan vi bruke $CLE =$ "Kan jeg utvide den gitte nodemengden C til en klikk med minst J noder (i den gitte grafen?)". Her har vi et optimaliseringsproblem, og får da to faser: (1) Bruk binærsøkning i området fra 1 til $n =$ *antall noder* til å finne størrelsen J^* på en maksimal klikk. Vi må under dette søket kalle CLE -prosedyren med tom C ". (2) Bygg opp en klikk med nøyaktig J^* noder ved i hvert skritt å foreslå alle mulige en-node-utvidelser av C for prosedyren CLE , og velge en som gir positivt svar.

Man kan her merke seg at vi i (1) egentlig ikke behøvede å bruke binærsøkning, da det å gjøre noe n ganger ikke ville gjøre algoritmen eksponensiell. Det

kommer altså av at vi her kjenner en så lag øvre grense for J^* . Ofte (f.eks. i (c) under) vil vi oppleve at denne øvre grensen er eksponensiell i forhold til instanslengden, og *da* blir det viktig å bruke binærsøking.

c) Her kan vi bruke KSE med spørsmålet: "Kan jeg utvide et gitt utplukk U til et 'lovlig' utplukk med v -sum minst K ". Her får vi også to faser, og i fase (1) må vi binærsøke etter maksimal K i området opp til summen av alle v -verdier. Denne summen er eksponensiell i forhold til instanslengden, så vi *må* bruke binærsøking. Fase (2) blir helt rett etter nesa.

d) Dette er en slags dobbelt-optimering: Først skal vi maksimalisere antall noder, og siden minimalisere vektsom. Slike problemer kan vanligvis gjøres til "enkeltoptimeringer" ved å lage en ny vektning som inneholder begge de gamle, og der det mest signifikante kriteriet er gitt en så stor faktor at dette alltid får første-prioritet, og der alle ønsker trekker i samme retning. Dette kan f.eks. gjøres ved at vi legger nye vekter u_i på nodene, definert ved: $u_i = (V + 1) - v_i$, der $V = \sum v_i$, der summen er tatt over alle noder ($u_i = V - v_i$ gir tull om G har én node).

Da må alltid k noder ha større vektsom enn $k - 1$ noder:

- min ny vektsom av k noder er: $(V + 1)k - V = V(k - 1) + k$

- max ny vektsom av $k - 1$ noder er: $((V + 1) - 1)(k - 1) = V(k - 1)$

Med de nye vektene u_i kan vi løse dette "tradisjonelt": E-spørsmålet vil da være: "Kan jeg utvide dette node-utplukket til et uavhengig utplukk med u -vektsom minst K ". Den innledende binær-søkninga må her søke i området opp til $U = \sum u_i = \sum((V + 1) - v_i) \leq \sum V = n \sum v_i$ (alle summer tatt over alle noder).

Antall søkesteg er derfor begrenset av: $\log(n \sum v_i) = \log n + \log(\sum v_i)$

Et passelig lengdemål for instanser kan være: $Length(I) = n + \log(\sum v_i)$, og antall søkesteg er jo faktisk allerede mindre enn dette. Søkefasen blir derfor polynomisk i tid, også med den nye definisjonen av node-vektene.

Solution to problem 36

(A) Tøv! \mathcal{NP} -hard betyr "minst like vanskelig som de i \mathcal{NPC} , men gjerne vanskeligere". Så selv om \mathcal{NP} -komplette problemer kan løses i pol. tid så kan det finnes enda vanskeligere problemer i \mathcal{NPH} .

(B) Perfekt uttalelse!

(C) Ja, som det framgår av side 158 i G&J, så er dette ikke avklart. Generelt er det forskjell på $A \propto B$ og $A \propto_T B$, men det er ikke kjent om dette også gjelder om man holder seg innenfor \mathcal{NP} . Muligheten er altså til stede for at de \mathcal{NP} -harde problemene innenfor \mathcal{NP} er en større klasse enn de \mathcal{NP} -komplette.

(D) Tøv! Det at det kan løses i polynomisk tid betyr jo også at det kan bevitnes i polynomisk tid, med tomt vitne.

(E) Tøv! F.eks. tilfredstiller alle \mathcal{NP} -komplette problemer dette.

(F) Igjen er dette svært så uavklart! For det første vil utsagnet være sant dersom det viser seg at $\mathcal{P} = \mathcal{NP}$, men anta nå at det ikke er tilfelle. Vi har definert \mathcal{NP} som de desisjonsproblemer der et ja-svar kan 'bevitnes i polynomisk tid'. For de \mathcal{NP} -komplette problemene tror man at et nei-svar *ikke* kan bevitnes i polynomisk tid, men heller ikke dette er avklart (men om det gjelder for ett i \mathcal{NPC} , så gjelder det for alle). Om vi antar at slike nei-svar ikke kan bevitnes i polynomisk tid, kan vi snakke om de 'co- \mathcal{NP} -komplette', nemlig de \mathcal{NP} -komplette problemer der spørsmålet er negert, og rent formelt vil disse da ligge utenfor \mathcal{NP} . Problemene i co- \mathcal{NPC} er imidlertid opplagt \mathcal{NP} -lette, og vil i så fall gjøre utsagn (F) galt.

Forskjellen mellom \mathcal{NP} og co- \mathcal{NP} er imidlertid av svært formalistisk art, og det interessante spørsmål er kanskje om det finnes \mathcal{NP} -lette desisjonsproble-

mer som *hverken* ligger i \mathcal{NP} eller i $\text{co-}\mathcal{NP}$. Dette problemet har jeg ikke funnet ut noe om, og kanskje har heller ikke dette noe kjent svar.

Solution to problem 37

Denne algoritmen står altså omtalt i boka side 134 i G&J. For tydelighets skyld er det engelske "maximal" oversatt med "ikke-utvidbar".

a) La M (en kantmengde) være en ikke-utvidbar matching, og la V være dens mengde av endenoder. Anta at det finnes kant k som ikke er "dekket" av V . Da er ingen av endenodene av k "berørt" av noen kant i M , og M må dermed kunne utvides med kanten k . Dette er mot forutsetningen om at M er ikke-utvidbar. Altså dekker nodemengden V alle kanter i grafen.

b) La M være en matching, og la V være en nodeoverdekning. For hver kant i M må minst en av endenodene være med i V , og siden hver kant i M har sitt helt separate sett med endenoder, må en nodeoverdekning ha minst $|M|$ noder.

c) Viser først $R_A \leq 2$ og $R_A^\infty \leq 2$:

Vi antar at vi har en graf G , og har laget en ikke-utvidbar matching M . Vi har da ut fra (a) og (b):

$$|M| \leq \text{opt}(G) \leq A(G) = 2|M|$$

- Den første ulikheten tilsvare (b).

- Den andre ulikheten sier at det optimale ikke er større en det som algoritmen gir (vi har et minimaliseringsproblem).

- Likheden til venstre stammer fra definisjonen av algoritmen.

Den første ulikheten gir $2 \cdot |M| \leq 2 \cdot \text{opt}(G)$, og sammen med den siste likheten gir dette $A(G) \leq 2 \cdot \text{opt}(G)$. Dette gjelder for alle G , og viser dermed at $R_A \leq 2$ og $R_A^\infty \leq 2$.

For å vise $R_A \geq 2$, holder det å finne et tilfelle G der $A(G) \geq 2 \cdot \text{opt}(G)$. Om vi klarer å finne tilfeller med så stor $\text{opt}(G)$ vi bare vil med denne egenskapen, har vi også vist $R_A^\infty \geq 2$ (Dette er imidlertid ikke *nødvendige* betingelser, se pkt (d) under).

Det første er greit, da vi kan velge G som grafen med to noder med en kant mellom. Vi ser her at $A(G) = 2$, mens $\text{opt}(G) = 1$.

For å lage en så stor graf vi bare vil med denne egenskapen. er det bare å oppmultiplisere denne grafen. Altså: G er en graf med m kanter som ikke har noen felles endenoder, og dermed $2m$ noder. Dette gir $A(G) = 2m$, mens $\text{opt}(G) = m$.

Her vil noen kanskje innvende at dette er "juks" siden grafen ikke er sammenhengende. Det er det imidlertid *ikke*, siden VC-problemet ikke sier noe om at grafen skal være sammenhengende. Det er imidlertid ikke noe problem å lage et eksempel der det samme gjelder i en sammenhengende graf. Se f.eks. på grafen med $2m$ noder, som er bundet sammen til en lineær struktur ved $2m - 1$ kanter. I "worst case" vil her M bli valgt som annenhver kant, og da vil alle de $2m$ nodene bli valgt av algoritmen. Vi kan imidlertid lett dekke alle kantene med bare å ta annenhver node, altså med m noder.

d) Vi skal vise at selv om vi legger inn et "styrkesteg", så kan vi fremdeles få $A(G) = K \cdot \text{opt}(G)$, med K så nær 2 man bare måtte ønske og med så stor $\text{opt}(G)$ man måtte ønske. Til dette bruker vi en graf som ser ut som følger:

I midten har den en "liggende stige", med $2m$ trinn. Denne inneholder $4m$ noder, og $6m - 2$ kanter. I tillegg har vi to "ekstranoder", en over og en under stigen. Disse har kanter til alle "trinnender" på sin side av stigen, altså $2m$ kanter fra hver. I alt er det altså $4m + 2$ noder. TEGN!

I verste fall velges alle matchingkantene "langs stigen", mellom trinn 1 og 2, mellom trinn 3 og 4 osv., på begge sider av stigen. Dette gir $2m$ kanter i matchin-

gen. På grunn av kantene til ekstranodene vil vi her ikke få strøket *noen* noder i strykesteget. Dermed får vi $A(G) = 4m$.

Vi kan imidlertid her dekke alle kantene ved $2m+2$ noder, nemlig med begge ekstranodene, samt en kant fra hvert trinn, annenhver oppe og nede (Dette er antageligvis også det optimale, uten at dette er viktig). For stor nok m blir derfor $R_A(G) = A(G)/opt(G) \geq 2 - 2/(m+1)$, og dette kan vi for stor nok m få så nær 2 vi bare vil (og vi kan også få det til med så stor $opt(G)$ vi bare vil).

e) Dette vil altså være et forsøk på å sikre seg "heldigste tilfelle" når matchingen velges i algoritmen over, men forsøk på å sikre dette i polynomisk tid vil (antageligvis ?) slå feil. Dette problemet er nemlig (i en desisjonsvariant) å finne helt direkte i listen over \mathcal{NP} -komplette problemer, side 192: "Minimum Maximal Matching".

