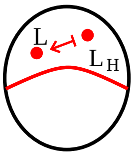
Review of unsolvability



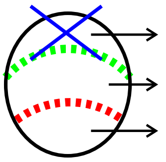
To prove unsolvability: show a reduction.

To prove solvability: show an algorithm.

Unsolvable problems (main insight)

- Turing machine (algorithm) properties
- Pattern matching and replacement (tiles, formal systems, proofs etc.)

Complexity



Unsolvable

Horrible (intractable)

Nice (tractable)

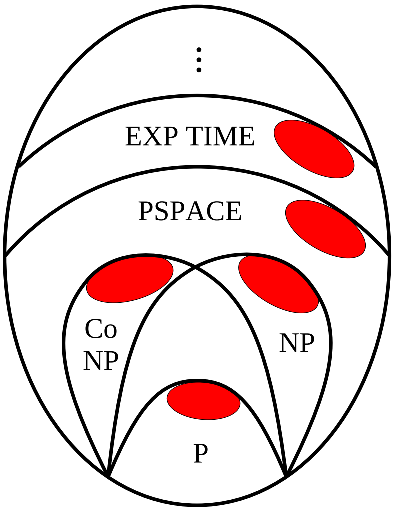
- Horrible problems are solvable by algorithms that take billions of years to produce a solution.
- Nice problems are solvable by “proper” algorithms.
- We want **techniques** and **insights**

Complexity ⟷ **resources**: time, space

↕

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

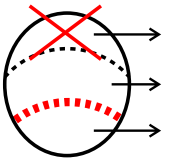
Goal



Map of classes

🔴 = complete or "hardest" problems in a class

Complexity: techniques



Impossible

Horrible (intractable)

Nice (tractable)

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 similiar 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.

Time complexity and the class $\mathcal{P}$

We say that Turing machine M **recognizes language L in time $t(n)$** if given any $x \in \sum^*$ 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 **$L \in$ TIME$(t(n))$** if there is a Turing machine M which recognizes L in time $\mathcal{O}(t(n))$.

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

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

"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).

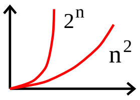
Consequence: $\mathcal{P}$ is **robust** (i.e. machine independent).

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…"
- $n^{100} \le n^{\log n}$. 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:

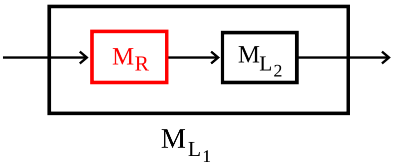


Polynomial-time simulations & reductions

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.

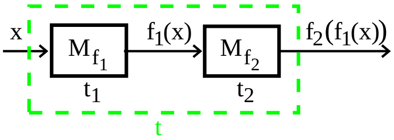
For arguments of the type

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

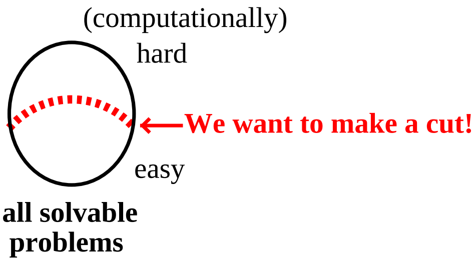
we need the following lemma:

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

Proof:



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



(computationally) hard

We want to make a cut!

easy

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.

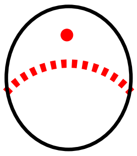
Difficulty

Finding the first truly/provably "hard" problem.

Way out

Completeness & **Hardness**

$\mathcal{NP}$-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.

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

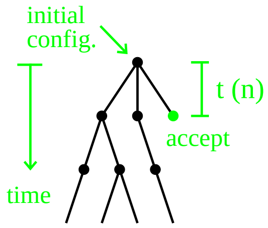
A **non-deterministic Turing machine (NTM)** is defined as deterministic TM with the following modifications:

- NTM has a **transition relation** $\triangle$ instead of transition function δ

$$\triangle : \Big\{ \big((s,0),(q_1,b,R)\big), \big((s,0),(q_2,1,L)\big), \dots \Big\}$$

- 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

We say that a non-deterministic Turing machine M **accepts language L** if there exists a halting computation 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 **$L \in$ NTIME$(t(n))$** if L is accepted by some non-deterministic Turing machine M in time $\mathcal{O}(t(n))$.

$$\mathcal{NP} = \bigcup_k \text{NTIME}\left(n^k\right)$$

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 "run" forever).

