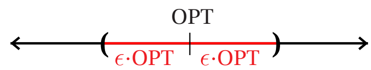
Alternative approaches to algorithm design and analysis

- **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 → average-case analysis
  - — best solution → approximation
  - — best solution → randomized algorithms

Approximation

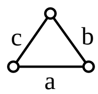


Def. 1 *Let L be an optimization problem. We say that algorithm M is a* ***polynomial-time ϵ-approximation algorithm*** *for L if M runs in polynomial time and there is a constant $\epsilon \geq 0$ such that M is guaranteed to produce, for all instances of L, a solution whose cost is within an ϵ-neighborhood from the optimum.*

Note 1: Formally this means that the **relative error** $\frac{|t_M(n) - \text{OPT}|}{\text{OPT}}$ 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.

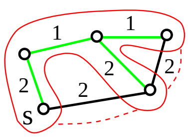
Example: TSP with triangle inequality has a polynomial-time approximation algorithm.



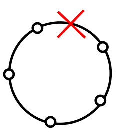
$c \leq a + b$

Algorithm TSP-△:

```
Phase I: Find a minimum spanning tree.
Phase II: Use the tree to create a tour.
```



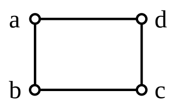
The cost of the produced solution can not be more than 2·OPT, otherwise the OPT tour (minus one edge) would be a more minimal spanning tree itself. Hence $\epsilon = 1$.



Opt. tour

Theorem 1 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.



∝

	a	b	c	d
a	2+ϵ n	1	2+ϵ n	1
b	1	2+ϵ n	1	2+ϵ n
c	2+ϵ n	1	2+ϵ n	1
d	1	2+ϵ n	1	2+ϵ n

$$K = n(= 4)$$

The **error** resulting from picking a non-edge is: Approx.solutin - 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.