Solution to problem 38

a) Oppdelingen i U_1 og U_2 er jo nettopp valgt slik at $t(U_2)$ er så liten som mulig, under den begrensning at $t(U_1) \leq t(U_2)$. Oppdelingen U'_1 og U'_2 er en annen oppdeling som tilfredstiller $t(U'_1) \leq t(U'_2)$, og da må jo $t(U_2) \leq t(U'_2)$.

Ser vi på vårt optimaliseringsproblem så er jo også $opt(I) = t(U_2)$ og $A_K(I) = t(U'_2)$, og siden det er et *minimaliserings*problem er det jo betryggende at $opt(I) \leq A_K(I)$.

b) $R_{A_K} = A_K(I)/opt(I)$. Dette kan også skrives på følgende måter:

$$R_{A_K} = \frac{A_K(I)}{opt(I)} = 1 + \frac{A_K(I) - opt(I)}{opt(I)} = 1 + \frac{t(U'_2) - t(U_2)}{t(U_2)} = 1 + \frac{t(U_1) - t(U'_1)}{t(U_2)}$$

c) Vi vet at $opt(I) = t(U_2) \geq T/2$. Med den oppgitte ulikheten her vi derfor:

$$R_{A_K} = 1 + (t(U'_2) - t(U_2))/t(U_2) \leq 1 + 2ank/T$$

Vi kan derfor oppnå $R_{A_K} \leq 1 + 1/k$ ved å sette $K = T/(2ank)$.

Tiden (regnet i antall "steg" i den dynamiske programmering) blir for fullstendig løsning av problemet $O(nT)$. Når vi har skalert ned alle T -verdiene med K blir dette $O(nT/K) = O(n \cdot 2ank)$, og da a er en konstant er dette $O(n^2k)$. Denne er polynomisk både i *Length* (som hvertfall vokser polynomisk med n) og k . Altså har vi et fullstendig polynomisk tilnærmingsskjema.

Solution to problem 39

N/A

Solution to problem 40

The algorithm picks three and three vertices at random until a triangle subgraph is found or all possible 3-vertex sets are tried. If the worst-case behaviour is important, then the algorithm can keep track of previously tried 3-vertex sets, so that it doesn't try the same set many times.

a) In the worst case we have to go through all 3-vertex sets before concluding that there are no triangle subgraphs. The number of 3-vertex sets on a graph with n nodes is $\binom{n}{3} = n(n-1)(n-2)/2 = \mathcal{O}(n^3)$.

b) We assume a random graph model with the edge probability $p = 1/2$. The probability that a random 3-vertex set is a triangle subgraph is then $(1/2)^3 = 1/8$. The expected number of 3-vertex sets examined before a triangle subgraph is found, is $\sum_{i=1}^{\infty} i(1 - 2^{-3})^{i-1} 2^{-3}$. A calculation along the lines of the

3-COLORABILITY example given in lecture 11, should give the answer $1/8$. So we need to check only 8 3-vertex sets on average before finding a triangle subgraph.

This average-case performance can be improved even further by the following little trick: We start by picking a random vertex v . Then we pick two of its neighbours s and t and check whether (s, t) is an edge. If (s, t) is an edge, then we have found a triangle subgraph. Otherwise we repeat the procedure.

The expected number of trials is now 2, since the probability of the 3-vertex set being a triangle subgraph is depending only on (s, t) being an edge or not, which is a $1/2$ probability. We already know that (v, s) and (v, t) are edges.

Solution to problem 41

As a rough estimate we say that the number of people on earth is $5 \cdot 10^9 \leq 10^{10}$, and that the number of seconds in a year is $60 \times 60 \times 24 \times 365 \leq 10^{2+2+2+3} = 10^9$. For sake of simplicity we can assume that one experiment (ranging from 1 to 100 coin tosses, depending on the number of successive heads) takes 1 second.

This gives a total of $10^{10+5+9} = 10^{24}$ trials. The probability of getting 100 heads in a row is 2^{-100} which is approximately 10^{-30} since $2^{10} = 1024 \approx 10^3$.

The probability of somebody getting 100 heads in a row in the 10 000 years history of human kind is then roughly $\frac{10^{24}}{10^{30}} = 10^{-6} = 1/1.000.000$, which is not a very high probability.

Solution to problem 42

N/A

Solution to problem 43

Heuristic H1 for finding the shortest path between two nodes s and t in a network:

```
Start in node s;
Until you reach t do:
  At each step extend the path by
  choosing the lowest-weighted edge.
```

Imagine that there are two possible paths from s to t , with weights: $1 + 10$ and $2 + 2$. The greedy algorithm will choose the $1 + 10$ path because it has a cheaper first edge. But $1 + 10$ is obviously longer than $2 + 2$. So H1 is no proper algorithm. In fact it is easy to see that doesn't always find a solution, even if there is one.

What is even worse is that the solution returned can be infinitely large compared to the optimal solution, because 10 is just a random number. It could be $1 + 1000000000$ and H1 would still return this path as its approximate solution. The error grows as a function of $|n|$, where $|n|$ is the length of the input. Because of the binary coding, the relative error can be as big as $\mathcal{O}(2^{|n|})$. This is certainly not a constant.

Solution to problem 44

N/A

Solution to problem 45

N/A

Kapittel 5

Kommentarer til læreboka G&J

5.1 Innledning

Dette er en serie notater som ble skrevet av Stein Krogdahl til kurset IN210 våren 1990. De er forsøk på å utdype og understøtte stoffet, i forhold til framstillingen i læreboka: Garey and Johnson: *Computers and Intractability* (senere angitt som 'G&J'). Stoffet er delt opp som kommentarer til de forskjellige kapitlene i læreboka, men det vil ikke si at kommentarene strengt holder seg til stoff bare om dette kapitlet.

Det må understrekes at disse notatene her utgis i nokså rå form, mer eller mindre direkte slik de ble skrevet våren 1990. Pensum og terminologi har forandret seg en del etter den tid, men kommentarene er forhåpentligvis fortsatt til hjelp for de som synes at G&J er litt tung å få tak på. Den største forandringen i pensum siden 1990 er at kurset nå omhandler flere emner, slik som uavgjørbarhet og tilfeldighetsbaserte algoritmer. Dermed har vi ikke tid til å gå så mye i dybden i stoffet om \mathcal{NP} -komplekthet – og da spesielt \mathcal{NP} -komplekthetsbevisene.

5.2 Kommentarer til Kapittel 1

G&J: Kapittel 1.1

Kapittel 1.1 er jo bare ment som litt innledende motivasjon til boka, og skulle være grei lesing. Den siste figuren understreker altså

dette at det avgjørende bevis i den teorien vi skal se på mangler, men at vi likevel kan ha god nytte av denne teorien.

Begrepet '(inherently) intractable' brukes altså allerede her, og vi skal ikke ha noen standard oversettelse av dette, men kanskje 'umedgjørlig' eller 'vrangt' kunne passe. Ordet presiseres siden (side 8) til å beskrive problemer som ikke kan løses ved noen "polynomisk tid algoritme", altså ikke ved noen algoritme som kan garanteres å løpe i polynomisk tid for alle lovlige data.

G&J: Kapittel 1.2

Her beskrives altså hva som menes med et *problem*, og hvordan boka typisk vil beskrive disse (ved en 'instansbeskrivelse' og et spørsmål). Foreløpig er det snakk om generelle problemer, der en løsning kan være hva som helst. Siden blir det ofte (men ikke alltid!) snakk om problemer der de aktuelle løsninger bare er 'ja' eller 'nei'. Disse kalles 'desisjonsproblemer' eller 'ja/nei-problemer'.

Et problem består altså av mange 'instanser', som er fullt spesifiserte spørsmål. Selv om det ikke presiseres her, blir det siden avgjørende at et problem

har uendelig mange (og derved 'uendelig store') instanser. Dette stikker i at O-notasjon (og derved begrepet 'polynomisk tid') bare har mening når vi kan la problemstørrelsen vokse mot uendelig.

Merk at en 'algoritme som løser et problem' her er definert slik at den *alltid* finner en løsning etter endelig tid. For desisjonsproblemer skal vi i spesielle tilfeller (i forbindelse med ikke-deterministiske turing-maskiner) løse litt på dette.

Ellers er altså 'størrelsen av en instans' et avgjørende begrep for teorien videre, og den er ment å være et mål for *den informasjonsmengde som skal til for å angi en spesiell instans, til forskjell fra alle de andre instanser*. Merk altså at selve problemstillingen, samt hvordan man har blitt enig om å kode dataene som angir en spesiell instans, da skal anses som 'kjent'. Når man skal måle størrelsen av en instans, er det dermed bare de rene 'rådata' som skal telles med.

Informasjonsmengden, og derved problemstørrelsen, regnes som antall tegn (i et passelig alfabet) som går med når vi gjør en 'fornuftig' koding. Det nøyaktige størrelsesbegrepet vil derfor avhenge av akkurat hvilken slik koding som brukes. Men merk her at størrelsen på alfabetet bare vil forandre antall tegn med en konstant faktor. Skal vi f.eks. bruke bit (to tegns alfabet) i stedet for ASCII-tegn (128 tegns alfabet) så vil vi bruke opp til 7 ganger så mange tegn. Dette vil derfor ikke ha noen betydning om vi måler algoritmens effektivitet med 'O-mål', der konstante faktorer jo blir abstrahert bort.

Oftest vil også andre typer forskjeller i kodingsmåte bare forandre størrelsen med (grovt sett) en konstant faktor, men vi kan også oppleve kraftigere forskjeller. Vi kan f.eks. representere en graf som (1): Listen av kanter, eller (2): En nabomatrise lagt ut linje for linje. Her kan størrelsen av den andre vokse opp til kvadratisk i forhold til den første (om det er få kanter). Dette kunne derfor få betydning om vi måler effektiviteten av en algoritme med O-mål (den kan virke mer effektiv om vi bruker det siste som problemstørrelse).

