

Introduction to Algorithms and Complexity

IN3130 Algorithms and Complexity



Introduction to Algorithm Theory

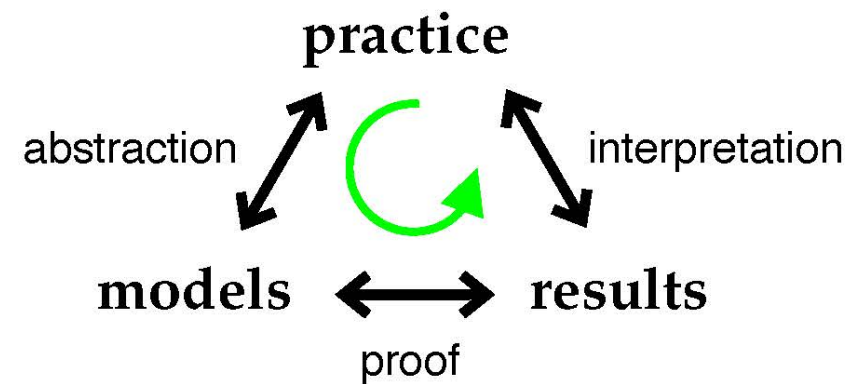
Overview

- Our approach - modeling
- The subject matter - what is this all about
- Historical introduction
- How to model problems
- How to model solutions

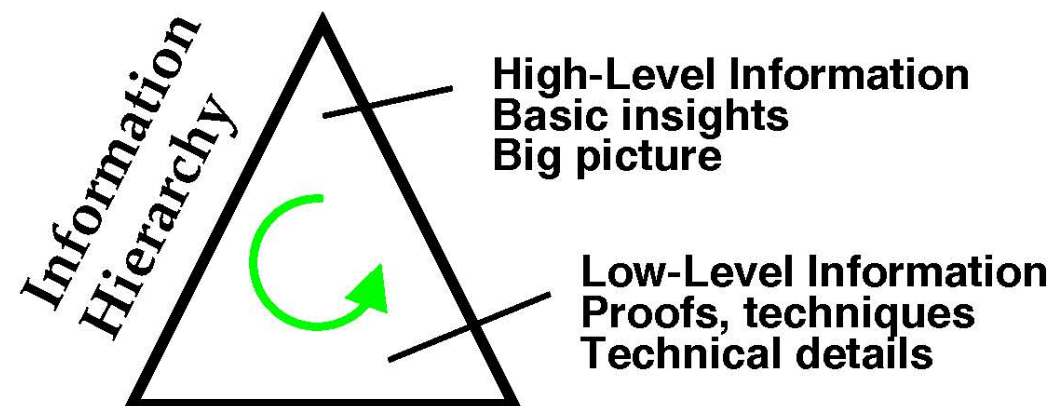


Our approach

Modeling



Perspective



Lectures → Mainly high-level understanding

Group sessions → Practice skills: proofs, problems

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

Subject matter

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



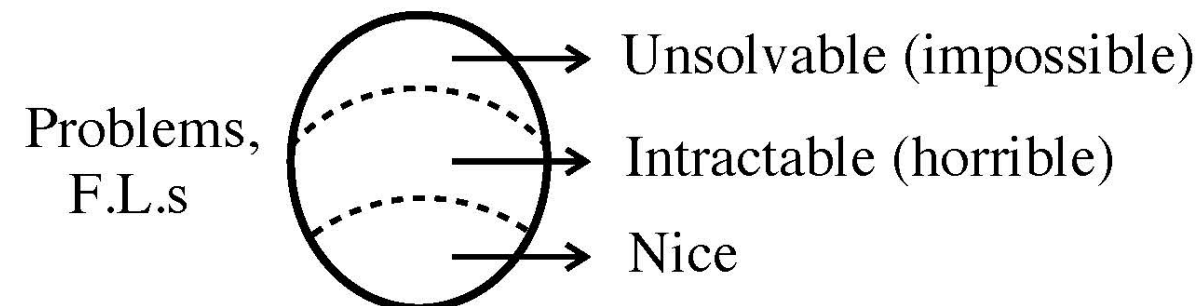
abstraction
formalisation
modeling

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

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

Solutions \rightsquigarrow algorithms \rightsquigarrow Turing machines

Efficiency \rightsquigarrow complexity \rightsquigarrow complexity classes





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?



Applications of Euclid's algorithm

From Wikipedia:

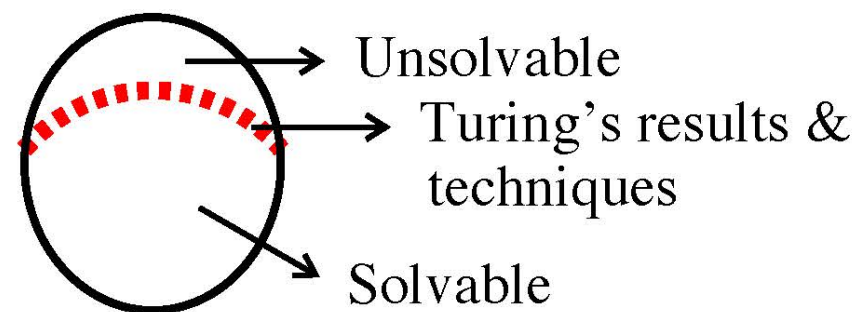
“The Euclidean algorithm has many theoretical and practical applications. It may be used to generate almost all the most important traditional musical rhythms used in different cultures throughout the world. It is a key element of the RSA algorithm, a public-key encryption method widely used in electronic commerce. It is used to solve Diophantine equations, such as finding numbers that satisfy multiple congruences (Chinese remainder theorem) or multiplicative inverses of a finite field. The Euclidean algorithm can also be used in constructing continued fraction, in the Sturm chain method for finding real roots of polynomials, and in several modern integer factorization algorithms. Finally, it is a basic tool for proving theorems in modern number theory, such as Lagrange's four-square theorem, and the fundamental theorem of arithmetic (unique factorization).”



Origins of undecidability theory

\exists algorithm? \rightarrow metamathematics

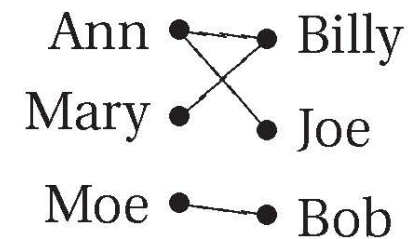
- K. Gödel (1931): nonexistent theories
- A. Turing (1936): nonexistent algorithms
(article: “On computable Numbers ...”)





Origins of complexity theory

- 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!

Rough complexity analysis of trivial algorithm

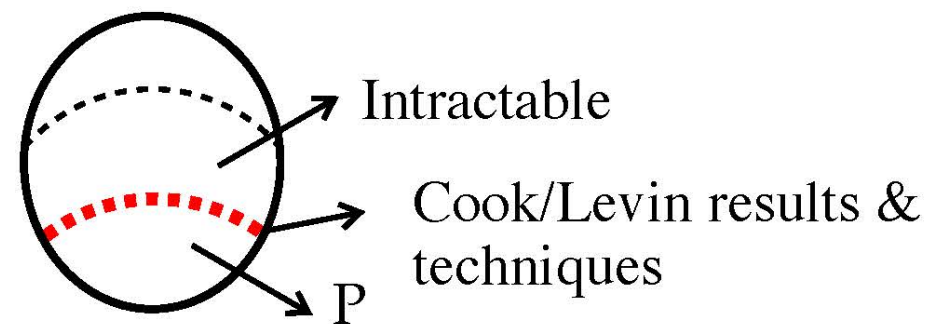
- $n = 100$ boys
- $n! = 100 \times 99 \times \dots \times 1 \geq 10^{90}$ possibilities
- assume $\leq 10^{12}$ possibilities tested per second
- $\leq 10^{12+4+2+3+2} \leq 10^{23}$ tested per century
- running time of trivial algorithm for $n = 100$ is $\geq 10^{90-23} = 10^{67}$ centuries!

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



Edmonds: My 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**







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.



Def. 1 *Alphabet* = finite set of symbols

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

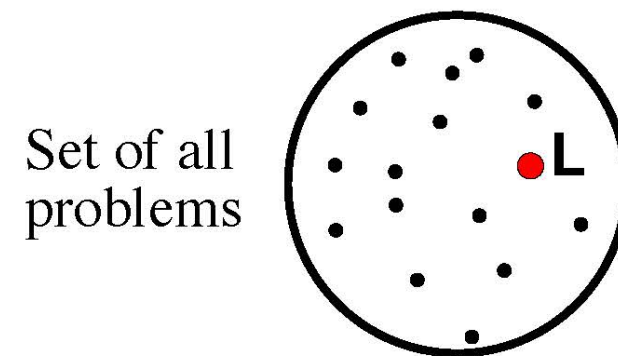
Coding: binary \leftrightarrow ASCII

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

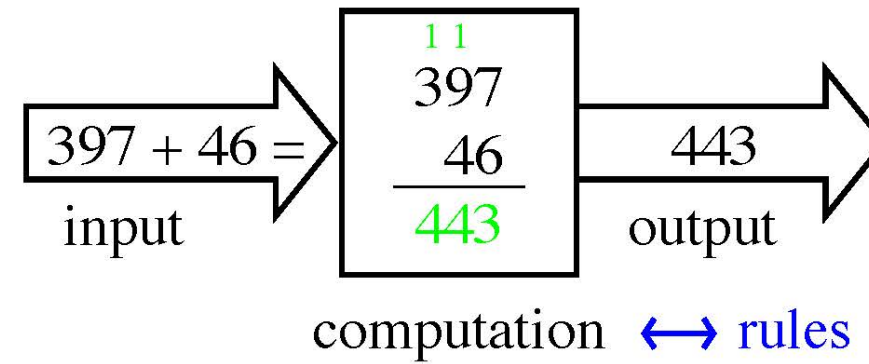
$\Sigma^* = \{\epsilon, 0, 1, 00, 01, \dots\}$ — in **lexicographic order**