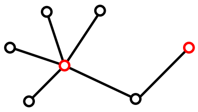
Example: VERTEX COVER

- **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;
```



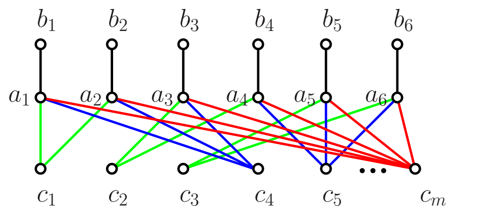
Theorem 2 *The heuristic VC-H1 is not an ϵ-approximation algorithm for* VERTEX COVER*-opt. for any fixed ϵ.*

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.



- 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:

$$\frac{|\text{VC-H1}| - |\text{OPT}|}{|\text{OPT}|} = \frac{(n \log n + n) - n}{n} = \frac{n \log n}{n} = \log n \neq \epsilon$$

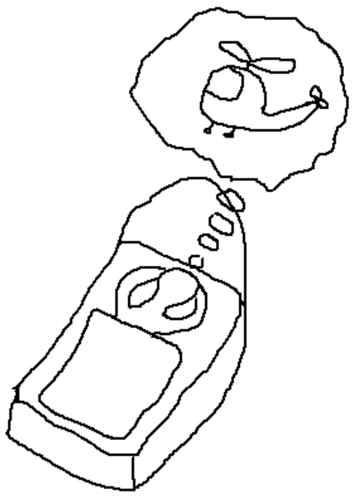
- The relative error **grows as a function of n**.

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, $|\text{VC-H2}| \leq 2 \cdot |\text{OPT}|$.
- So VC-H2 is a polynomial-time ϵ-approximation algorithm for VC with $\epsilon = 1$.
- Surpisingly, this "stupid-looking" algorithm is the best (worst case) approximation algorithm known for VERTEX COVER-opt.

Average-case analysis & algorithms



~~Worst~~ case

- **Problem = (L, P_r)** where P_r is a probability function over the input strings: $P_r : \sum^* \to [0,1]$.
- $\sum_{x \in \sum^*} P_r(x) = 1$ (the probabilities must sum up to 1).
- **Average time** of an algorithm:

$$T_A(n) = \sum_{\{x \in \sum^* \mid |x| = n\}} T_A(x) P_r(x)$$

- **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).

Random graphs

Uniform probability model (UPM)

- Every graph G has equal probability
- If the number of nodes = n, then $P_r(G) = \frac{1}{\#\text{graphs}} = \frac{1}{2^{\binom{n}{2}}}$, 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 probabilility p of occuring
- 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:

$$P_r(G) = \left(\frac{1}{2}\right)^m \cdot \left(\frac{1}{2}\right)^{\binom{n}{2}-m} = \frac{1}{2^{\binom{n}{2}}}$$

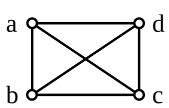
- IEPM is easier to work with

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 3 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.

- 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):

$$\begin{aligned}\sum_{i=1}^{\infty} iq^{i-1} &= \frac{\delta}{\delta q}\left(\sum_{i=1}^{\infty} q^i\right) = \frac{\delta}{\delta q}\left(\frac{q}{1-q}\right) \\ &= \frac{1}{(1-q)^2}\end{aligned}$$

Conlusion: 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 $p = 5/n$.

0-1 Laws

as a link between probabilistic and deterministic thinking.

Example: “Almost all” graphs are

- not 3-colorable
- Hamiltonian
- connected
- ...

Def. 2 *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* ($\lim_{n\to\infty}$).

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. $P_r(\exists$ 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
  $\leq \mathcal{O}(n) + \mathcal{O}(n^2)\, P_1(n) + \mathcal{O}(2^{2n})\, P_1(n)P_2(n)$
  $= \mathcal{O}(n)$ if $P_1(n) \cdot \mathcal{O}(n^2) = \mathcal{O}(n)$
  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.

Randomized computing

Machines that can **toss coins** (generate random bits/numbers)

- Worst case paradigm
- Always give the correct (best) solution

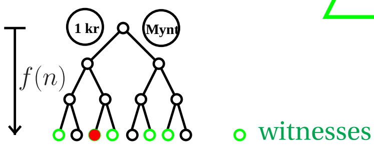
Randomized algorithms

Idea: Toss a coin & simulate non-determinism

Example 1: Proving polynomial non-identities

$$(x+y)^2 \overset{?}{\neq} x^2 + 2xy + y^2$$
$$\overset{?}{\neq} x^2 + y^2$$

- 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 LHS $\neq$ RHS, then we know that the polynomials are different.
  - — If LHS $=$ 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.



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 then the probability that a meteorite hits the computer while the program is running!

Metaheuristics

Simulated Annealing

- Analogy with physical annealing
- 'Temperature' T, annealing schedule
- 'Bad moves' with probability $\exp(-\delta f/T)$

Genetic algorithms

- Analogy with Darwinian evolution
- 'individuals', 'fitness', 'cross breeding'

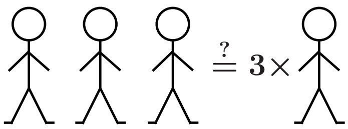
Neural Networks

- Analogy with human mind
- 'neurons', 'learning'

Taboo search

- Analogy with culture
- adaptive memory, responsive exploration

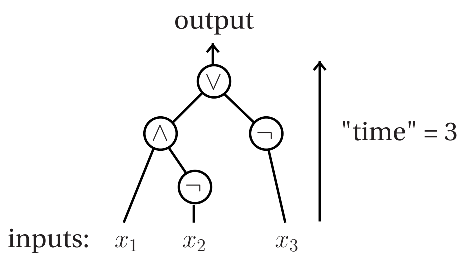
Parallel computing



- some problems can be efficiently parallelized
- some problems seems inherently sequential

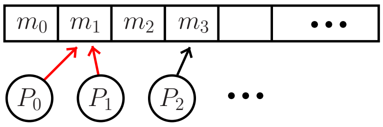
Parallel machine models

- **Alternating TMs**
- **Boolean Circuits**



— Boolean Circuit complexity: **"time"** (length of longest directed path) and **hardware** (# of gates)

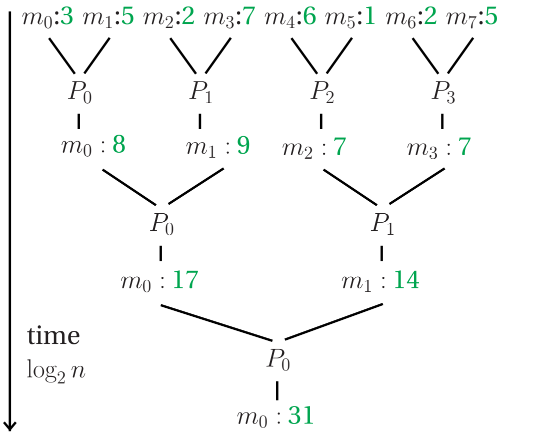
- **Parallell Random Access Machines (PRAMs)**



— 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.

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 $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$.

Bad news

Theorem 4 *With polynomial many processors*

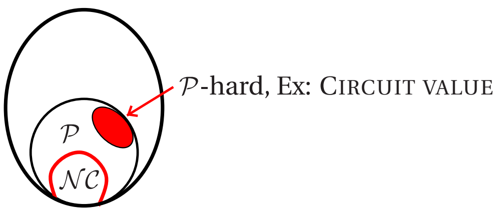
parallel poly. time = sequential poly. time

Proof:

- 1 processor can simulate one step of m processors in sequential time $t_1(m) = \mathcal{O}(m)$
- Let $t_2(n)$ be the polynomial parallel time of the computation. If m is polynomial then $t_1(m) \cdot t_2(n) =$ polynomial.

Parallel complexity classes

Def. 3 *A language is said to be in class $\mathcal{NC}$ if it is recognized in polylogarithmic, $\mathcal{O}(\log^k(n))$, parallel time with uniform polynomial hardware.*



- $\mathcal{P} \stackrel{?}{=} \mathcal{NC}$