I den videre teori skal vi imidlertid bruke et enda 'grovere' mål enn O-mål, nemlig at vi regner algoritmer som like effektive dersom de 'ikke skiller seg mer enn polynomisk fra hverandre', og da får heller ikke de kvadratiske forskjeller i størrelse som vi så over, noen betydning. I forbindelse med koding av instanser er det spesielt viktige å huske på at man ikke regner størrelsen av et tall som selve tallet, men som antall siffer tallet behøver (og grovt regnet er dette 10-logaritmen til tallet, om det er desimalt kodet). Tenker man tilbake på IN-110 (sukk) så ble der vanligvis ikke størrelsen på tallene regnet med i det hele tatt. Derfor ble størrelsen på et sorteringsproblem der bare regnet som antall tall, mens vi her vil si at problemstørrelsen er summen av antall siffer for alle tallene. Se forøvrig videre filosofier om størrelsesbegrepet på side 9 og side 21.

G&J: Kapittel 1.3

Her innføres altså både O-notasjon og 'polynomisk tid algoritme' (som vi vanligvis skal forkorte til 'polynomisk algoritme') i et par håndgrep. Dette at vi bare er interessert i "worst case"-tid (altså at vi for hver størrelsesklasse bare ser på tidsforbruket for det problem som tar *lengst* tid) kommer litt dårlig fram. Dette stikker hovedsakelig i definisjonen av 'time complexity function' i siste avsnitt av kap. 1.2, der det står *largest amount of time* for hver størrelse.

Det stikker imidlertid også i at O-notasjonen i seg selv er å tolke som en øvre beskrankning. Vil man angi andre typer beskrankninger finnes varianter av O-notasjonen som har litt annen betydning. De som vanligvis brukes er følgende:

$f(n)$ er $O(g(n))$ $g(n)$ vokser minst like fort som $f(n)$
 $f(n)$ er $\Omega(g(n))$ $g(n)$ vokser ikke fortere enn $f(n)$ (egentlig: enn 'toppene' av $f(n)$)
 $f(n)$ er $\Theta(g(n))$ $g(n)$ vokser like fort som $f(n)$

Legg ellers merke til at det å være $O(p(n))$ for et polynom $p(n)$ like godt kan angis som å være $O(n^k)$ for en konstant k .

Man skal også være klar over at det i mange sammenhenger kan være vel så interessant å se på hvordan *gjennomsnittlig* tidsforbruk (for hver størrelsesgruppe) vokser. Dette er imidlertid ofte mer grumsete å beregne, bl.a. fordi vi må ha en sannsynlighetsfordeling å gå ut fra for instansene i hver størrelsesgruppe. Vil f.eks. alle rekkefølger være like sannsynlig input til vår sorteringsalgoritme, eller vil den vanligvis få ‘nesten sorterte’ arrayer?

Ellers følger i kap. 1.3 de vanlige malende beskrivelser av hvor sørgelig fort eksponensielle algoritmer (og det som verre er) kommer til kort, dog med litt sunne motforstillinger. Videre kommer litt mer om størrelser, som nå skulle være greit å lese.

På slutten av kapittelet kommer et viktig poeng, og det gjelder forholdet mellom vanlige “random access” maskiner (som bruker like lang tid på å hente noe fra hvilkensomhelst celle i lageret), og de enkle ‘Turing-maskinene’ (som kommer i diverse varianter) som har et lesehode som bare kan bevege seg ett hakk av gangen. Om vi forsøker å programmere samme algoritme for en RAM-maskin og en Turing-maskin så blir det selvfølgelig stor forskjell på tidsforbruket for disse, også om vi beskriver det med O -notasjon. Tabellen i fig. 1.6 angir f.eks. at det som kan gjøres i tid t på en RAM-maskin kan gjøres i tid $O(t^3)$ på en standard (én-tapes) Turing-maskin.

Dersom vi imidlertid gjør den grovere oppdeling at vi plasserer alle polynomiske algoritmer i samme gruppe, så ser vi at det ikke spiller noen rolle hva slags maskintype vi snakker om. Det som kan gjøres i polynomisk tid på de kraftigere maskinmodeller, kan også gjøres i polynomisk tid på de aller enkleste maskiner (dog med polynomer av høyere grad).

Heri ligger også mye av begrunnelsen for at ‘polynomiske algoritmer’ er et behagelig og robust begrep å arbeide med. Sett fra en annen synsvinkel stikker dette også i at om $p(n)$ og $q(n)$ er polynomer, så er også $p(n) + q(n)$, $p(n) * q(n)$ og $p(q(n))$ alle polynomer.

G&J: Kapittel 1.4

Her skiller boka først ut de problemer som av helt opplagte grunner ikke kan løses i polynomisk tid, nemlig de som kanskje må ‘skrive ut’ en løsning som er av eksponensiell størrelse. Dessuten finnes en del problemer der man beviselig ikke kan lage noen løsningsalgoritme i det hele tatt, og disse kan selvfølgelig heller ikke løses i polynomisk tid.

Men selv om man kutter ut disse tilfellene, har man altså for noen få problemer klart å vise at de umulig kan løses i polynomisk tid. På slutten av kapittelet forsøker boka å karakterisere hvor vanskelige alle disse problemene er (de er “ikke i NP”), men dette er nok vanskelig å få tak i på dette stadium, så man behøver ikke fundere for mye over det.

G&J: Kapittel 1.5

Her diskuteres først begrepet ‘reduksjon’, og essensen av den type reduksjoner vi skal gjøre står i 6. avsnitt: “First, he ...”. Lenger opp på siden (og de fleste andre steder i boka) beskrives en ‘reduksjon av problemet P til problemet Q ’ (siden skrevet $P \propto Q$) som en ‘transformasjon’ eller en ‘funksjon’ av en instans av P til en instans av Q .

Dette kan virke veldig matematisk, og det kan være like greit å tenke seg at en reduksjon av P til Q består i å *angi en prosedyre som løser P , der man får lov å kalle en (tenkt) prosedyre som løser Q* . For at denne reduksjonen skal være

en 'polynomisk reduksjon', må den angitte prosedyren løpe i polynomisk tid, forutsatt at Q løper i polynomisk tid (og da kan man like godt tenke seg at Q løper i konstant tid).

Ut fra denne framstillingen ligger også det også snublende nær å vurdere å kalle Q-prosedyren flere (men bare et polynomisk antall) ganger. Og ut fra vårt hovedanliggende, nemlig å vise at *dersom Q kan løses i polynomisk tid så kan P løses i polynomisk tid*, er jo dette helt OK. I de første kapitlene (kap. 2, 3 og 4) er vi imidlertid i den spesielle situasjon at vi bare ser på desisjonsproblemer (ja/nei-problemer), og at det er stor forskjell på det å gi et *ja*-svar og det å gi et *nei*-svar.

Derved forlanges det også at alle reduksjonene 'bevarer forholdet mellom ja og nei', og under disse betingelser er det greiest å forlange at prosedyren som løser P bare gjør *ett kall* på Q-prosedyren, og at det alltid er slik at et *ja-svar fra Q-prosedyren tilsvarer et ja-svar for P*, og omvendt. Det er for å få dette forholdet tydeligere fram at man sier at en reduksjon er en *transformasjon* av P, nemlig rett og slett den transformasjon vår P-prosedyre må gjøre av den aktuelle P-instansen før den har en Q-instans som den kan be Q-prosedyren å løse. Kravet blir dermed at denne transformasjonen kan gjøres i polynomisk tid.

Når vi senere, i kapittel 5, frigjør oss fra bare ja/nei-problemer, tar også boka fram muligheten til å gjøre flere kall, og denne type reduksjon kalles da "Turing-reduksjon".

Man kan her også tenke over følgende lille poeng: Når man sier at *P kan reduseres til Q*, kan det gi inntrykk av at Q må være et *enklere* problem enn P, mens det faktisk er det omvendte som er tilfelle. Det som ligger i dette er jo at en prosedyre som løser Q også har 'kraft nok' til å løse P, altså at Q er minst like 'vanskelig å løse' som P.

Videre beskrives uformelt problemklassen NP. Det man først og fremst skal merke seg er altså at NP representerer en *øvre* beskrankning på vanskeligheten av et problem (og at alle 'lette' problemer derved uten videre faller i NP). Denne øvre beskrankningen sier for det første at det skal være et ja/nei-spørsmål, og for det andre at selv om ikke problemet nødvendigvis kan *løses* i polynomisk tid, så skal man i hvert fall ved et ja-svar kunne gi et *slående argument* (et 'vitne') for at svaret faktisk er ja, og det å *sjekke at dette argumentet er riktig* skal kunne gjøres i polynomisk tid.

Et slikt slående argument kan f.eks. være å eksplisitt angi en 'Hamiltonsk krets' dersom spørsmålet er om en slik finnes. Dersom spørsmålet er om et tall er 'sammensatt' (altså *ikke* er et primtall), så kan et passelig 'vitne' være å angi to tall som ganget sammen gir tallet. Begge disse spørsmålene er derfor i NP, og det gjelder faktisk svært mange av de probleme vi kommer bort i til daglig (både de lette og de vanskelige), forutsatt at vi (1) stiller dem som ja/nei spørsmål, og (2) lar ja-svaret gjelde den varianten som lar seg 'bevitne'.

Klassen NP kan også løslig beskrives som de desisjons-problemer som kan løses ved rekursiv søking, der *rekursjonsdybden er polynomisk begrenset*. De problemer i NP som kan løses i polynomisk tid (uten noe vitne e.l. i det hele tatt) kalles gjerne P. Merk altså at klassen P dermed bare inneholder desisjonsproblemer.

Det er først når vi innfører de NP-komplette problemer at det snakk om en *nedre* begrensning av vanskeligheten. De NP-komplette problemer er pr. def. de 'vanskeligste' problemene i NP, i den forstand at dersom vi finner en polynomisk algoritme for et slikt problem, så vil den uten videre også være kraftig nok til å løse *alle* problemer i NP i polynomisk tid. Det Cook gjorde var å vise at det faktisk *finnes* slike NP-komplette problemer, og at 'Tilfredstillbarhet' (for visse logiske uttrykk) faktisk er et slikt problem. Det er, som vi skal se, også svært mange andre NP-komplette problemer.

Og altså, til slutt i kapittelet, nevnes det store åpne spørsmål: Lar de NP-komplette problemer seg løse i polynomisk tid? Om de gjør det vil jo *alle* problemer i NP la seg løse i polynomisk tid, og derved vil vi ha $P = NP$. I motsatt fall er altså P *ekte* inneholdt i NP, og dette holder de fleste som mest sannsynlig.