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 $\mathcal{NPC}$problems need exponential time to be solved.

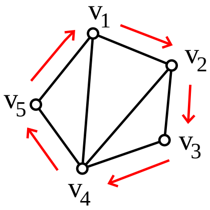
L is computationally hard ($L \in \mathcal{NP}$-complete):

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

Physiognomy

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

Example: HAMILTONICITY



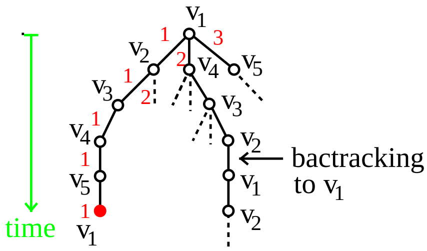
- A deterministic algorithm “must” do exhaustive search:

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

  $\searrow v_2 \to$

  $n!$ possibilities (exponentially many!)

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



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

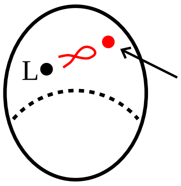
Note: A certificate is like a ticket or an ID.

Proving $\mathcal{NP}$-completeness

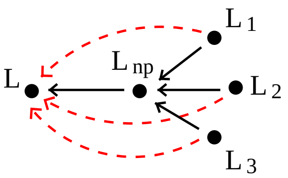
1. $L \in \mathcal{NP}$
   Prove that L has a “short certificate of membership”.

   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.



First NP-complete language



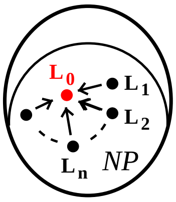
Skills to learn

- 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).

BOUNDED HALTING problem

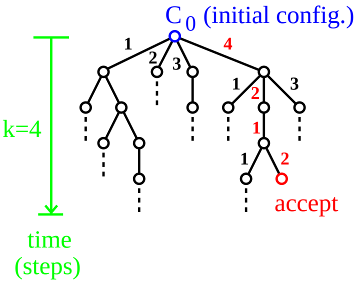
$$L_{BH} = \{(M, x, 1^k) \,|\, \text{NTM } M \text{ accepts string } x \text{ in } k \text{ steps or less}\}$$

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.*

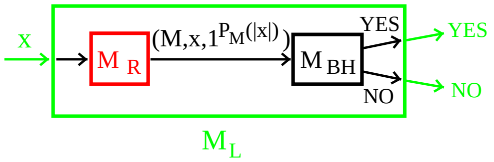
Proof:

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



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!

- $L_{BH} \in \mathcal{NP}$-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 in the length of x.

— If M_{BH} says 'YES', M_L answers 'YES'. If M_{BH} says 'NO', M_L answers 'NO'.

SATISFIABILITY (SAT)

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

Instance: A set $C = \{C_1, \ldots, C_m\}$ 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**, $T : U \to \{\text{TRUE}, \text{FALSE}\}$, 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. $I \in$ 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.

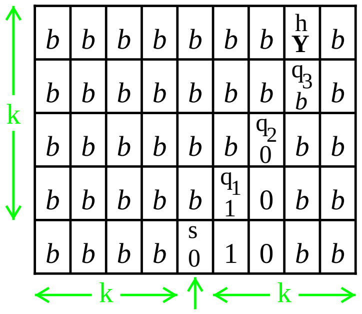
Theorem 2 (Cook 1971) SATISFIABILITY *is $\mathcal{NP}$-complete.*

Proof – main ideas:

BOUNDED HALTING		SATISFIABILITY
"There is a computation"	$\longmapsto$	"There is a truth assignment"

computation $\rightsquigarrow$ (computation) matrix

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



b	b	b	b	b	b	b	h **Y**	b
b	b	b	b	b	b	b	q_3 b	b
b	b	b	b	b	b	q_2 0	b	b
b	b	b	b	b	q_1 1	0	b	b
b	b	b	b	s 0	1	0	b	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.

tape squares ⟼ boolean variables

Ex. Square $A(2,6)$ gives variables $B(2,6,0)$, $B(2,6,b)$, $B(2,6,\genfrac{}{}{0pt}{}{q_0}{0})$, etc. – but only polynomially many.

input symbols ⟼ single-variable clauses

Ex. $A(1,5)=\genfrac{}{}{0pt}{}{s}{0}$ gives clause $\big(B(1,5,\genfrac{}{}{0pt}{}{s}{0})\big)\in C$.