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

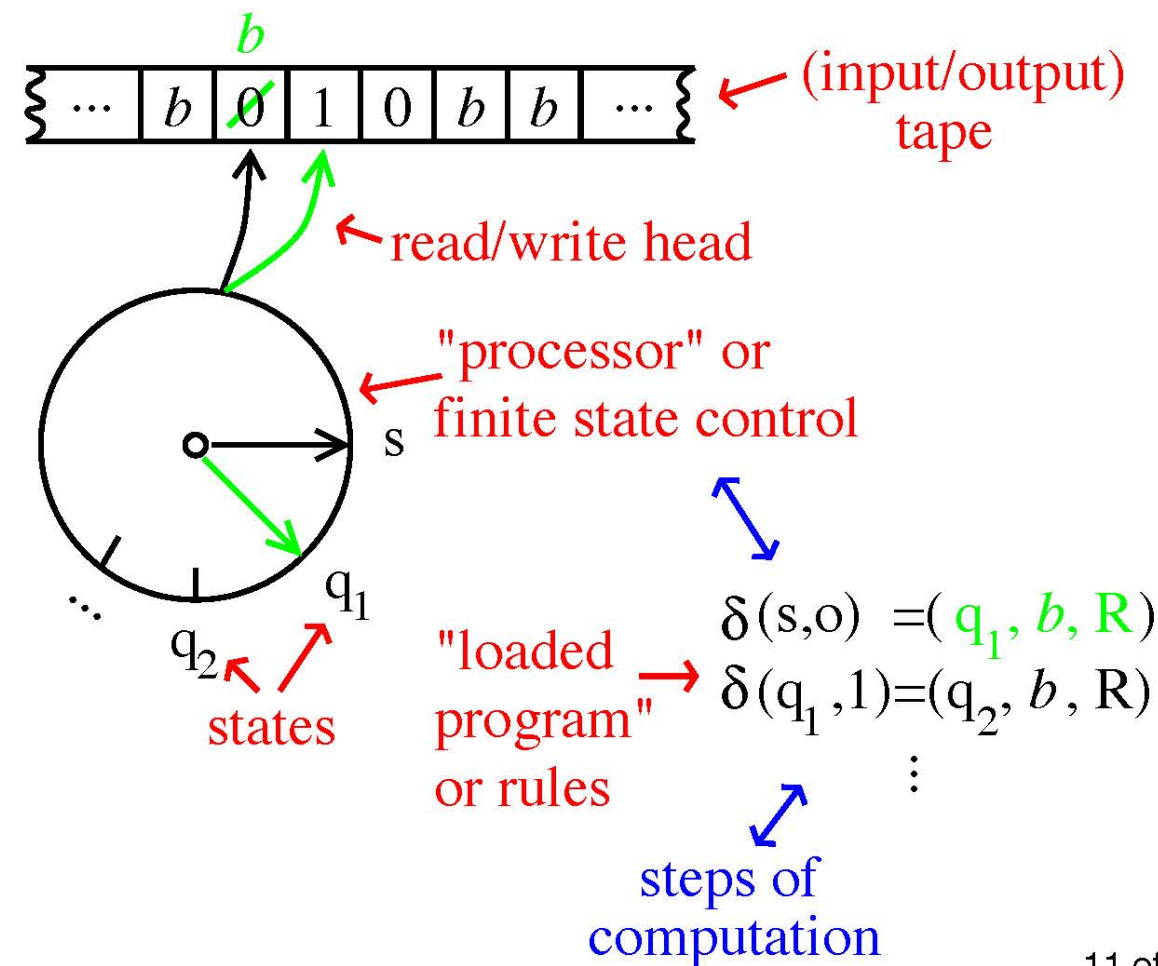
L is the set of all “YES-instances”.



Algorithm



Turing machine – intuitive description





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

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

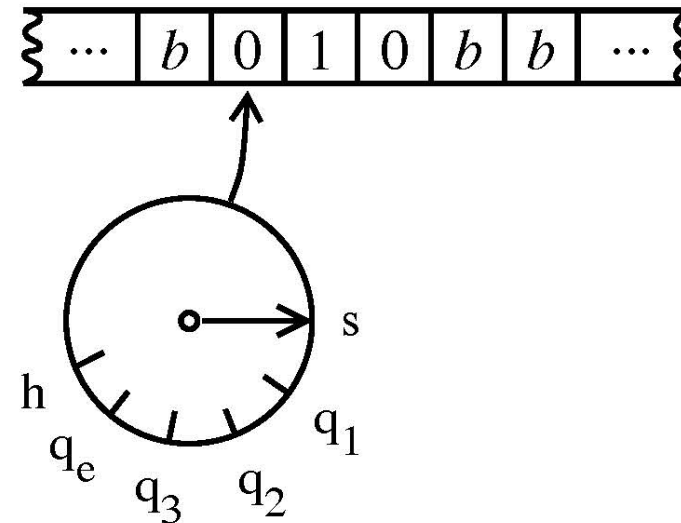
Language L is **(Turing) decidable** if (and only if) there is a Turing machine which decides it.

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

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

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")



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 (McCulloch-Pitts neuron).

Uncomputability

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

Undecidability

IN3130 Algorithms and Complexity

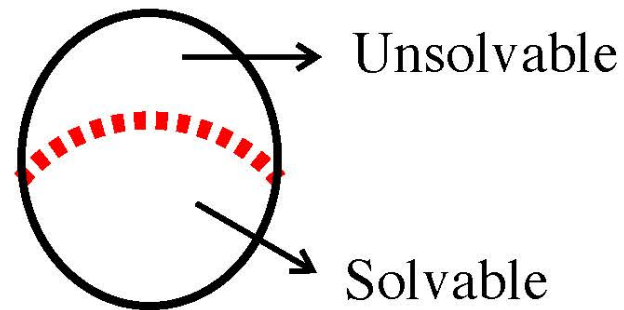


Review

Objective

techniques — how to prove that a problem is unsolvable

insights — what sort of problems are unsolvable



unsolvable
(by algorithms)
problems



undecidable
languages

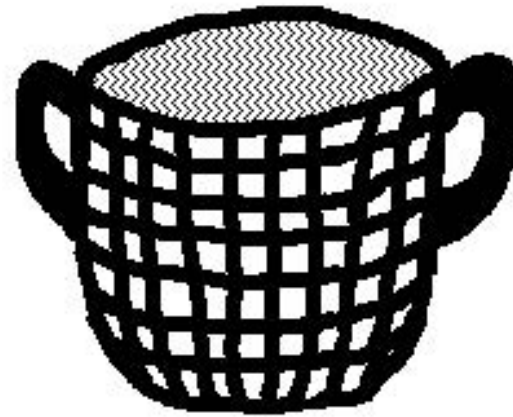
solvable
problems



decidable
languages



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)



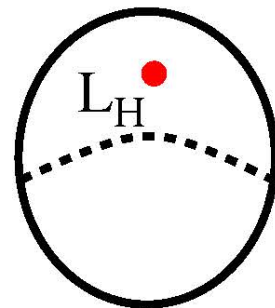
Techniques

To prove that

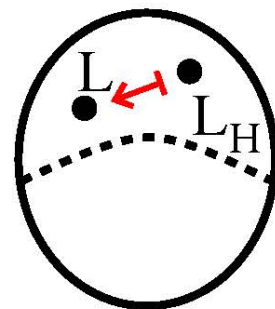
- **L is solvable:** Show an algorithm
- **L is unsolvable:** Difficulty: Cannot check all the algorithms in the basket. Cannot even see most of them, because they have not yet been constructed ...

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 .





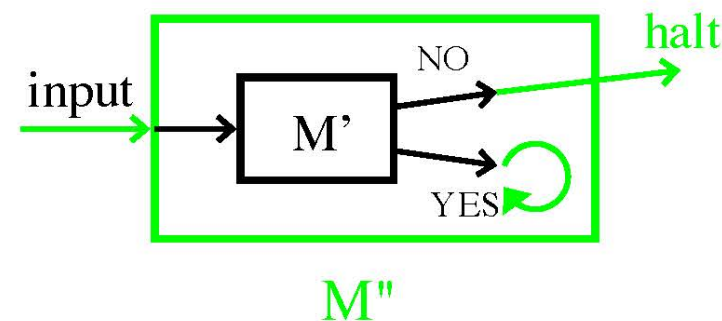
Step 1: HALTING is unsolvable

Def. 1 (HALTING)

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

Theorem 1 *The Halting Problem is undecidable.*

Proof (by **diagonalization**): Given a Turing machine M' that decides L_H we can construct a Turing machine M'' as follows:



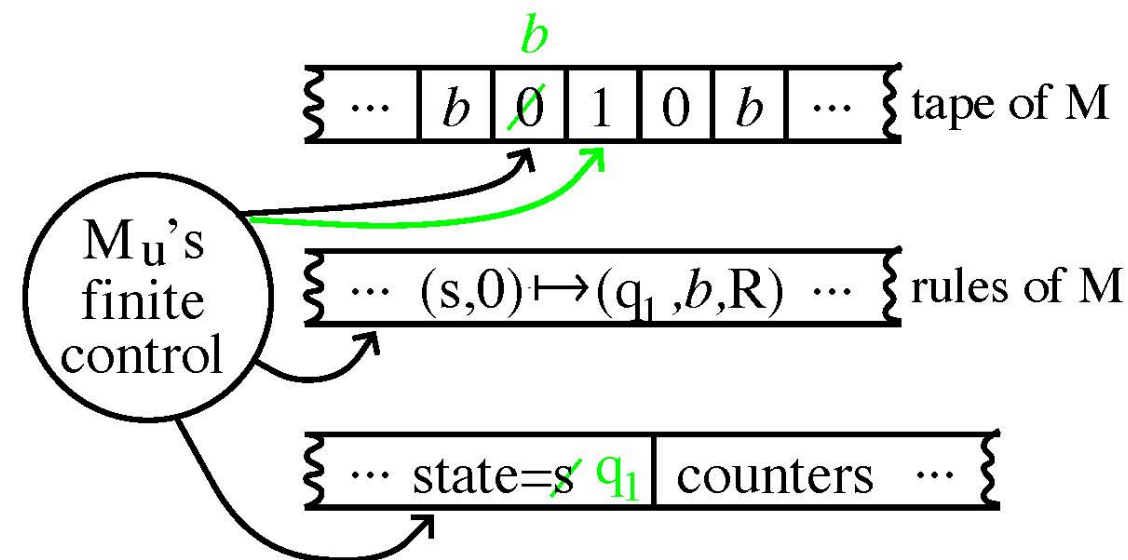
QUESTION: What does M'' do when given M'', M'' as input?

CONCLUSION: Since the assumption that M' exists leads to a contradiction (i.e. an impossible machine), it must be false.



The universal Turing machine M_u

- 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 :





Alternative proof of Theorem 1:

	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!



Meaning

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

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

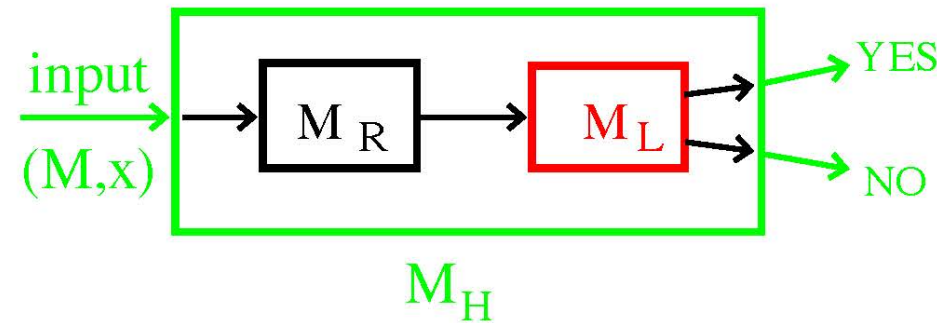
$L_2 = \{K \mid \text{There are } K \text{ consecutive zeros in the decimal expansion of } \Pi\}$

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.