En kommentar til desisjonsproblemer

Til slutt en liten kommentar til dette at vi (hvertfall i første omgang) begrenser oss til bare desisjonsproblemer. Det kan jo umiddelbart virke som en meget grov forenkling av f.eks. 'Handelsreisendes problem', når vi i stedet for å spørre etter korteste reiserute, bare spør om det *finnes* en reiserute som er kortere enn en oppgitt lengde.

Det er opplagt at om vi her kan løse optimaliseringsvarianten i polynomisk tid, så kan vi også løse ja/nei-varianten i polynomisk tid. Men det interessante er at også det omvente i en viss forstand gjelder: *Om vi kan løse ja/nei-varianten i polynomisk tid så kan vi også finne lengden av den korteste ruten i polynomisk tid.*

For å få vist dette skal vi anta at alle avstandene mellom byer er gitt som heltall, og vi skal benytte en slags binærsøking. Detaljene i dette skal vi ta som en oppgave, og boka kommer tilbake til liknende ting i kapittel 5, der man også tar opp spørsmålet om å *finne en kortest mulig tur*, og ikke bare hvor *lang* en slik tur må være. Konklusjonen er altså at det å begrense seg til bare desisjonsproblemer ikke er en så dramatisk forenkling som det i første omgang kan synes.

5.3 Kommentarer til kapittel 2

Kapittel 2 i G&J presenterer altså det helt tekniske grunnlag for stoffet omkring P og NP og NP -kompletthet. Sett fra vår synsvinkel er hovedsaken med dette kapittelet: (1) Gi teorien et formelt grunnlag som man kan gå tilbake til om det er noe i det uformelle bildet man lurer på hva 'egentlig' betyr. (2) Få utført beviset for Cooks teorem, som er avhengig av at man har en forholdsvis enkel maskin.

G&J: Kapittel 2.1

Boka ser her først på den typiske måten de vil beskrive problemer på, og de ser på hvordan max. og min. problemer naturlig kan lages om til desisjonsproblemer. En del av filosofien her (og noe vi får mer av i kap. 5) behandles også i oppgave 21.

Øverst på side 20 defineres sammenhengen mellom ja/nei-problemer og språk over et visst alfabet. Legg her merke til at om vi har et problem og en gitt koding for dette, så får vi naturlig *tre* typer strenger: De som er koder av ja-instanser, de som er koder av nei-instanser, og de som ikke er koder av noen problem-instans i det hele tatt (og den siste gruppen er gjerne den største). Poenget er nå at når man går ned på nivået med språk og Turing-maskiner, så slår man gjerne de to siste gruppene sammen, og snakker bare om ja-strenger og 'de andre'.

Videre er det altså mer snakk om måter å kode instanser på. Dette ble også kommentert noe i forrige notat, og skulle være grei lesing. Merk at man kan gjøre seg mer kodings-uavhengig ved å lage en funksjon 'Length(I)' som direkte fra en instans angir et slags abstrakt lengdemål (uten tanke på noen spesiell koding). For at denne skal være fornuftig må lengden den gir være være 'polynomisk relatert' til den lengden vi får ved enhver rimelig koding. Dessuten må tallverdiene som inngår i instansen påvirke Length-funksjonen med sitt *antall siffer* (som

vil være logaritmisk i forhold til tallstørrelsen, slik som i eksempelet for Traveling Salesman på side 20).

G&J: Kapittel 2.2

Her defineres altså en vanlig (én-tapes og deterministisk) Turing-maskin. Selve programmet i en slik maskin blir ofte tegnet som en rettet tilstandsgraf, mens den i boka er satt opp i en tabell. Dette er en smaksak. Det er en del ting som er viktige å merke seg angående Turing-maskiner:

- Begrepene ‘tid’ og ‘input-lengde’ blir helt veldefinert.
- Det eneste som er *ubegenset* med en slik maskin er ‘tapen’. For en bestemt DTM er både alfabetet og ‘programmet’ gitt og endelig.
- Da hodet til en DTM bare flytter seg ett hakk i hvert skritt, kan en DTM til enhver tid bare ha ‘vært borte i’ en endelig del av tapen, og mer eksplisitt har den etter tiden t bare vært i området fra $-t$ til $t + 1$. Det vil også si at den etter ‘polynomisk tid’ bare har vært borte i område av tapen hvis totale lengde er begrenset av to ganger det samme polynom.
- Turing maskiner kan i prinsippet gjøre alle beregninger som en hver annen ‘rimelig’ maskin kan gjøre, og hva mer er: Dersom det finnes en rimelig maskin som kan gjøre en beregning i polynomisk tid, så kan også en DTM gjøre den samme beregningen i polynomisk tid.

Boka definerer begreper som ‘accept’ og ‘reconize’ (side 24), men disse er egentlig ikke så viktige for oss, da alle DTM’er som interesserer oss skal stoppe etter endelig tid. Vi kan like godt definere de begrepene vi trenger direkte, og det viktige for oss (se nederst side 25) er at en DTM sies å *løse* et problem (med en viss koding) dersom den stopper etter endelig tid for *enhver* input-streng, og at den stopper i ja-tilstand (q_Y), *nøyaktig* for de strenger som er koder av ja-instanser (og for alle andre stopper i q_N).

Man kan selvfølgelig her frigjøre seg fullstendig fra noe problem, og bare snakke om strenger og språk. Igjen ser vi da at “tulle-strenger” (ikke koding av noen instans) og nei-strenger faller i samme gruppe, og at den også må stoppe etter endelig tid for slike tulle-strenger.

Formelen for tidsbruk nederst på side 26 sier altså at for hver størrelsesgruppe n skal vi regne $T_M(n)$ som den *lengste tiden* den bruker for noen input-streng av lengde n , altså ‘worst case’. Dette brukes så til å definere polynomiske DTM’er og derigjennom klassen P (som formelt her består av språk).

G&J: Kapittel 2.3

Som innledning starter de her med en litt motivasjon, som skulle være greit å lese. På norsk vil vi ofte bruke ordet “vitne” som betegnelse på en gjetning som kan brukes til å verifisere (bevitne) et ja-svar.

Selve utvidelsen fra en deterministisk til en ikke-deterministisk TM er altså i denne boka gjort meget enkelt, ved å bare å innføre et gjette-hode som gjør seg ferdig før en vanlig DTM så overtar. I andre modeller av NDTM’er har man lagt inn ikke-determinisme i selve programmet, omtrent som ved en ikke-deterministisk endelig automat. Alle slike varianter ville vært ekvivalente for vårt formål, men den som er valgt er spesielt enkel å behandle teoretisk.

Man kan tenke på en NDTM, og hva det vil si at den godkjenner en input-streng, på flere måter:

- At den gjetter *tilfeldig*, og vil vi si at den godkjenner en input-streng dersom det *finnes* en gjetting som gjør at den stopper i q_Y .
- Eller vi kan tenke oss at gjette-hodet er *meget intelligent* og alltid velger den best mulige 'gjetting' (ut fra å skulle komme så raskt som mulig til q_Y etterpå). Da vil vi si at den godkjenner en input-streng dersom den i det hele tatt stopper i q_Y .
- Eller vi kan tenke på den som en maskin som hver gang den har flere valg deler seg i et tilsvarende antall maskiner som forfølger hvert sitt alternativ. Da vil vi si at den godkjenner en input-streng dersom *en eller annen* av alle disse maskinene stopper i q_Y .

Med NDTM'er kommer det altså inn en kraftig usymmetri mellom ja og nei, ved at en NDTM sies å "løse" et problem dersom den vil godkjenne (ende opp i q_Y for en eller annen gjetning) enhver streng som er koding av en ja-instans, og for øvrig gjøre *hva som helst* for kodinger av nei-instanser og for tull-strenger som ikke svarer til noen instans.

Dette reflekterer seg også i definisjon av tidsforbruket (side 31), som er definert *bare* ut fra strenger som tilsvarende ja-instanser, og for hver ja-instans igjen ut fra den gjettingen som gir *færrest mulig skritt*. Dersom det i en størrelsesklasse i det hele tatt ikke er ja-instanser sies tidsforbruket for denne størrelsen n rett og slett å være 1. Ut fra dette defineres så polynomiske algoritmer for NDTM'er og klassen NP rett fram.

G&J: Kapittel 2.4 (feilaktig angitt som kap 2.1 i boka)

Det viktige her er Teorem 2.1, og det sier på en måte at selv om vi sågar tillot NDTM'er å gå i evig løkke for nei- og tulle-instanser, så kan vi for *polynomiske begrensede* NDTM'er gjenvinne 'endelig kontroll' for alle typer instanser. For en gitt input(-lengde), og ut fra polynomet som begrenser antall steg for en ja-beregning, kan vi nemlig lett beregne lengden av det *lengste vitnet* som maskinen kan komme borti i en mulig ja-beregning. Om vi så kjører maskinen $P(n)$ skritt for alle mulige vitner med denne lengden, og aldri ender i q_Y , ja så er svaret helt sikkert NEI!! Vi kan også lett sette opp en øvre grense for hvor lang tid en slik beregning vil ta.

G&J: Kapittel 2.5

Her kommer altså den formelle definisjonen av begrepet *transformasjon* (som er den viktigste typen reduksjon vi skal arbeide med). Legg merke til at vi her bruker en DTM som en transformasjons-maskin, og at vi da egentlig bare behøver én stopp-tilstand, som rett og slett sier at transformasjonen er ferdig, og ligger på tapen. Igjen er det viktig at om dette er en polynomisk DTM, så kan lengden av output-strengen bare være av polynomisk lengde i forhold til lengden av input-strengen. Dette er viktig i beviset både av Lemma 2.1 og Lemma 2.2.

Rent formelt går altså en transformasjon fra et språk til et annet (slik at det å 'være med i språket' er det som 'bevares'), men i vår senere praksis går den fra kodinger av instanser for ett problem til kodinger av instanser for et annet, slik at ja-instanser avbildes på ja-instanser og nei-instanser på nei-instanser.

Kapittel 2.5 slutter så med definisjon av 'polynomisk ekvivalente' problemer, samt den viktige definisjonen av hva det vil si at problem er 'NP-komplett'.