Note that any satisfying truth assignment must map $B(1,5,\genfrac{}{}{0pt}{}{s}{0})$ to TRUE.

rules/templates ⟼ "if-then clauses"

Ex. $\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & d & \multicolumn{1}{c}{} \\ \hline a & b & c \\ \hline \end{array}$ gives $\Big(\big(B(i-1,j,a)\wedge B(i,j,b)$ $\wedge B(i+1,j,c)\big)\Rightarrow B(i,j+1,d)\Big)\in C$.

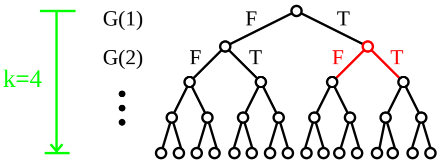
Note: $(u\wedge v\wedge w)\Rightarrow z\equiv\neg u\vee\neg v\vee\neg w\vee z$

Since the tile can be anywhere in the matrix, we must create clauses for all $2\le i\le 2k$ and $1\le j\le k$, but only polynomially many.

---

non-determinism ⟼ "choice" variables

Ex.



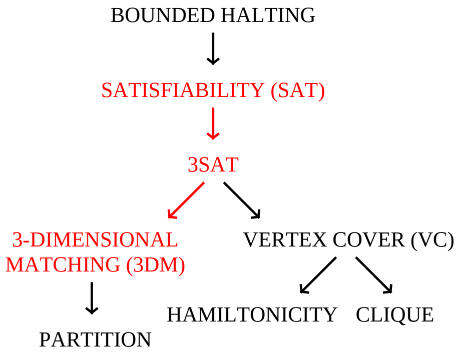
$G(t)$ tells us what non-deterministic choice was taken by the machine at step t. We extend the "if-then clauses" with k choice variables:

$$\big(G(t)\wedge\text{“a”}\wedge\text{“b”}\wedge\text{“c”}\Rightarrow\text{“d”}\big)\vee\big(\neg G(t)\wedge\cdots\big)$$

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.

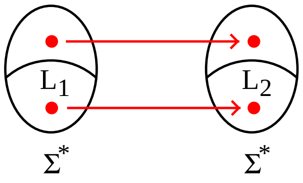
Further (basic) reductions



Polynomial-time reductions (review)

$L_1\propto L_2$ means that

- $R:\sum^*\rightarrow\sum^*$ 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

SATISFIABILITY ∝ 3-SATISFIABILITY

SAT		**3SAT**
Clauses with any number of literals	⟼	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		$C_j^{'}$
$(x_1\vee x_2\vee x_3)$	⟼	$(x_1\vee x_2\vee x_3)$
$(x_1\vee x_2)$	⟼	$(x_1\vee x_2\vee y_j),(x_1\vee x_2\vee\neg y_j)$
(x_1)	⟼	$(x_1\vee y_j^1\vee y_j^2),(x_1\vee\neg y_j^1\vee y_j^2),$ $(x_1\vee y_j^1\vee\neg y_j^2),(x_1\vee\neg y_j^1\vee\neg y_j^2)$
$(x_1\vee\cdots\vee x_8)$	⟼	$(x_1\vee x_2\vee y_j^1),(\neg y_j^1\vee x_3\vee y_j^2),$ $(\neg y_j^2\vee x_4\vee y_j^3),(\neg y_j^3\vee x_5\vee y_j^4),$ $(\neg y_j^4\vee x_6\vee y_j^5),(\neg y_j^5\vee x_7\vee x_8)$

Question: Why is this a proper reduction?