Reductions



Meaning of a reduction

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

How the reduction goes

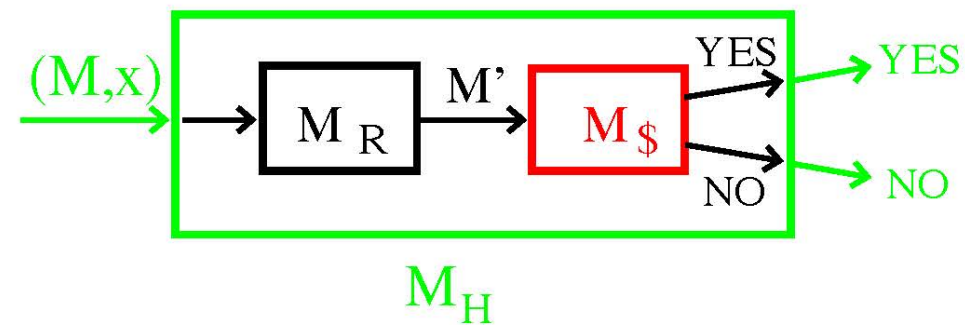
Image (an old riddle): You are standing at a crossroad deep in the forest. One way leads to the hungry crocodiles, the other way to the castle with the huge piles of gold. In front of you stands one of the two twin brothers. One of them always lies, the other always tells the truth. You can ask one question.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

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 an arbitrarily chosen action!



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

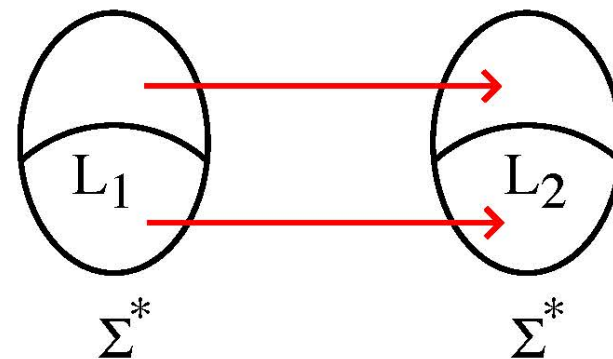
Given a reduction from L_1 to L_2 . Then M_R computes a function

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

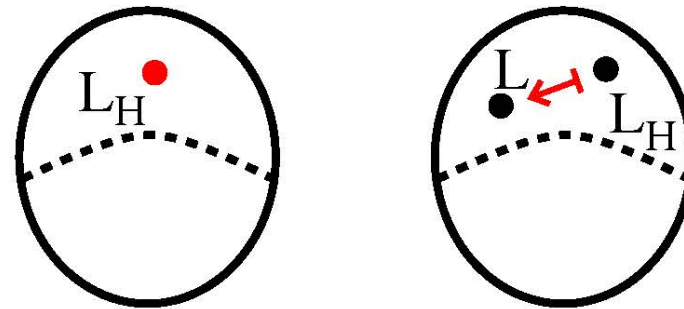
which is such that

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

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

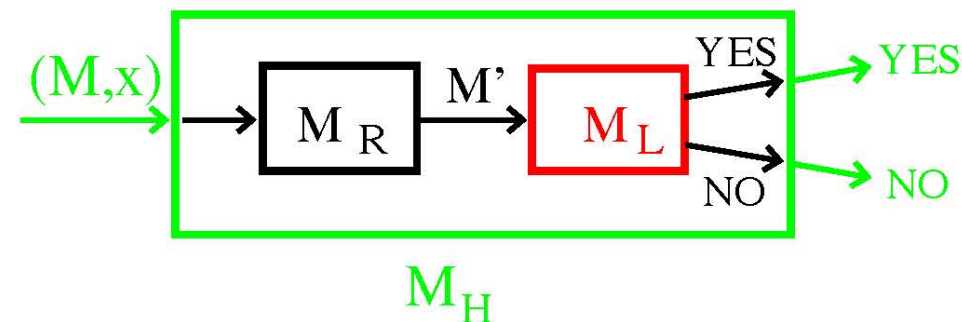


Undecidability in a Nutshell



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

Reductions



M' :

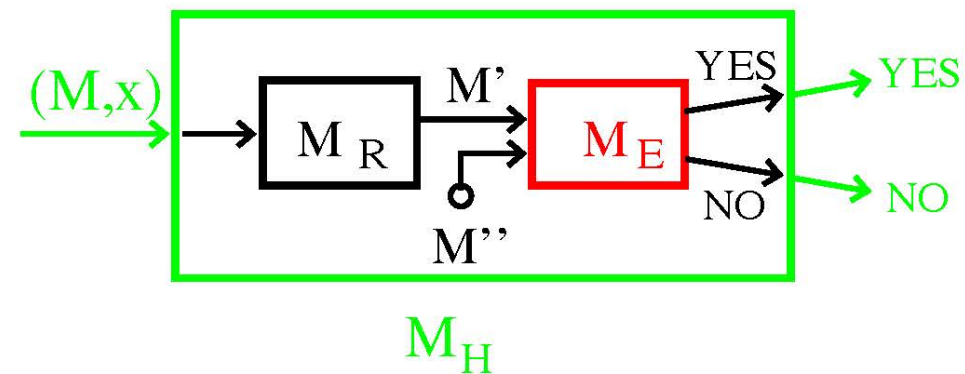
Simulate M on input x ;
Do <ACTION>;



Example

Theorem 2 *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 .



A solvable problem

$L_s = \{M_s \mid M_s \text{ (eventually) moves its R/W head}$
when started with a blank tape}

“Proof” that L_s is undecidable:

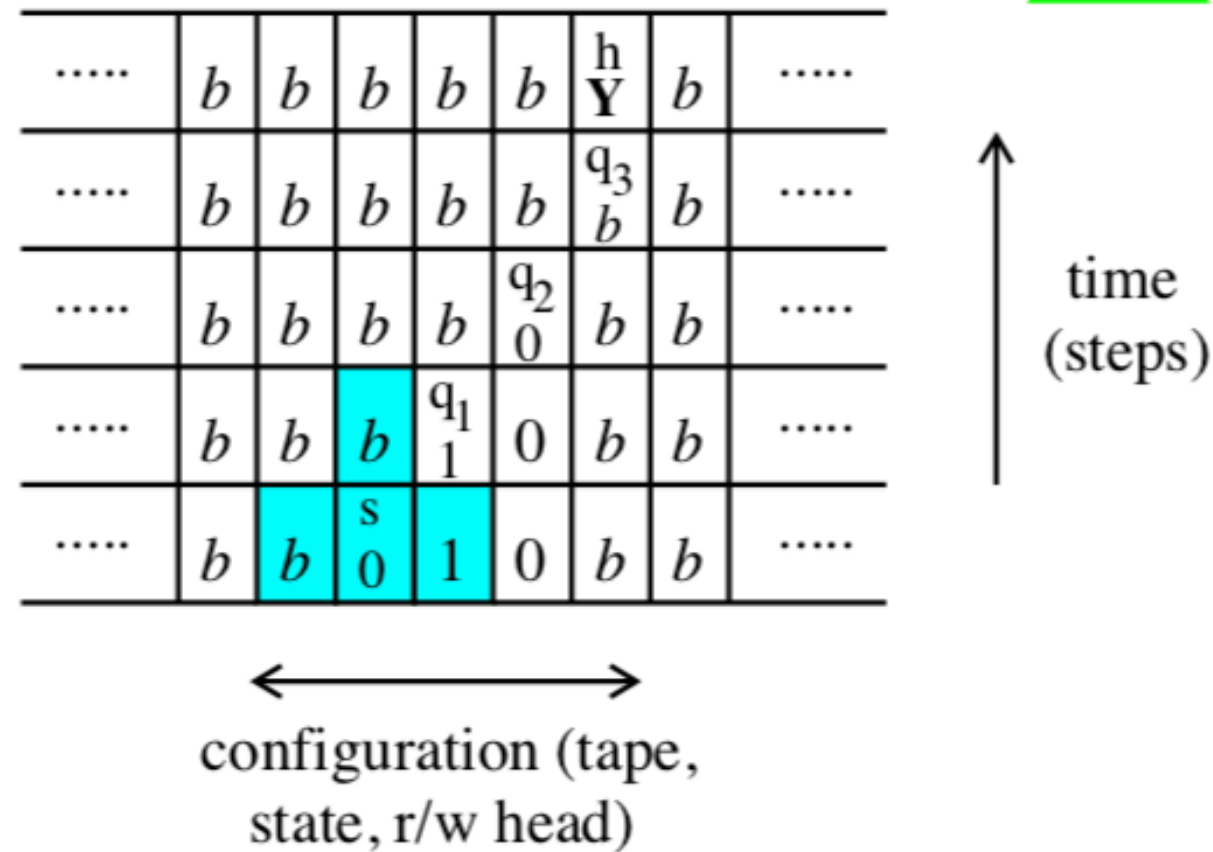
Simulate M on input x ;
Move the R/W head;

“Proof” that L_s is decidable:

Simulate M_s on empty string as input;
for $|\Gamma| \times |Q|$ steps;

NP-Completeness

IN3130 Algorithms and Complexity



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

$$\delta(s, 0) = (q_1, b, R) \text{ is } \begin{array}{|c|} \hline b \\ \hline \end{array} \begin{array}{|c|c|c|} \hline s & 0 & 1 \\ \hline \end{array} \text{ and } \begin{array}{|c|} \hline q_1 \\ \hline \end{array} \begin{array}{|c|c|c|} \hline s & 1 & 0 \\ \hline \end{array}$$

$$\text{but also } \begin{array}{|c|} \hline Y \\ \hline \end{array} \begin{array}{|c|c|c|} \hline X & Y & s \\ \hline \end{array} \text{ and } \begin{array}{|c|} \hline b \\ \hline \end{array} \begin{array}{|c|c|c|} \hline Y & s & Z \\ \hline \end{array} \text{ and } \begin{array}{|c|} \hline q_1 \\ \hline \end{array} \begin{array}{|c|c|c|} \hline s & Z & W \\ \hline \end{array}$$

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

$$\text{We also have } \begin{array}{|c|} \hline Y \\ \hline \end{array} \begin{array}{|c|c|c|} \hline X & Y & Z \\ \hline \end{array} \text{ for all } X, Y, Z \in \{1, 0, b\}.$$

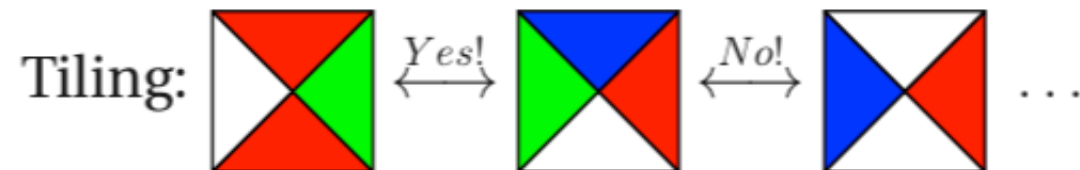
Halting configuration

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

I

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**

$$L_H^C \longmapsto L_T$$



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 $((()))$:

$$\mathbf{S} \xrightarrow[G]{R_1} (\mathbf{S}) \xrightarrow[G]{R_2} ((\mathbf{S})) \xrightarrow[G]{R_1} (((\mathbf{S}))) \xrightarrow[G]{R_3} (((\epsilon)))$$

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.