G&J: Kapittel 2.6

Selve Cook's teorem er sentralt. Dette sier altså at dersom P_i er et problem i NP, så gjelder alltid $NP \propto SAT$. Angående beviset for dette teoremet, så var de viktigste punktene i dette altså at:

- For en gitt instans av et tilfeldig problem Π i NP vil vi sette opp et logisk uttrykk (instans av SAT) som er tilfredstillbart hvis og bare hvis det finnes et vitne slik at NDTM'en som løser problemet (slik finnes pr. def.) ville ende opp i q_Y .
- Ut fra størrelsen av Π -instansen og polynomet som angir hvor lang tid NDTM'en kan bruke på å gjette og sjekke vitner, kan vi beregne hvor mange skritt en mulig ja-beregning kan ta, og derved hvor stor del av tapen som kan berørers.
- Variablene i det logiske uttrykket beskriver tilsammen en fullstendig beregning på NDTM'en, innenfor disse beskrankninger i tid og rum, og det er bare et polynomisk antall av dem.
- Selve det logiske uttrykket i SAT-instansen legger på begrensninger slik at de tilstandene som variablene vil foreskrive for NDTM-utførelsen (a) på alle måter er konsistenete, (b) er i overenstemmelse med programmet i den aktuelle NDTM'en, (c) starter med den gitte Π -instans som input, og (d) slutter i tilstand q_y . Det logiske uttrykket blir langt, men det er dog polynomisk begrenset i forhold til lengden av den Π -instansen vi startet med.
- Det logiske uttrykket sier ikke noe om hva som initielt står i gjette-området, og SAT-spørsmålet: "Finnes logiske verdier for variablene slik at uttrykket er tilfredstilt", blir da essensielt et spørsmål om "Finnes mulig initialisering av gjette-området slik at NDTM'en ender i q_Y i løpet av det antall skritt vi ser på". Og dette er jo nettopp det samme som å spørre om Π -instansen er en ja-instans.

5.4 Kommentarer til kapittel 3

Innledningen: Hvordan vise at et problem er NP-komplett

Forrige kapittel sluttet altså med å vise at det fantes minst ett NP-komplett problem, nemlig SATESFIABILITY (senere kalt SAT). Dette problemet er altså slik at for alle problemer Π i NP finnes en polynomisk ja/nei-bevarende transformasjon fra Π til SAT. Dette vil igjen si at om vi klarte å løse SAT i polynomisk tid, så ville *alle* problemer i NP kunne løses i polynomisk tid (og vi ville ha $P=NP$), men dette holder altså de fleste for usannsynlig.

Vi snakker derfor i det videre som om $P \neq NP$, og da er de NP-komplette problemer i en passelig forstand de *vanskeligste* problemer i NP. At vi i klassen av NP-komplette problemer først traff på SAT var imidlertid en 'tilfeldighet', klassen inneholder også en rekke andre problemer og det er en grunnstamme av slike problemer man bygger opp i kapittel 3.1.

Når vi nå først har ett problem vi vet er NP-komplett, er det svært mye greiere å vise at vi har flere, og den vanlige metoden for å vise dette står summert opp i fire punkter på side 45 i G&J. Vi skal kommentere hvert av punktene (1) – (4):

Angående (1):

For å vise at et problem Π er i NP kreves to ting:

(i) For hver ja-instans av Π må man vise at det finnes et 'vitne' som bekrefter at dette er en ja-instans. Det finnes som regel mange valg m.h.t. hva slags form

eller format et slikt vitne skal ha, og man har frihet til å velge dette så behagelig som mulig med tanke på å kunne vise punkt (ii) under så lett som mulig. (For eksempel kan et Hamiltonian Circuit-vitne være en mengde kanter i tilfeldig rekkefølge, en sekvens av kanter i en eksplisitt syklisk rekkefølge eller en sekvens av 'byer'. Ofte er vel den siste greiest, mens den første formen gjør det unødvendig tuklete å sjekke at vitnet er korrekt.)

(ii) Videre må man komme opp med en prosedyre som, for en gitt Π -instans og et påstått vitne for at dette er en ja-instans, kan sjekke i polynomisk tid (i forhold til størrelsen av Π -instansen!) om vitnet holder hva det lover. Merk her at man bare behøver å vise at prosedyren løper i polynomisk tid for det tilfellet at vitnet *blir* godkjent. Om det er et galt vitne (som f.eks. kan være alt for langt) kan prosedyren godt bruke aldri så lang tid, eller kanskje til og med ende i evig løkke.

Angående (2):

Her skal man altså velge et problem som man allerede *vet* er NP-komplett, og som man skal transformere til problemet Π . Her er i prinsippet alle kjente NP-komplette problemer like gode, men for å få til en rimelig enkel transformasjon er gjerne noen bedre enn andre. Når man som forsker står med et nytt problem Π er det her ingen regler som kan hjelpe en i dette valget ut over erfaring og god nese. Når man sitter på eksamen er det imidlertid håp om at det ligger et hint et sted.

Angående (3):

Her skal man finne en transformasjon fra instanser av problemet valgt under (2), til instanser av Π . Hvordan dette skal gjøres er det også lite å si om, men dersom problemet man skal transformeres *fra* er gitt har man hvertfall noe å arbeide ut fra. Se forøvrig kommentarer lenger ned angående de tre 'teknikkene' i G&J kap. 3.2.

I tillegg til å vise at denne transformasjonen kan utføres i polynomisk tid (punkt (4) under), må man vise at denne transformasjonen er ja/nei bevarende. I G&J ligger dette kravet i definisjonen av en 'transformasjon' (nemlig i pkt. 2, side 34: x er ja-instans hvis og bare hvis $f(x)$ er ja-instans).

Om vi transformerer fra Π' , så gjøres dette vanligvis slik:

(i) Se på en tilfeldig ja-instans av Π' , og vis at den transformerer seg til en ja-instans av Π .

(ii) Se på en tilfeldig instans (altså ikke nødvendigvis ja-instans) av Π' , og *anta* at den avbilder seg på en ja-instans av Π . Vis ut fra dette at Π' -instansen vi startet fra måtte være en ja-instans.

Punkt (ii) kan også gjøres ved å vise at alle nei-instanser blir transformert til nei-instanser. Det er lett å vise at de to variantene av (ii) er logisk ekvivalente, da alle instanser er enten ja- eller nei-instanser. I første variant viser vi nemlig '(transform(A) er ja-inst) \Rightarrow (A er ja-inst)', som ved negering og snuing gir: '(A er ikke ja-inst) \Rightarrow (transform(A) er ikke ja-inst)'. Når det gjelder teknikken 'restriksjon' kan dette ta seg litt annerledes ut.

Angående (4):

Man må her vise at transformasjonen valgt under (3) kan utføres i polynomisk tid.

Hva er en 'reduksjon'

Ordet *reduksjon* dukker opp jevnlig i dette stoffet, og det er kommet et par spørsmål om hva det egentlig betyr. Når vi sier at "problem A kan reduseres til problem B" så mener vi generelt at "jeg kan klare å løse A bare jeg kan klare å løse B". I vår setting har vi snevret dette noe inn, ved at vi lar det bety "jeg

kan klare å løse A *i polynomisk tid*, bare jeg kan klare å løse B *i polynomisk tid*". Dette er diskutert på side 13 i boka.

Nå er det imidlertid også flere varianter av denne type polynomiske reduksjoner, nemlig 'transformasjon' (som vi har sett på til nå, og som bare brukes på desisjons-problemer), og Turing-reduksjon (som fungerer mer generelt, og som dukker opp i kap. 5). På grunn av dette forsvinner plutselig begrepet 'reduksjon' etter kap. 1, og det blir i stedet bare snakk om (den spesielle reduksjonsformen) 'transformasjon'. Spesielt forvirrende kan dette bli når en av teknikkene for transformasjon heter 'restriksjon', og da med en 'retning' på problemstillingen som er helt omvendt av den i 'reduksjoner'.

Hvordan se på 'teknikkene' som er diskutert i kap 3.2

I kapittel 3.2 ser G&J på tre 'teknikker' for å lage transformasjoner, nemlig *restriksjon*, *lokal forandring* og *komponentvis design*. Det at dette kalles *teknikker* må imidlertid ikke forstås dithen at de på noen måte er konstruktive. Selv om vi f.eks. får høre at beviset for $VC \propto HC$ går ved 'komponentvis design' så ligger det ikke deri noen som helst anvisning på hvordan dette beviset skal lages. Det det stort sett sier er at dette beviset er ikke helt enkelt.

Det samme gjelder også de to andre teknikkene, selv om bevisene her stort sett er enklere, og det derfor lett kan virke som om de faller mer ut av seg selv. Som G&J selv understreker på side 63 så er denne grupperingen en typisk etterpå-gruppering av bevisene, og den skal stort sett ikke betraktes som mer enn en gruppering i 'lett', 'vanskeligere' og 'vanskeligst'.

Når det gjelder teknikken 'restriksjon' så er det her lett å tulle med retningen man snakker i, siden den her blir snudd i forhold til det vanlige (nemlig den vei transformasjonen går).

Vi ser på transformasjonene i kap. 3.1 som eksempler, og vi klassifiserte $SAT \propto 3SAT$ som en 'lokal forandring', mens både $3SAT \propto VC$ og $VC \propto HC$ ble klassifisert som 'komponentvis design'. Klassifikasjonen av $3DM \propto PARTITION$ (Som vi tok som $3DM \propto SUBSET\ SUM$, mens $SUBSET\ SUM \propto PARTITION$ gikk som en øvelse) er vel litt usikker, men den er vel helst en 'lokal forandring'. Hovedideene ved denne grupperingen skulle forøvrig framgå av innledningene i kap. 3.2.1, 3.2.2 og 3.2.3.

G&J: Kapittel 3.1

De seks problemene det her innledes med vil vi i dette kurset gå gjennom på kryss og tvers på forskjellige måter. Transformasjonene er formulert i boka med en god del indeksering, mengder, funksjoner og knappe skrivemåter. Noen liker dette og andre ikke, men for å skjønne hva som skjer bør man hele tiden støtte seg på eksempler og skisser for å få det hele ned på jorda. Vi skal ikke her komme med noen ny utlegning av de enkelte bevisene, men bare knytte noen korte kommentarer til hver av dem.

- $SAT \propto 3SAT$:

Som eksempel på denne kan man se på oppgave 25. Man kan observere at tilfellene 1 og 2 nederst på side 48 er behandlet så utførlig for å få fram en instans av $3SAT$ der alle variablene som forekommer i en literal alltid er forskjellige. En liten kommentar om dette står i oppgave 26 (kommentar 2).