Can machines think?

Example: Theorem proving in a formal system

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

Question: Can algorithms prove/verify theorems?

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

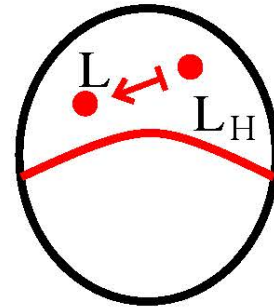
Algorithms can also **enumerate** theorems.

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

Question: What about automatic program correctness proving?



Review of unsolvability



To prove unsolvability: show a reduction.

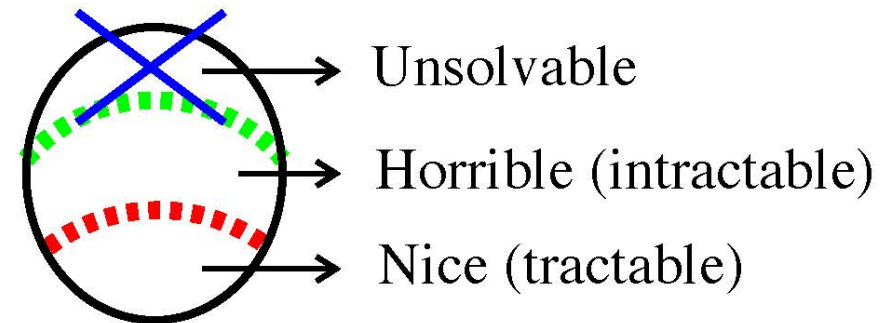
To prove solvability: show an algorithm.

Unsolvable problems (main insight)

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



Complexity



- 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

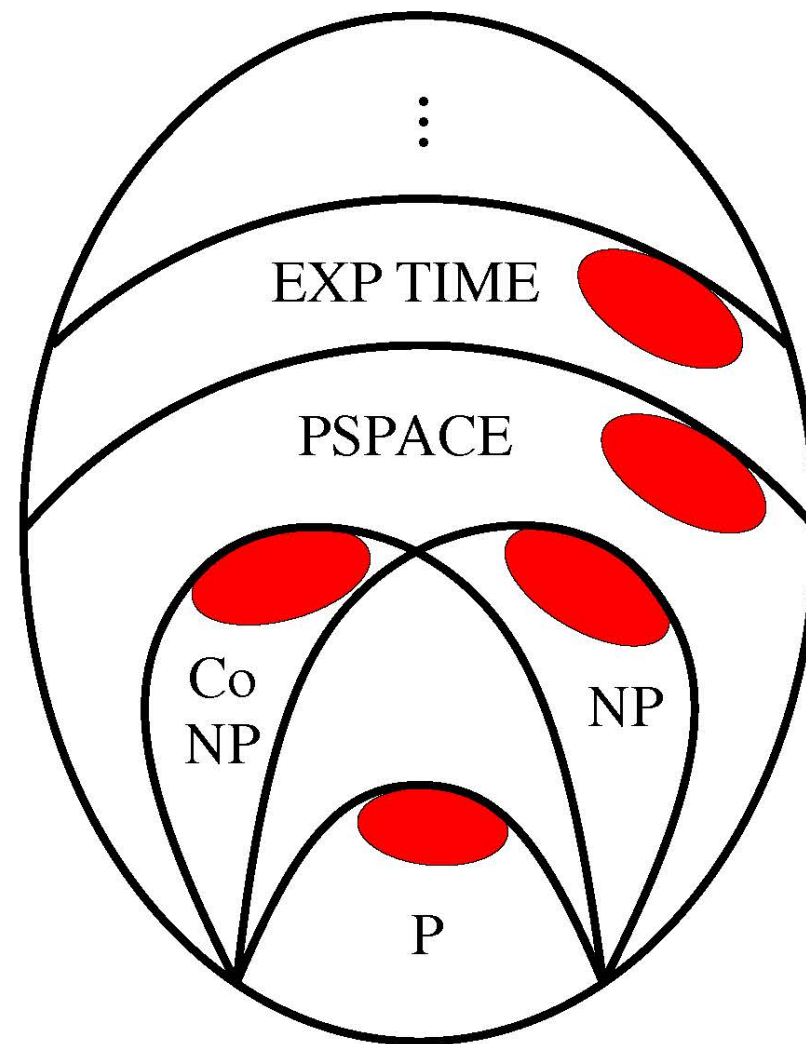


complexity classes:

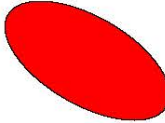
P(olynomial time), NP-complete,
Co-NP-complete, Exponential time,
PSPACE, ...



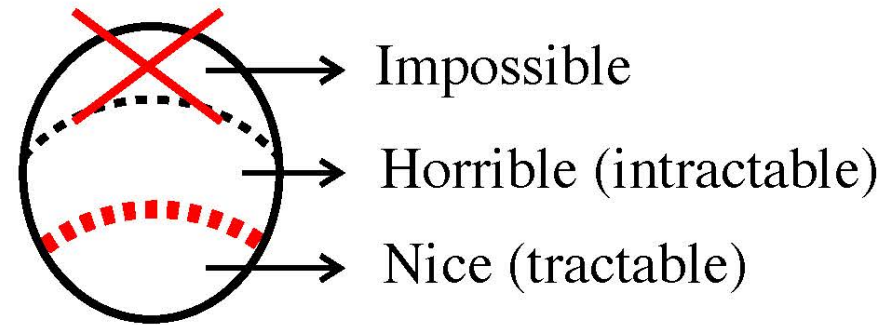
Goal



Map of classes

 = complete or "hardest" problems in a class

Complexity: techniques



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.



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 $L \in \text{TIME}(t(n))$ 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 \rightsquigarrow Real-world
complexity difficulty

Feasible \rightsquigarrow Polynomial-time
solution algorithm

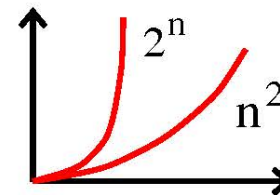
- $t(n) \rightsquigarrow \mathcal{O}(t(n))$

Argument: “for large-enough $n...$ ”

- $n^{100} \leq n^{\log n}$. 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:

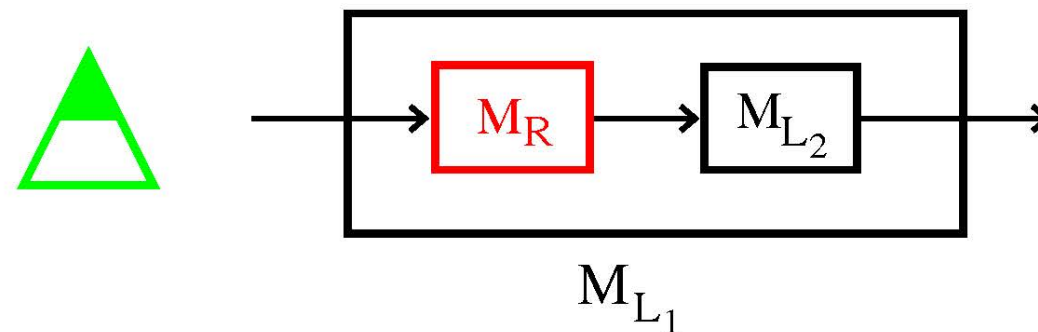


Polynomial-time simulations & reductions

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 .



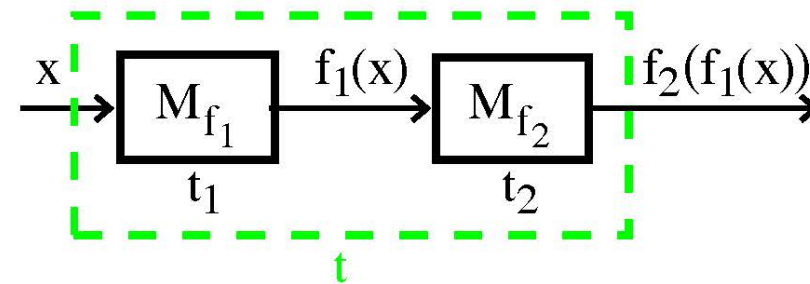
For arguments of the type

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

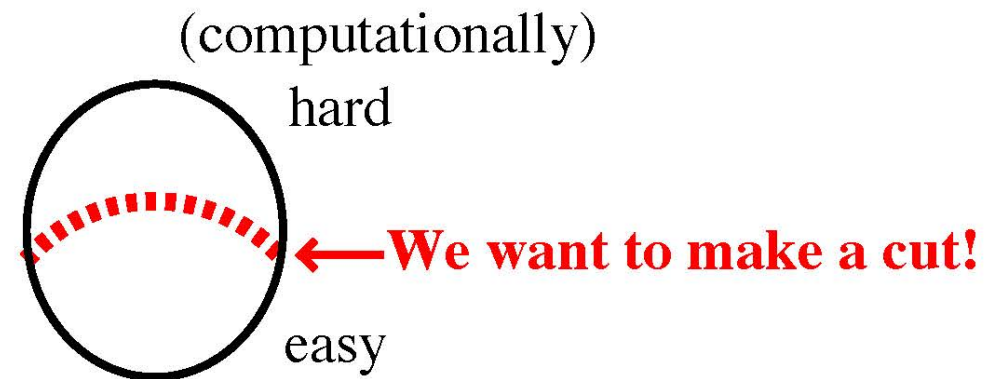
we need the following lemma:

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

Proof:



- $|f_1(x)| \leq t_1(|x|)$ 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.