- $3SAT \propto VERTEX\ COVER$:

At problemet Vertex Cover svært lett kan transformeres til/fra Clique eller Independent Set (Lemma 3.1, side 54) tok vi som en øvelse på gruppe.

Dermed kan jo Vertex Cover i transformasjonen over uten videre erstattes med en av de andre. Her er jo et godt eksempel i boka, og flere eksempler ble diskutert i oppgave 26.

- VERTEX COVER \propto HAMILTONIAN CIRCUIT:

Her kalte vi altså den grafstrukturen som er angitt i figur 3.4 for en “somerfugl”, og når vi skulle gjøre transformasjonen tenkte vi oss først at vi plasserte en slik sommerfugl på tvers av hver opprinnelig VC-kant. Sommerfuglene ble så koblet sammen i tilfeldig rekkefølge ‘rundt’ hver node (de som lå på kanter som har denne noden som endepunkt). Til slutt ble det hele koblet til de K nodene.

- 3-DIMENTIONAL MATCHING \propto PARTITION:

Denne transformasjonen er det greiest å se på som delt i to, nemlig først som 3DM \propto SUBSET SUM, og deretter som SUBSET SUM \propto PARTITION. Den siste av disse blir tatt opp i oppgave 27. Bokas forklaring fram til ‘The final step...’ på side 62, er rett og slett en forklaring av transformasjonen 3DM \propto SUBSET SUM. Som praktisk eksempel på denne, se oppgave 31.

Når man jobber med eksempler på denne transformasjonen kan det være like greit å bruke ti-tall-systemet i stedet for to-tall-systemet. Den eneste forskjellen er da at man må bruke 10-logaritme i stedet for 2-logaritme når man beregner feltbredden p (i figur 3.7, side 61).

Hvordan påvise at noe kan gjøres i ‘polynomisk tid’

I forbindelse med dette kurset er det en sentral ting å kunne påvise at noe kan gjøres i polynomisk tid. I det vi har vært gjennom til nå er dette interessant spesielt i to situasjoner:

(a) Når man vil påvise at et problem er i NP, ved å framvise vitner og vise at de kan sjekkes i polynomisk tid.

(b) Når vi vil vise $A \propto B$, ved å framvise en polynomisk transformasjon fra A til B .

Begge disse punktene er diskutert nærmere i starten av dette notatet.

Det å vise at en skissert algoritme vil bli utført i polynomsik tid betyr altså at vi skal kunne komme opp med ett bestemt polynom $p(n)$ (felles for alle utføring) som er slik at om inputlengden til algoritmen er n , så vil ikke algoritmen bruke mer enn $p(n)$ elementæreskritt.] gjøre dette helt i detalj er svært fiklete, og akkurat hvordan polynomet blir vil være ytterst avhengig av hva slags maskin man har (f.eks. Turing-maskin eller RAM-maskin), og hvordan dataene er representert i detalj.

Boka bygger generelt på en slags ‘følelse’ av at det man kan gjøre ved et passende antall systematiske gjennomløp av dataene uten rekursjon, det er polynomisk. Dette er greit nok om man har erfaring. Man bør imidlertid (også til eksamen!) kunne føre noe mer konkrete argumenter for at en skissert algoritme bare vil ta polynomisk tid (og ut fra slike argumenter kanskje etter hvert skaffe seg en rimelig sikker følelse for dette).

Telling av elementær-operasjoner

En vanlig måte å vise at en algoritme er polynomisk, er å velge visse grunnoperasjoner som man med rimelighet kan påstå går i tid $O(1)$, eller hvertfall helt sikkert i polynomisk tid. Man betrakter så programmet som satt sammen av slike operasjoner, og ut fra visse mål i utgangsdataene (antall noder e.l.) kan man

beregne en øvre grense for antall slike operasjoner som blir utført, og man kan så påvise at det totale antallet er polynomisk.

Dette er en helt legal måte, men ulempen er at det ofte blir for mye å ta med alle operasjoner og at man derfor må operere med noen argumenter for hva som er de 'dominerende operasjoner'. Disse er det ikke alltid så lett å få helt ned på bakken.

En mer syntaktisk tilnærming

Vi skal her se på en annen måte som ofte vil fungere bra, og som har den fordel at man ikke behøver å arbeide med noe eksplisitt polynom. I stedet må man formulere en programskisse etter visse syntaktiske regler, og dernest kunne påvise at de *enkelte* operasjoner som er brukt i programmet 'opplagt' kan utføres i polynomisk tid.

Opplagt polynomiske operasjoner

Grunnenhetene i en slik programskisse må være basis-aksjoner som 'opplagt' kan gjøres i polynomisk tid (ofte i 'konstant tid', eller ved et enkelt gjennomløp av input-dataene). Typisk 'opplagt polynomiske operasjoner' kan være:

- Lese av (lete fram?) verdier oppgitt i den opprinnelige instansen.
- Telle opp størrelser (så som antall noder, kanter eller elementer) i den opprinnelige instansen.
- Sjekke om det i en gitt graf finnes en kant mellom to oppgitte noder.
- Gjøre kjente polynomiske operasjoner, så som f.eks. å sortere en mengde på et gitt (direkte tilgjengelig) kriterium.

Dessuten må vi ved transformasjoner kunne bygge opp en ny instans. I den forbindelse kan det å generere og sette på plass avgrensede deler av den nye instansen, så som f.eks. følgende operasjoner, betraktes som 'opplagt polynomiske':

- lage en ny node,
- lage en ny kant mellom to gitte noder,
- lage en ny klausul med tre bestemte literaler,
- skifte ut en gitt node med en eller annen fast graf-struktur,
- ta en kopi av hele eller deler av den opprinnelige instansen over i den vi vil produsere.

Det er her en forutsetning at de dataene som skal til for å bygge opp den angitte del av den nye instansen må være direkte for hånden ut fra det stedet vi er på i programmet.

Ofte vil transformasjoner foregå ved at vi bare forandrer litt i den opprinnelige instansen, og det blir da et spørsmål om man direkte kan forandre på den opprinnelige innstansen, eller om man først må ta en kopi. Svaret er at begge deler er helt OK, men det er jo i alle tilfelle en opplagt polynomisk operasjon å ta en kopi først.

For- og If-setninger

Videre skal 'programmet' være bygget opp av for-setninger og if-setninger. I hver for-setningene må det antallet elementer for-løkka løper over være opplagt polynomisk, og det må være en opplagt polynomisk prosess og finne fram disse elementene ett for ett. F.eks:

```
for x:= <hver av nodene i grafen G1> do ...
for <alle kantene i grafen G1> do ...
for y:= <hver av kantene som går inn til noden y> do ...
```

```

for <alle klausulene i SAT-instansen> do ...
for z:= <hver av variablene i SAT-instansen> do ...
for u:= <hver av elementene i A i PARTITION-instansen> do ...
for i:= 1 to n do ...      (der 'n' er opplagt polynomisk i
                           forhold til input-lengden)

```

Når det gjelder if-setninger, så må betingelsen i disse kunne avgjøres i opplagt polynomisk tid, etter nøyaktig samme (uklare) prinsipper som for andre 'opplagt polynomiske grunnoperasjoner'. Dette kan f.eks. være:

```

if <antall kanter i G er mindre enn K> then ...
if <antall literaler i klausulen er større enn 3> then ...
if <det går en kant mellom node i og node j> then ...

```

Endelig forlanger vi at den algoritmen vi påstår er polynomisk skal kunne skisseres som et (fast) program satt sammen av slike elementer.

Hvorfor slike programmer er polynomiske

At et slikt program faktisk vil bli utført i polynomisk tid er forholdsvis lett å se. Enhver angitt basisaksjon (eller betingelse i if-setning) vil tekstlig ligge inne i et bestemt antall, si f.eks. d , nestede for-setninger, som hver bare vil bli utført et polynomisk antall ganger. Om den også ligger inne i en eller flere if-setninger så kan dette bare bevirke at operasjonen blir utført færre ganger enn løkketellingen skulle tilsi.

La oss nå si at den polynomiske begrensningen for antall ganger de enkelte omkringliggende for-setninger vil bli utført er $p_1(n), p_2(n), \dots, p_d(n)$, og at den polynomiske begrensningen for basisoperasjonen selv er $q(n)$. Da vil den totale tiden som går med i (alle utførelser av) denne operasjonen være begrenset av $p_1(n) \cdot p_2(n) \cdot \dots \cdot p_d(n) \cdot q(n)$, som igjen er et polynom.

Rent tekstlig vil jo et program bare bestå av et endelig antall slike basisoperasjoner, og dermed vil summen av polynomene for alle disse operasjonene, som igjen blir et polynom, være en øvre grense på den totale tiden for hele programmet. Altså kan algoritmen utføres i polynomisk tid.

Merk altså at vi ikke tillater noe prosedyre-begrep, men dette er bare for å forhindre rekursive kall. Man kan godt implisitt bruke ikkerekursive prosedyrer, ved at man i 'hovedprogrammet' bruker en større basis-operasjon, som man så gir et eget argument for å påvise at den faktisk kan gjøres i polynomisk tid. Argumentet for dette kan skrives som et eget lite program for denne operasjonen, som bringer det hele ned på enda mer basale (og opplagt polynomiske) operasjoner.

Programmet skal være polynomisk, ikke effektivt

Merk at når man lager en slik programskisse er ikke vitsen å lage et effektivt program, men å gjøre det helt klart at det er mulig å utføre algoritmen i polynomisk tid. Graden av polynomet kan man gjerne gjøre unødvendig høy dersom dette kan forenkle argumentasjonen. Det burde sjelden bli nødvendig å skrive slike program-skisser på mer enn 15 - 20 linjer.

Akkurat hva slags syntaks man bruker på if- og for-setningene er ikke så viktig, bare nestingen går tydelig fram. I eksemplene under er det brukt tradisjonell Simula-syntaks. Deklarasjon av variable etc. kan man stort sett heve seg over, dersom deres type etc. går fram av andre ting.