**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 .

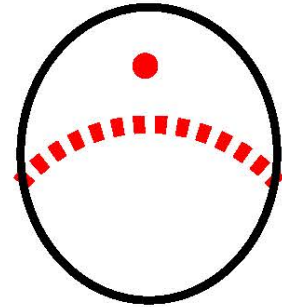
Difficulty

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)

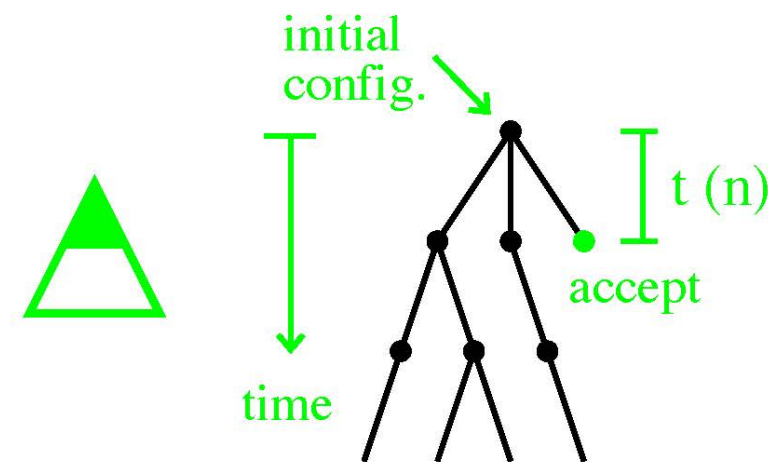
A **non-deterministic Turing machine (NTM)** is defined as deterministic TM with the following modifications:

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

$$\Delta : \left\{ ((s, 0), (q_1, b, R)), ((s, 0), (q_2, 1, L)), \dots \right\}$$

- 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 $L \in \mathbf{NTIME}(t(n))$ 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 "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{NP} problems need exponential time to be solved.

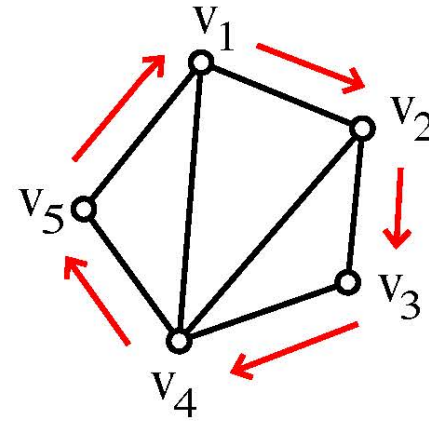
L is computationally hard ($L \in \mathcal{NP}$ -complete):

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

Physiognomy

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

Example: HAMILTONICITY



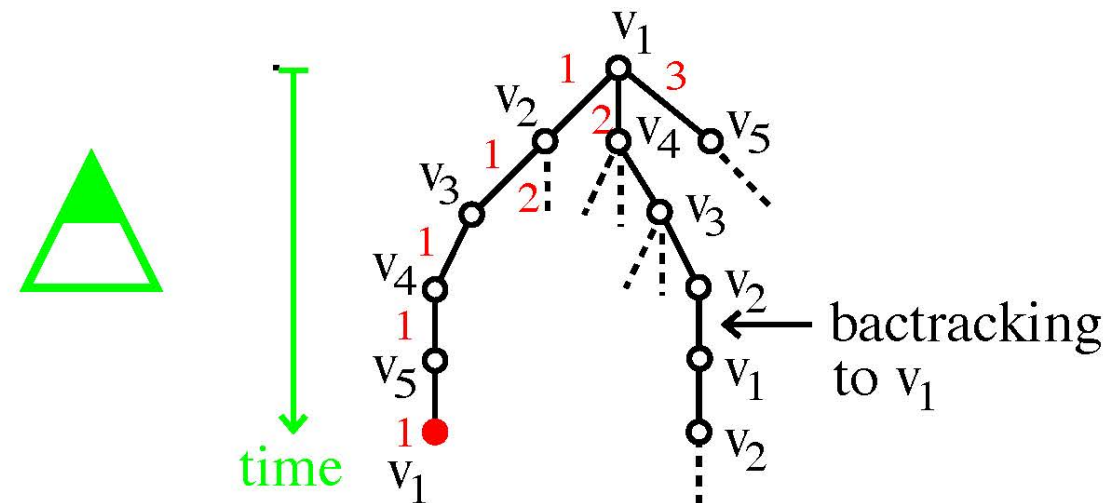
- A deterministic algorithm “must” do exhaustive search:

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

$\rightarrow v_2 \rightarrow$

$n!$ 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.



Proving \mathcal{NP} -completeness

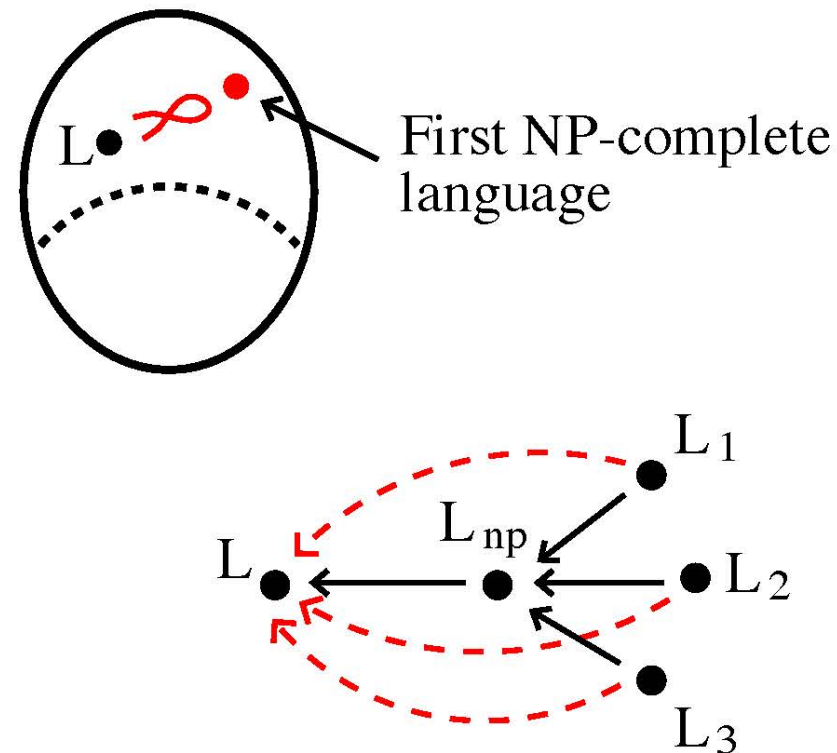
1. $L \in \mathcal{NP}$

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.





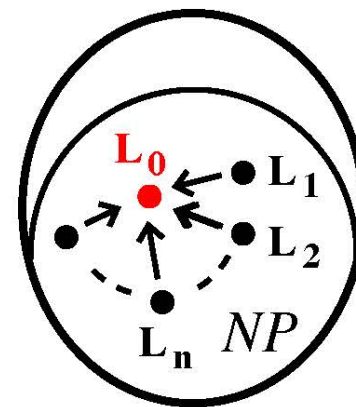
Skills to learn

- 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

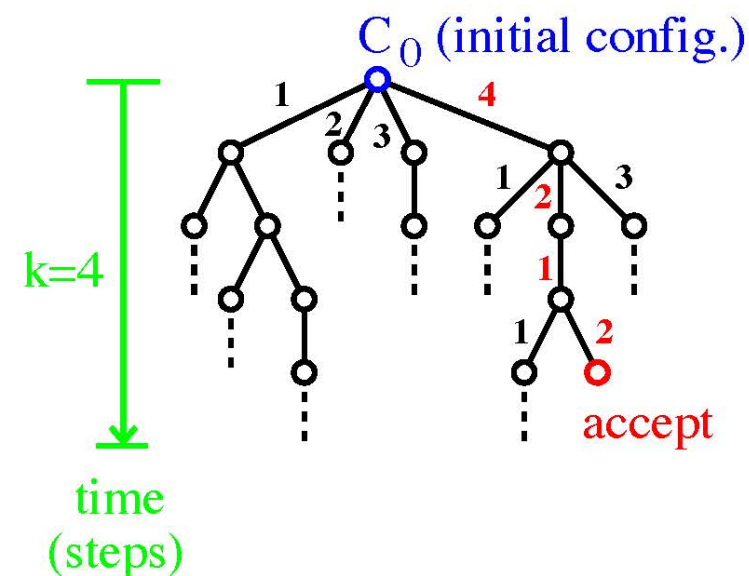
$$L_{BH} = \{(M, x, 1^k) \mid \text{NTM } M \text{ accepts string } x \text{ in } k \text{ steps or less}\}$$

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.

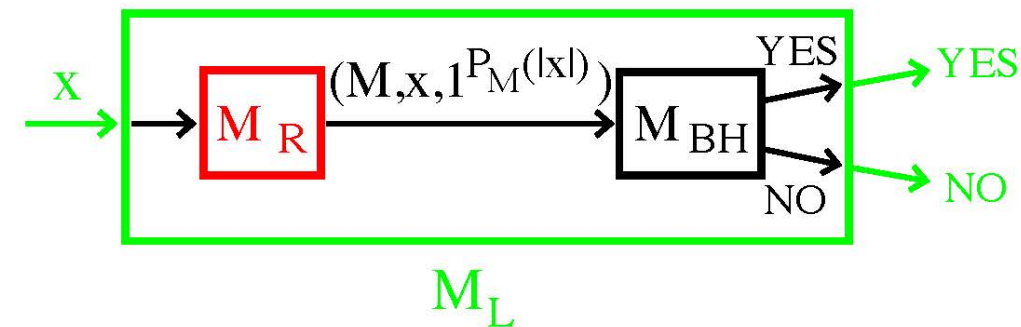
Proof:

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



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!

- $L_{BH} \in \mathcal{NP}$ -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 $C = \{C_1, \dots, C_m\}$ 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.



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

Proof – main ideas:

BOUNDED HALTING

“There is a
computation”

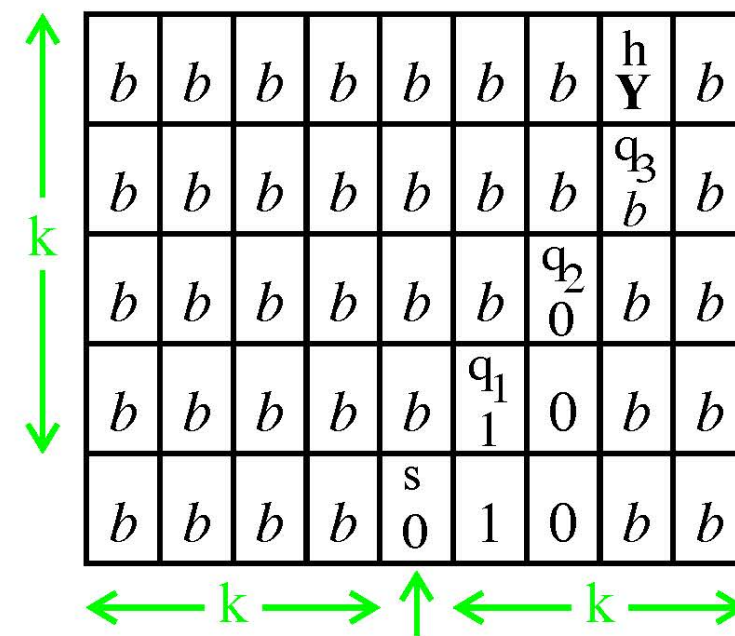


SATISFIABILITY

“There is a
truth assignment”

computation \rightsquigarrow (computation) matrix

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



b	b	b	b	b	b	b	$\frac{h}{Y}$	b
b	b	b	b	b	b	b	$\frac{q_3}{b}$	b
b	b	b	b	b	b	$\frac{q_2}{0}$	b	b
b	b	b	b	b	$\frac{q_1}{1}$	0	b	b
b	b	b	b	$\frac{s}{0}$	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 \mapsto boolean variables

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

input symbols \mapsto single-variable clauses

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

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

rules/templates \mapsto “if-then clauses”

Ex.

	d	
a	b	c

 gives $\left((B(i-1, j, a) \wedge B(i, j, b) \wedge B(i+1, j, c)) \Rightarrow B(i, j+1, d) \right) \in C$.

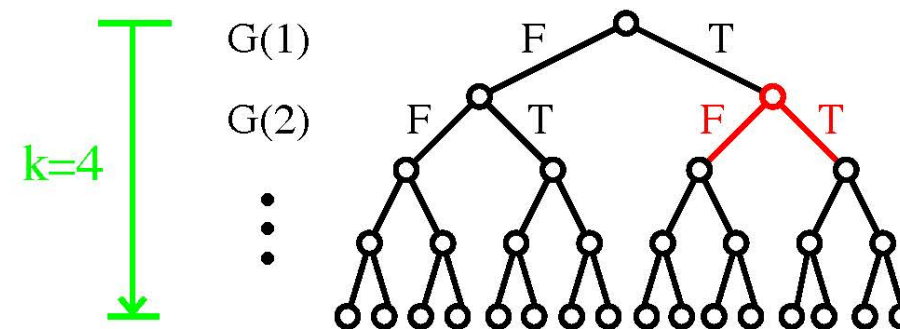
Note: $(u \wedge v \wedge w) \Rightarrow z \equiv \neg u \vee \neg v \vee \neg w \vee z$

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.



non-determinism \mapsto “choice” variables

Ex.



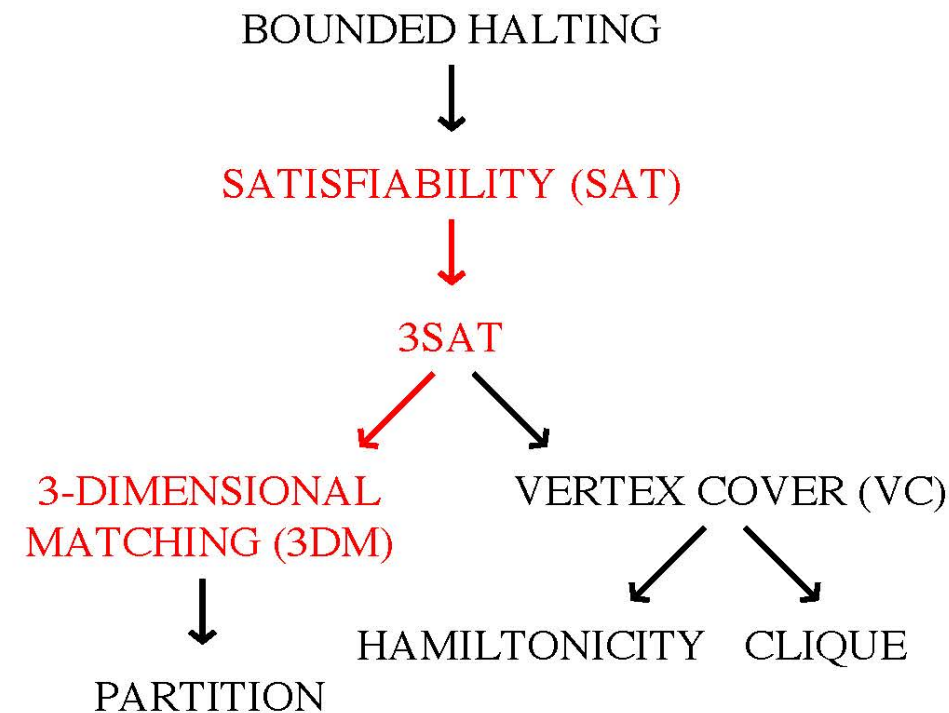
$G(t)$ tells us what non-deterministic choice was taken by the machine at step t . We extend the “if-then clauses” with k choice variables:

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

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.

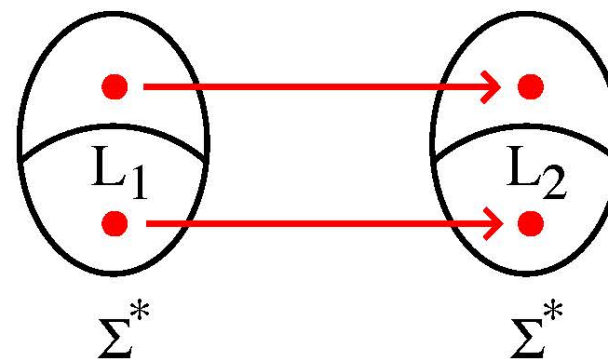
Further (basic) reductions



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



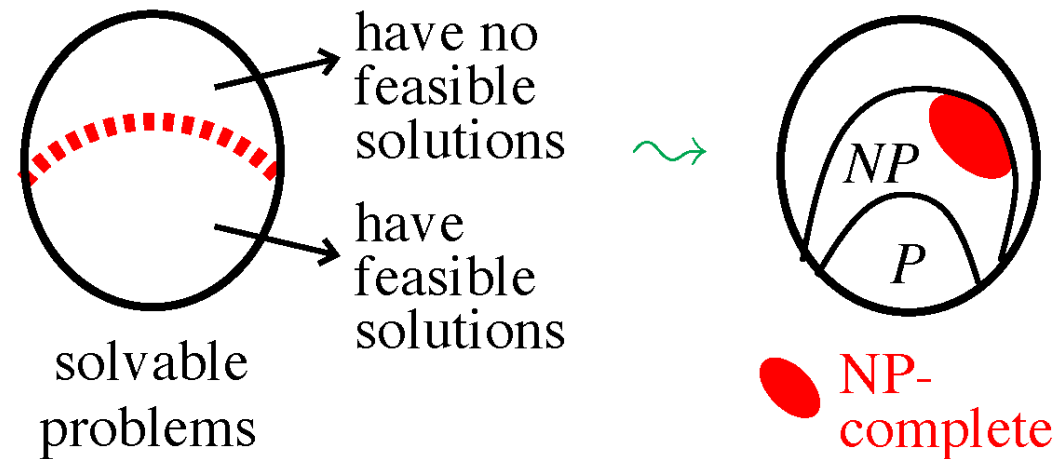
- $R \in P_f$, i.e. $R(x)$ is polynomial computable

Proving NP- completeness

IN3130 Algorithms and Complexity



NP-completeness (review)



$$L \in \mathcal{NPC} \Leftrightarrow \begin{array}{l} L \in \mathcal{NP} \text{ and} \\ L \in \mathcal{NP}\text{-hard} \end{array}$$

Today: Proving \mathcal{NP} -completeness

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

[†] polynomial (length, time ...)



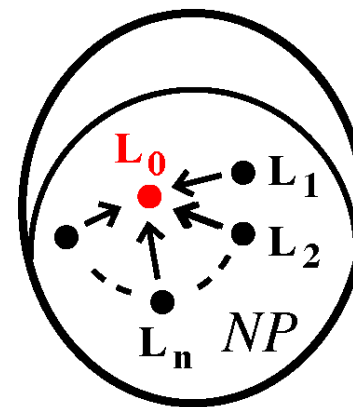
Skills to learn

- 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**.



SATISFIABILITY (SAT)

Example

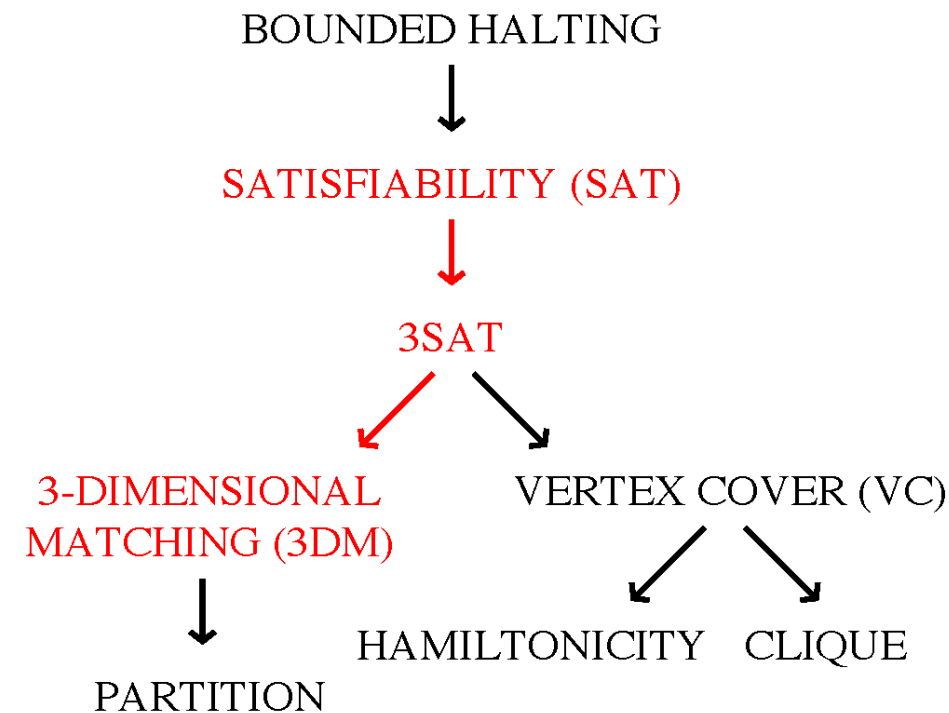
$$I = C \cup U$$

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

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

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

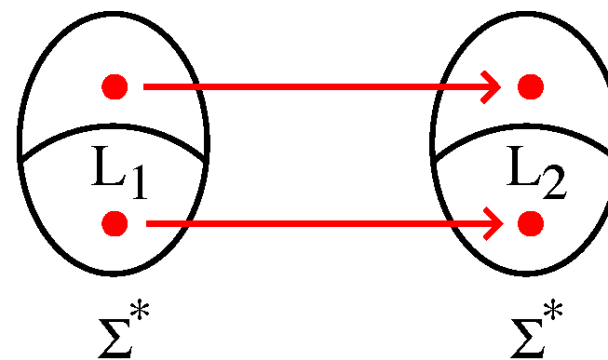
Further (basic) reductions



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

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' .

$$\begin{array}{ll} C_j & C_j' \\ (x_1 \vee x_2 \vee x_3) & \longmapsto (x_1 \vee x_2 \vee x_3) \\ \\ (x_1 \vee x_2) & \longmapsto (x_1 \vee x_2 \vee y_j), (x_1 \vee x_2 \vee \neg y_j) \\ \\ (x_1) & \longmapsto (x_1 \vee y_j^1 \vee y_j^2), (x_1 \vee \neg y_j^1 \vee y_j^2), \\ & (x_1 \vee y_j^1 \vee \neg y_j^2), (x_1 \vee \neg y_j^1 \vee \neg y_j^2) \\ \\ (x_1 \vee \dots \vee x_8) & \longmapsto (x_1 \vee x_2 \vee y_j^1), (\neg y_j^1 \vee x_3 \vee y_j^2), \\ & (\neg y_j^2 \vee x_4 \vee y_j^3), (\neg y_j^3 \vee x_5 \vee y_j^4), \\ & (\neg y_j^4 \vee x_6 \vee y_j^5), (\neg y_j^5 \vee x_7 \vee x_8) \end{array}$$

Question: Why is this a proper reduction?