Eksempler:

For å vise at transformasjonen $SAT \propto 3SAT$ (G&J, side 48) er polynomisk, kunne man f.eks. skissere den ut slik:

```
for<alle klausuler i SAT-instansen> do
begin
  j:= <antall literaler i klausulen>;
  if j=1 then <lag 4 nye 3-klausuler etter oppskriften> else
  if j=2 then <lag 2 nye 3-klausuler etter oppskriften> else
  if j=3 then <kopier klausulen direkte> else
  begin
    <lag spesiell første klausul for de to første literalene>;
    for <tredje til tredje-siste literal> do
      <lag standard midt-klausul>;
    <lag spesiell siste klausul for de to siste literalene>;
  end;
end;
```

Legg merke til at vi her f.eks. forutsetter at det å framskaffe en 'fersk' (ubrukt) variabel kan gjøres i polynomisk tid. Dette virker ikke urimelig.

Vi forsøker altså her ikke å beskrive hele algoritmen fullstendig, den vil vanligvis være beskrevet med andre midler på forhånd. Det viktige med å skissere algoritmen på denne formen er å få konstatert at når vi setter alle operasjonene sammen, så utgjør de en polynomisk algoritme.

Denne formen *kan* jo imidlertid også benyttes til å beskrive selve algoritmen, ved at man *først* skisserer den som en algoritme ved hjelp av skjemaene over, og siden beskriver nærmere (gjerne ved andre teknikker) de enkelte operasjonene som inngår. Om de enkelte operasjonene også er 'opplagt polynomiske', så får man jo slått to fluer i ett smekk.

For å vise at transformasjonen $3SAT \propto VERTEX COVER$ (G&J, side 55) er polynomisk, kan transformasjonen skisseres slik:

```
for <alle variable i 3-SAT-instansen> do
  <lag to 'literal-noder' for de to mulige literalene over
  denne variabelen, med en kant mellom>;

for <alle klausulene i 3-SAT-instansen> do
begin
  <lag en node for hver av de tre literalene, hver med en
  kant til den tilsvarende 'literal-noden'>;
  <lag de tre kantene som forbinder de tre nodene>;
end;
```

Om størrelsen på det som produseres

For at en algoritme skal kunne utføres i polynomisk tid, er det *forutsetning* at output-dataene fra algoritmen ikke vokser mer enn polynomisk. Dette fordi man i én grunnoperasjon bare kan produsere en avgrenset datamengde (tenk på Turingmaskiner). Dette gir dermed en måte å vise at en algoritme *ikke* er polynomisk: Om det ikke finnes noe polynom som begrenser den produserte datamengde, så kan heller ikke algoritmen være polynomisk i tid.

Det omvendt gjelder imidlertid ikke uten videre. Man kan bruke aldri så mye tid på bare å produsere ett bit output (f.eks. ja eller nei). Dette med lengden

av output *kan* imidlertid også bygges ut til å bli et gyldig argument for at algoritmen går i polynomisk tid. Man må da grovt sett påvise at tid per output-enhet (f.eks. node, kant osv) er polynomisk, og sammen med at outputlengden er polynomisk vil dette gi den ønskede konklusjon. Boka synes ofte å operere med underliggende argumenter av denne typen, men de kan være litt vonde å få helt klart ned på papiret.

5.5 Kommentarer til kapittel 4

Dette kapitlet starter med en advarsel om hvor farlig det er å innbille seg at dersom ett problem er i NPC, så vil også “liknende” problemer være i NPC. Figuren på side 79 skulle vise dette med all tydelighet. Det kan også ligge andre tilsvarende “fristelser” i farvannet, f.eks. det å tro at skillet mellom det som er i P og det som er i NPC alltid går “mellom 2 og 3”. I tillegg til de to eksemplene som er nevnt på dette øverst på side 78, kan også 2-fargbarhet (av grafer) i P og 3-fargbarhet i NPC være med å underbygge en slik teori. Men dette må man altså langt fra ta som noen generell regel.

Men *hovedsaken* i dette kapitlet er å bevisstgjøre seg at et problem alltid har en indre struktur, og at det vanligvis er en rekke forskjellige “dimensjoner” i denne strukturen som problemstørrelsen kan variere langs. I en graf har vi f.eks. *antall noder* og *antall kanter*, og vi kan også f.eks. snakke om grafens *grad* (det maksimale antall kanter mot en node) og om den er *plan* eller ikke. For SAT har vi bl.a. dimensjonene *antall variable*, *antall klausuler* og *størrelsen* av den største klausulen. For PARTITION har vi *antall tall* og *størrelsen av tallene*.

Og videre: Selv om vi har vist at et problem er NP-komplett når vi tillater instansene å vokse fritt langs alle dimensjoner, så kan det fremdeles være mye å si om hvordan problemet forholder seg til vekst langs de enkelte dimensjonene, når de andre dimensjonene ‘holdes fast’. Dette kommer spesielt tydelig fram når det gjelder tallproblemer, for her vil lineær vekst i det tradisjonelle lengdemålet (antall siffer) tilsvare eksponentiell vekst i tallstørrelsen.

I dette kapitlet gjøres grovt sett to typer analyser: (1) Blir problemet polynomisk løsbart dersom man begrenser en eller flere av dimensjonene på bestemte måter? Og: (2) hvordan kan en øvre tidsgrense for en løsningsalgoritme se ut, uttrykt som en formel der størrelsen langs de forskjellige dimensjonene inngår. Dette siste blir spesielt studert i kapittel 4.3, mens kapitlet om tallproblemer (4.2) kan sies å ha en flik av både (1) og (2).

Man skal her legge merke til at de transformasjonene vi har sett på i forbindelse med NPC-bevis vanligvis ikke “bevarer” disse dimensjonene (som jo også kan ha helt forskjellige natur i de to problemene). Det eneste som blir bevart (relativt til polynomiske forandringer) er den totale instanslengden. Dermed vil en studie av hvordan ett problem forholder seg når dets forskjellige dimensjoner forandres eller begrenses, si lite om hvordan dets “nærliggende” problemer (i forhold til transformasjoner) vil oppføre seg. Hvert nytt problem må i stor grad undersøkes ved en helt egen studie. (Det samme fenomenet vil vi forøvrig finne igjen når det gjelder tilnærmingssalgoritmer i kap. 6, se side 134).

G&J: Kapittel 4.1

I kapittel 4.1 er PRECEDENCE CONSTRAINED SCHEDULING bare å se på som et eksempel, og det er valgt her nettopp fordi det har flere interessante dimensjoner man kan studere det langs, nemlig *antall prosessorer* og *kompleksiteten av rekkefølgekravene*. Ellers skulle dette være forståelig ut fra kommentarene over.

G&J: Kapittel 4.2

Bakgrunnen for dette kapitlet er at det tradisjonelle lengdemålet for tall (nemlig antall siffer) ofte stemmer dårlig med våre behov og vår intuisjon når det gjelder de deler av av instans-beskrivelsen som er tall. Vi gjorde følgende betraktning:

Dersom vi regner at ting blir “store” (i den forstand at de forårsaker ubehagelig lange løsningsstider) for NP-komplette problemer ved størrelser omkring 10 så vil det indikere at tall i instansen ikke vil forårsake tidsproblemer før de får 10 siffer. Om vi tenker i 2-tall-systemet, vil dette si for tallverdier på omkring 1000. Ofte vil imidlertid de tallene som inngår i våre instanser være vesentlig mindre enn dette, f.eks. bare opp til 100 (om vi f.eks. regner med 1% nøyaktighet). Altså: det som er “stort nok” sett fra brukeren, kan fremdeles være så lite (sett i forhold til vårt tradisjonelle lengdemål) at det ikke behøver å lage alvorlige tidsproblemer for en løsningsalgoritme. Dette stikker selvfølgelig i at tallstørrelsen vokser eksponensielt i forhold til antall siffer.

Dermed er det altså håp om at vi kan få til en praktisk brukbar algoritme, selv om problemet i tradisjonell forstand er NP-komplett. Denne problemstillingen kan formaliseres på flere måter. Den mest brukte er å tenke seg at man koder tallene i instansen i 1-tall-systemet (altså ved det antall 1-siffer som tilsvarer tallverdien) i stedet for f.eks. i 2-tall-systemet. Det tilsvarer at man i instanslengden regner tallene med sin *verdi* i stedet for med sitt antall siffer, og man kan så spørre om løsningsalgoritmer fremdeles er eksponensielle i forhold til dette nye lengdemålet.

Dersom det finnes algoritmer som er *polynomiske* i forhold til dette nye lengdemålet, så sies disse å være *pseudopolynomiske*, og slike algoritmer finnes altså for noen problemer, men ikke for alle.

Merk at denne type betraktninger bare er av interesse dersom tallstørrelsene i instansen kan “vokse fritt” (og dermed eksponensielt) i forhold til det tradisjonelle lengdemålet. Problemer der dette er tilfelle kalles altså (*ekte*) *tallproblemer*. Det finnes mange problemer der instansene inneholder tall, men der tallene *ikke* kan vokse på denne måten. Dette gjelder i særdeleshet de tall som brukes til navngiving av elementer i instansen (se G&J, side 94), men også tall som f.eks. K-en i Vertex Cover.

Boka snakker ikke om koding av tall i 1-tall-systemet, men innfører i stedet et ekstra lengdemål *Max*, som skal representere tallstørrelsene i instansen. Denne kan for et gitt problem defineres på forskjellige måter (f.eks. som *det maksimale* tall, *summen* av tallene eller *gjennomsnittet* av tallene), bare med det krav at de forskjellige variantene er polynomisk relatert til hverandre slik som angitt på side 92 og 93.

Merk at for å få større frihet i valg av *Max*-funksjonen bruker boka denne aldri alene i krav om polynomitet, men alltid sammen med det tradisjonelle lengdemålet. Det er derfor vi kan bruke så forskjellige ting for *Max* som *maksimalverdi* og *sum* av tallen. (Når vi bruker metoden med å kode instansens tall i 1-tall-systemet vil imidlertid det nye instans-lengdemålet alltid inneholde både en *Length*-komponent og en *Max*-komponent, slik at vi kan klare oss med dette ene målet når vi skal definere pseudopolynomitet.)

På side 95 motiverer og definerer boka begrepet “NP-kompletthet i sterk forstand”. Dette begrepet har samme forhold til pseudopolynomiske algoritmer som vanlig NP-kompletthet har til polynomiske algoritmer. Definisjonen av dette har som bakgrunn at om man i et tallproblem “kunstig” begrenser tallstørrelsene polynomisk i forhold til instans-størrelsen (med et vilkårlig gitt polynom), så vil en pseudopolynomisk algoritme også være (ekte) polynomisk.