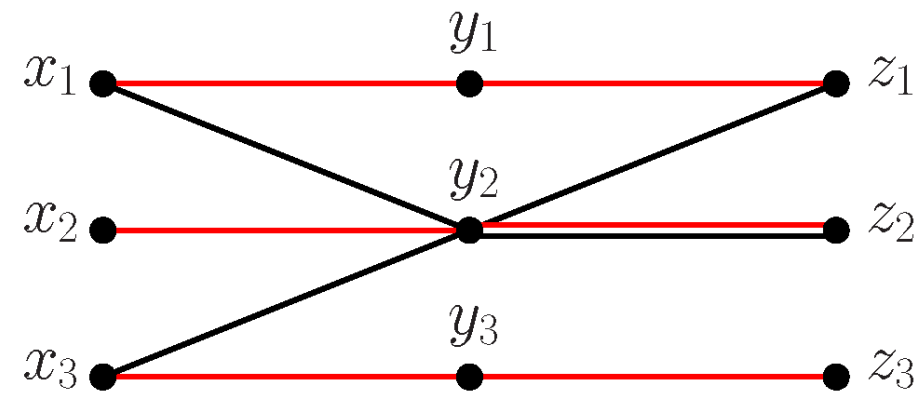


3-DIMENSIONAL MATCHING (3DM)

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 ($|A| = |B| = |C| = q$).

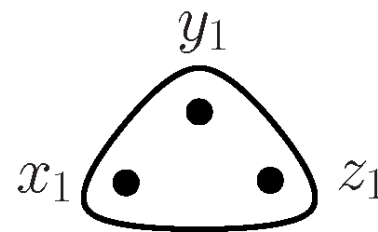
Question: Is there a **matching in M** , i.e. a subset $M' \subseteq M$ such that every element of A , B and C is part of exactly 1 triple in M' ?

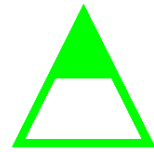
Example



$$M = \{ (x_1, y_1, z_1), (x_1, y_2, z_2), \\ (x_2, y_2, z_2), (x_3, y_3, z_3), (x_3, y_2, z_1) \}$$

We will use sets with 3 elements to visualize triples:





Reductions are like translations from one language to another. The same properties must be expressed.



3SAT \propto 3DM

3SAT

variables x_1, \dots, x_n

literals $x_1, \neg x_1$

clauses

$$C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$$

“There exists a sat.
truth assignment”

3DM

variables $x_3^j, a_3^j, b_j^2, c_k^1$

variables $x_1^j, \neg x_1^j$

triples (x_1^j, b_j^1, b_j^2)

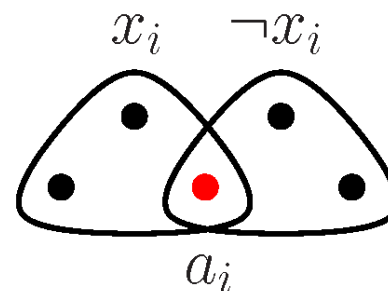
$$(\neg x_3^j, b_j^1, b_j^2)$$

“There is a
matching”

“There is a truth assignment T ”

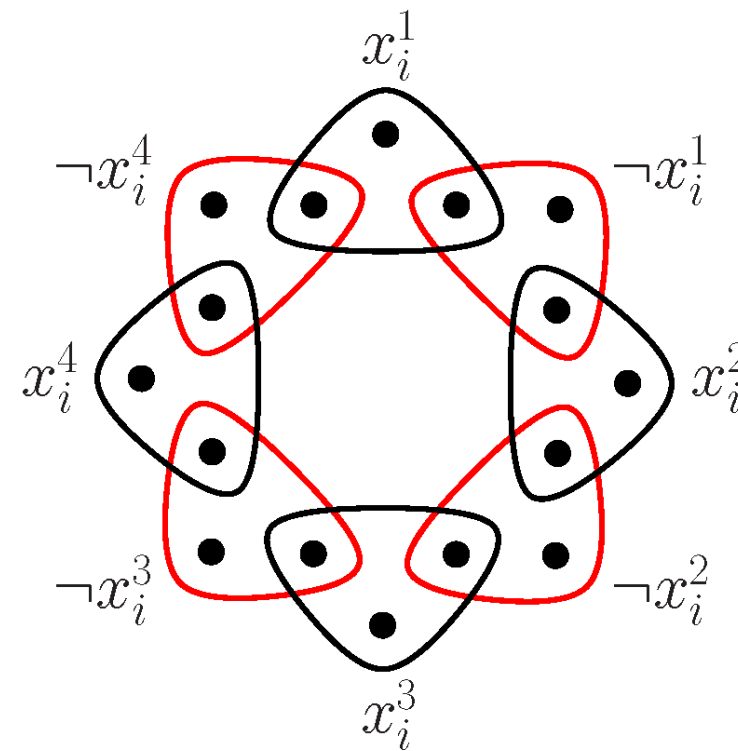
- $\exists T : \{x_1, \dots, x_n\} \rightarrow \{\text{TRUE}, \text{FALSE}\}$
- $T(x_i) = \text{TRUE} \Leftrightarrow T(\neg x_i) = \text{FALSE}$

The second property is easily translated to the 3DM-world:



$T(x_i) = \text{TRUE} \longrightarrow x_i \text{ is not “married”}$

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:

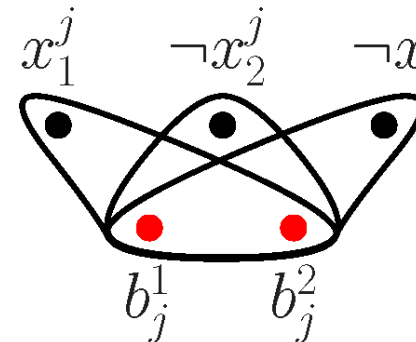


- Either all the black triples must be chosen (“married”) or all the red ones!
- If $T(x_i) = \text{TRUE}$ then we choose all the red triples, and the black copies of x_i are free to be used later in the reduction. And vice versa.
- We make one such **truth setting component** for each variable x_i in 3SAT.



“ T is satisfying”

We translate each clause (example:
 $C_j = (x_1 \vee \neg x_2 \vee \neg x_3)$) into 3 triples:

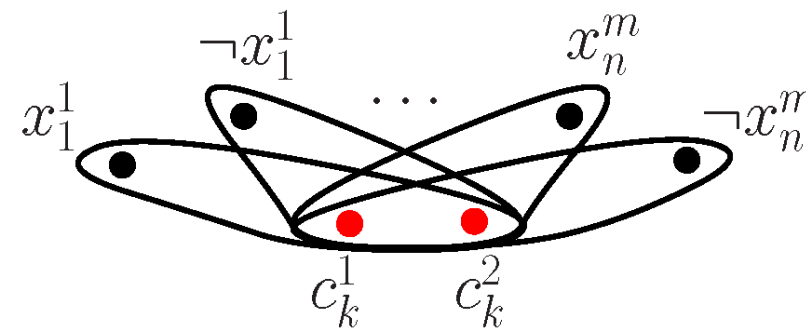


- b_j^1 and b_j^2 can be married if and only if at least one of the literals in C_j is not married in the truth setting component.
- If we have a satisfiable 3SAT-instance , then all b_j^1 and b_j^2 -variables ($1 \leq j \leq m$) can be married.
- If we have a negative 3SAT-instance , then some b_j^1 and b_j^2 -variables will not be married.



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 $1 \leq k \leq m(n - 1)$.



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

sets and

triples (subsets)

“There is

a matching M' ”

SUBSET SUM

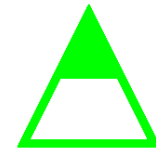
numbers

“There is

a subset with
total size B ”



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 $M' \longleftrightarrow$ There is a subset M'
 $\sum_{M'} \text{sizes} = B$

It is natural to set $B = \overbrace{111 \dots 11}^n$, 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 $01_b + 01_b + 01_b = 11_b$.



3DM-instance:

$$M \subseteq W \times X \times Y$$

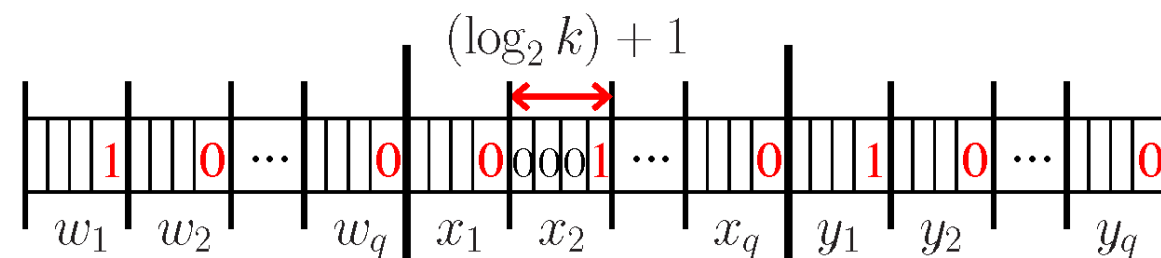
$$W = \{w_1, w_2, \dots, w_q\}$$

$$Y = \{y_1, y_2, \dots, y_q\}$$

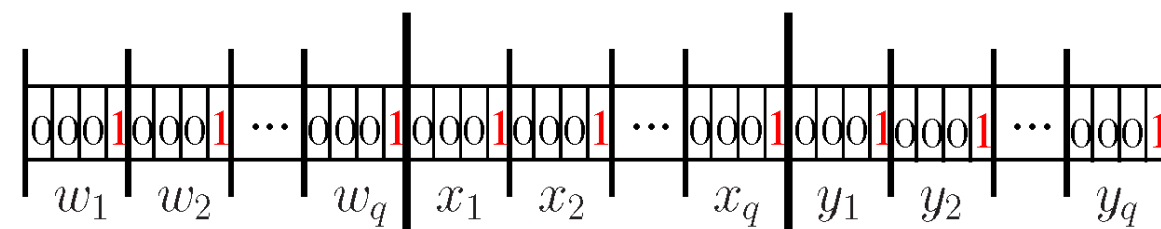
$$Z = \{z_1, z_2, \dots, z_q\}$$

$$M = \{m_1, m_2, \dots, m_k\}$$

- 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:





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.

$$\left[\begin{array}{c} S(b_1) \\ \\ \\ \\ \\ B \end{array} \right] \quad \left[\begin{array}{c} S(b_2) \\ \\ \\ \sum S(a) - B \end{array} \right]$$

- 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.