Ut fra dette blir følgende definisjon rimelig: “Et problem i NPC er NP-kom-

plett *i sterk forstand* dersom det finnes et polynom $p(u)$ slik at problemet *forblir* NP-komplett *selv om* vi begrenser tallene i instansen slik at deres verdier ikke overgår $p(L)$, der L er den tradisjonelle instanslengden”.

Det er da hvertfall klart at et problem ikke både kan ha en pseudopolynomisk løsningsalgoritme og samtidig være NP-komplett i sterk forstand (medmindre $P=NP$).

I kapittel 4.2.2 går boka dypere inn på NP-kompletthet i sterk forstand. Vi så imidlertid på noen mer opplagte tilfeller av sterkt NP-komplette problemer, der situasjonen var så enkel at problemene forble NP-komplett selv om vi la *absolutte begrensninger* på størrelsen av de tallene som inngikk. Et typisk eksempel her er Traveling Salesman som forblir NP-komplett selv om vi begrenser avstandene til bare å kunne være 1 eller 2 (se starten av kap. 4.2.2).

G&J: Kapittel 4.3

Merk at det som i dette kapittelet kalles “naturlige parametere” på mange måter tilsvarer det vi tidligere snakket om som forskjellige “dimensjoner” i problemet.

Legg også merke til at noen av de naturlige parameterne kan ha et *polynomisk* forhold til det tradisjonelle lengdemålet (f.eks. antall noder i en graf, antall elementer i en mengde eller antall siffer i et tall), mens andre kan ha et eksponentielt forhold til lengden (men til gjengjeld da gjerne et polynomisk forhold til *Max*) som f.eks. størrelsen av et tall. Det er spillet mellom disse forhold som utgjør spenningen i kapittel 4.3.

5.6 Kommentarer til kapittel 5

Hovedideen med dette kapittelet er å utvide de betraktninger vi til nå har gjort for desisjonsproblemer, til også å kunne utsi noe om andre typer problemer. Vi må da f.eks. også for disse kunne snakke om at et problem er *minst like vanskelig som* eller *ikke vanskeligere enn* et annet problem.

Vanskeligheten med dette er at vårt gode gamle begrep ‘polynomisk transformasjon’ (skrevet $A \times B$) ikke fungerer for annet enn desisjonsproblemer. Derfor innføres her en mer ‘liberal’ reduksjonsform (i den forstand at den innbefatter $A \times B$, og mye mer) som kalles Turing-reduksjon, og som skrives $A \times_T B$.

Vår definisjon av $A \times_T B$ er rett og slett at vi må kunne skrive en prosedyre som løser A i polynomisk tid dersom vi får lov å kalle (gjærne flere ganger) en (orakel-)prosedyre som løser B i polynomisk tid. “Parameterene” til A -prosedyren er da instanser av A , og parameterene til B -prosedyren er instanser av B . Det følger da helt opplagt at om $A \times_T B$ og B kan løses i polynomisk tid, så kan også A løses i polynomisk tid, og dermed også omvendt: Dersom A *ikke* kan løses i polynomisk tid, så kan heller ikke B løses i polynomisk tid.

Øverst på side 110 defineres noe som kalles *søkeproblemer*. Boka konstaterer lenger ned at også desisjonsproblemer kan sees som søkeproblemer, og tenker man litt over saken innser man fort at nær sagt *ethvert* problem (av den typen vi kan tenke på å få en datamaskin til å løse) kan sees som et slikt søkeproblem. Begrepet ‘søkeproblem’ representerer altså ikke en *avgrensning* i forhold til ‘mengden av alle problemer’, men snarere en *terminologi* og en *form* som vi kan bruke når vi vil diskutere generelle problemer.

Når vi vil sammenlikne vanskeligheten av søkeproblemer kan vi nå bruke Turing-transformasjon, og vi tolker da $A \times_T B$ til å bety at “ B er minst like vanskelig som A ”. Ut fra dette definerer vi “problem A er NP-hardt” til å bety at A er minst like vanskelig som de NP-komplette problemer, altså at $B \times_T A$ for et eller annet NP-komplett problem B .

Tilsvarende definerer vi “A er NP-lett” til å bety at $A \propto_T B$ for et problem B i NP (altså ikke nødvendigvis B i NPC, selv om det vil følge av definisjon at også for et problem C i NPC må $A \propto_T C$ gjelde). De problemer som både er NP-harde og NP-lette kalte vi NP-ekvivalente, og disse kan vi altså tenke på som en utvidelse av de NP-komplette problemer til alle søkeproblemer. Vi fortsetter imidlertid å la problemklassene P, NP og NPC bare inneholde desisjonsproblemer. Når vi ser på søkeproblemer generelt skal vi snakke om at “de er løsbare i polynomisk tid”, “de er NP-harde”, “de er NP-lette” osv.

Siste del av kapittel 5.1 dreier seg om å vise at de søkeproblemer som naturlig oppstår ut fra et typisk NP-komplett problem, gjerne selv blir NP-ekvivalente. Dette rettfærdiggjør på mange måter det at vi lenge holdt oss til bare desisjonsproblemer. Ved en enkel vri fikk vi her klassifisert også de søke- og optimaliseringsproblemene vi kanskje egentlig interesserte oss for, og de falt til og med i samme klasse (de NP-ekvivalente) som de litt kunstige desisjonsprobleme vi startet ut med.

Vi skal ikke her ytterligere kommentere metodene for å vise NP-ekvivalens, men henviser til oppgavene 35 og 36.

5.7 Kommentarer til kapittel 6

Hovedideen med dette kapittelet er å se på hvordan man kan forholde seg til NP-komplette problemer når man innen “rimelig tid” *må* løse dem så godt det lar seg gjøre. Den ene muligheten er da å bruke en av de eksponensielle algoritmene (vanligvis en variant av kombinatorisk søking eller av dynamisk programmering), og å forsøke å gjøre denne så effektiv som mulig med all slags avskjæring og heuristikk. For en hel del “typeproblemer” er det her masse tilgjengelig litteratur, og med forskjellige lure triks og synsmåter er det utrolig hva man kan vinne av tid, selv om algoritmene fremdeles forblir eksponensielle.

Dersom vi har et optimaliseringsproblem foreligger også en annen mulighet, nemlig å lage en algoritme som gir et “bra nok” svar for det aktuelle formålet. Også angående dette finnes en mengde litteratur, og man kan som regel også komme langt med å legge inn heuristikk som utnytter de karakteristika vi vet at nettopp våre instanser av problemet vil ha.

Det resten av kapittelet konsentrerer seg om er å forsøke å si noe mer konkret og håndfast om hvordan slike tilnærmingsalgoritmer vil oppføre seg i *verste tilfellet*. Som et mål på dette innfører boka på side 128 en del begreper som noe løslig kan karakteriseres som følger:

- $R_A(I)$: Hvor dårlig algoritmen er for instansen I .
- R_A : Hvor dårlig algoritmen er for den verst mulige instans.
- R_A^∞ : Hvor dårlig algoritmen er i verste tilfellet, om vi ser på instanser med større og større optimalverdier.

Betraktningene omkring bingepakking på sidene 124 – 127 kan taes som et eksempel på hvordan man kan gjøre betraktninger som fører fram til verdier for “ytelsesgarantiene” R_A og R_A^∞ . Merk at det å få bestemt nøyaktige verdier for disse vanligvis vil innebære meget kompliserte betraktninger, men at vi ofte med nokså løse betraktninger kan få nedre eller øvre grenser for hva R_A eller R_A^∞ kan være.

Det å begrense dem nedenfra er ofte det enkleste, da dette bare innebærer å komme opp med eksempler der algoritmen virker så og så dårlig. For R_A kan disse eksemplene være helt spesifikke, mens man for R_A^∞ må vise at vi kan finne slike eksempler med så stor optimalverdi vi bare ønsker. Det å begrense dem ovenfra innebærer derimot å vise at algoritmen i en passelig betydning “aldri kan gi verre svar enn ...”, og dette er jo en mer generell (og dermed ofte

vanskeligere) problemstilling.

Som nok et eksempel på hvordan slike betrakninger kan gjøres ser boka også på problemet “Travel Salgsmann”, og man kan her legge merke til hvordan man kan vise (for algoritmen NN) at R_A eller R_A^∞ er ∞ , ved å komme opp med (h.h.v. spesielle eller generelle) eksempler på at vi kan få svaret så dårlig vi bare vil (i forhold til det optimale). Dette ligger i siste delen av Teorem 6.3.

Betraktningene på side 133 og 134 representerer først og fremst en påminnelse om at ting ikke er så enkle som man kunne tro. Selv om en del av de transformasjonene vi har sett på tidligere i kurset vil “bevare optimale løsninger”, så er det altså langt fra slik at de vil bevare ytelsesgarantier. På grunn av vridninger i perspektivet, vil altså samme algoritmen brukt på to så beslektede problemer som nodeoverdekning og uavhengige nodemengder, gi fullstendig forskjellige ytelsesgarantier.

I betraktningene om ryggsekkproblemet på side 135 – 137 la vi mest vekt på ideen med å nedskalere tallene i problemet, for derved å få et enklere problem å løse, som dog ikke gav helt nøyaktig svar. Det overraskende er jo at man ved å kontrollere nedskaleringen nøye, kan få et “algoritme-skjema” som er polynomisk både i lengden av instansen og i den nøyaktigheten man forlanger. Mer om dette stoffet i oppgave 38.

Vi la en viss vekt på dette med “polynomsike tilnærmingsskjemaer”, og “fullstendig polynomsike tilnærmingsskjemaer” (side 137).

Begrepet $R_{MIN}(\Pi)$ som er omtalt på side 138 i kap. 6.2 uttrykker hvor “gode” (regnet som R_A^∞) polynomiske tilnærmelsesalgoritmer det er mulig å finne for det aktuelle problemet. Det er ingen enkel sak å bestemme $R_{MIN}(\Pi)$, men for noen problemer kjenner man den, eller har gode indikasjoner på hva den kan være. Dette gjelder spesielt i de tilfellene der den er 1 (vi kan få så gode algoritmer vi vil, slik som for de polynomiske tilnærmingsskjemaene) og der den er ∞ (for alle algoritmer finnes instanser der algoritmen gir så dårlig svar vi bare ønsker).

Kapittel 6.3 er hovedsaklig en oppsummering over hva vi vet om $R_{MIN}(\Pi)$ for noen kjente problemer.