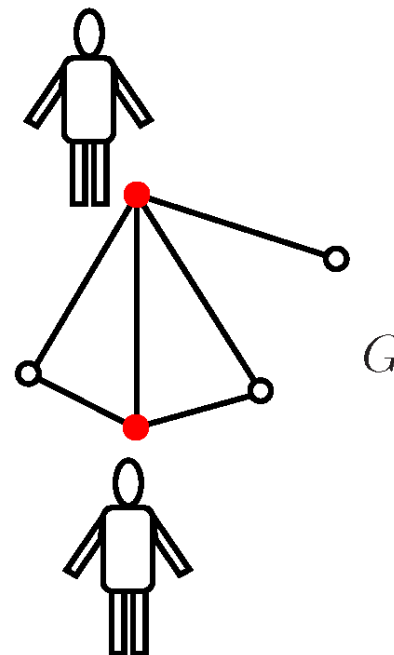


VERTEX COVER (VC)

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?”



3SAT \propto VC



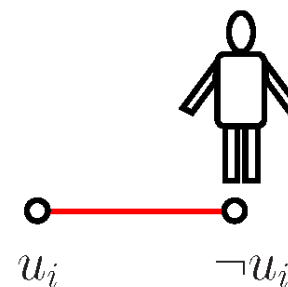
3SAT

literals
clauses
“There exists a sat.
truth assignment”

VERTEX COVER

vertices
subgraphs
“There is a VC
of size K ”

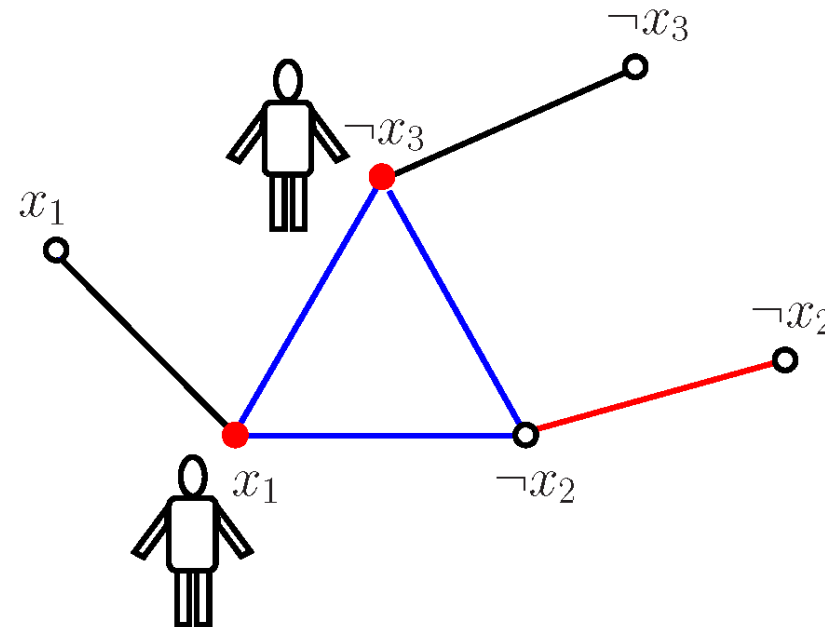
literals \mapsto vertices



- 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).

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)!

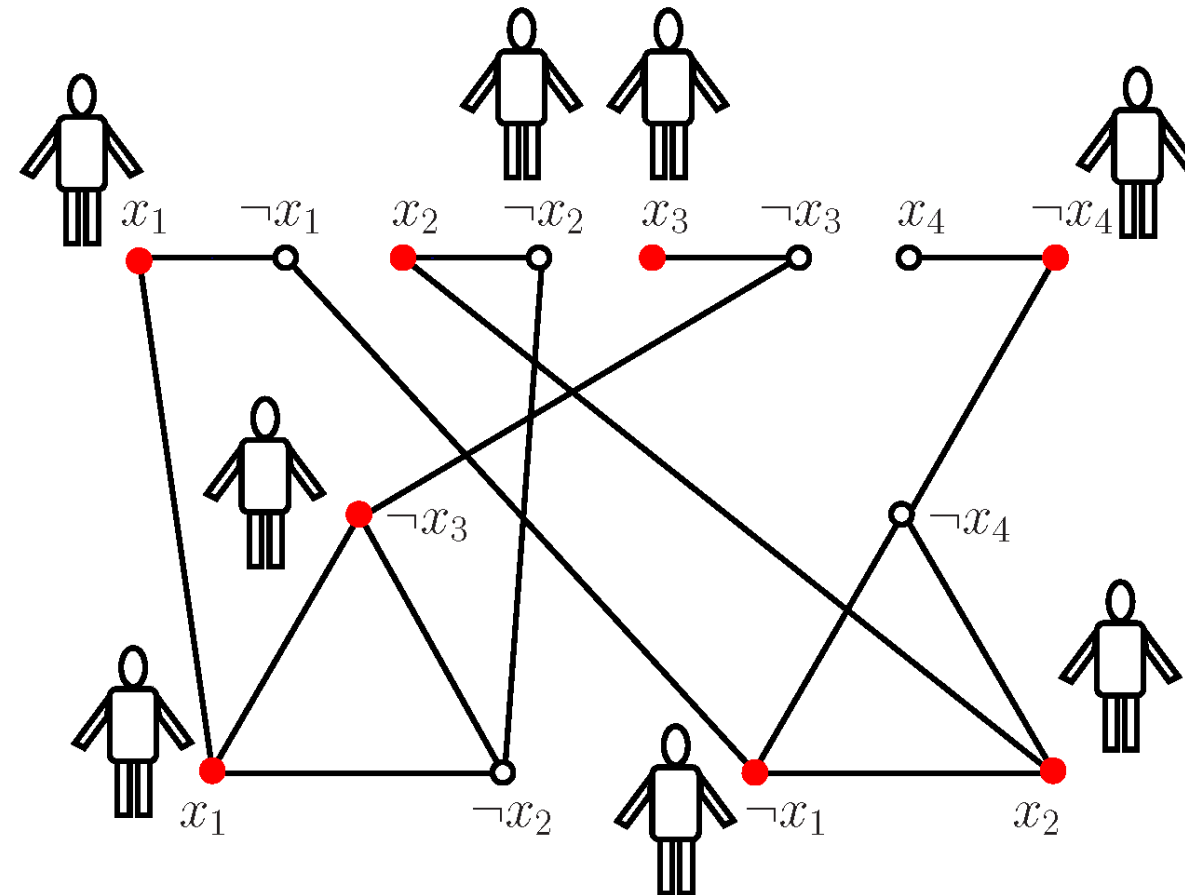


Example

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 ...



VERTEX COVER, CLIQUE AND INDEPENDENT SET

For $G = (V, E)$ and subset $V_1 \subset V$, the following statements are equivalent:

- (a) V_1 is a vertex cover of G
- (b) $V - V_1$ is an independent set in G
- (c) $V - V_1$ is a clique in G^c .

Corollary:

CLIQUE and INDEPENDENT SET are NP-complete.

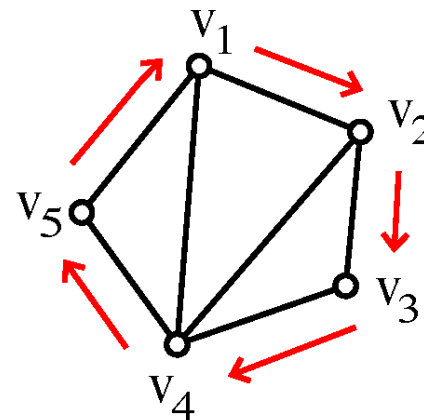


HAMILTONICITY

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)



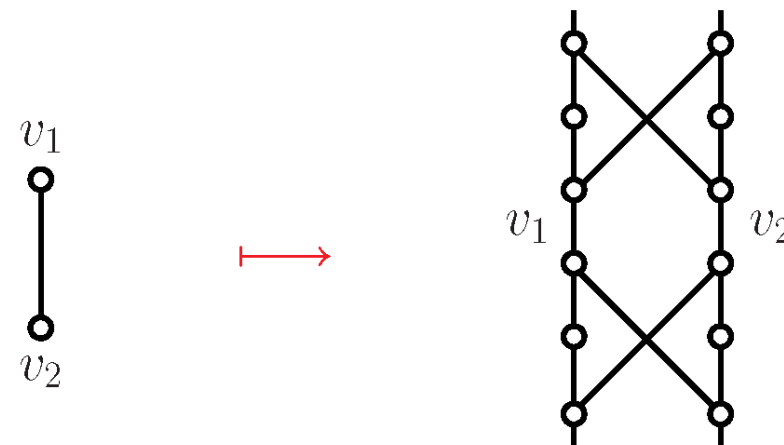


VC

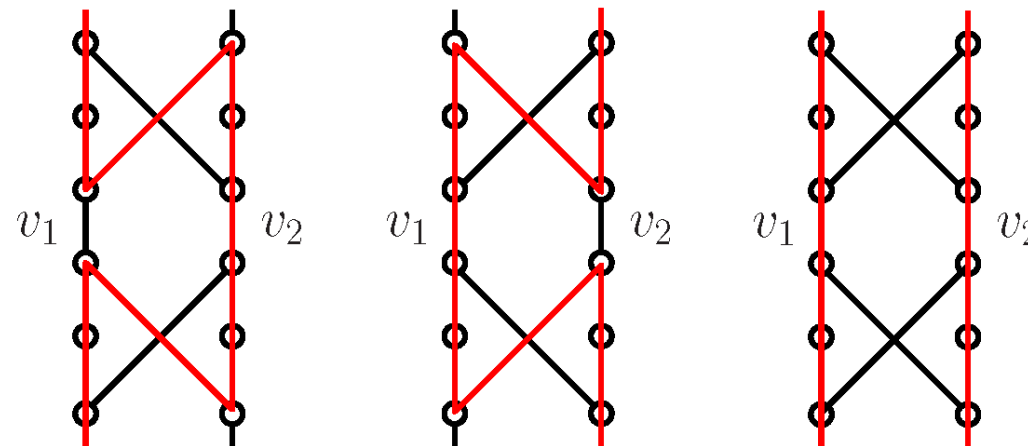
HAMILTONICITY

edges	\mapsto	edge gadgets
vertices	\mapsto	how gadgets are connected
K guards	\mapsto	K selector nodes

edges \mapsto edge gadgets



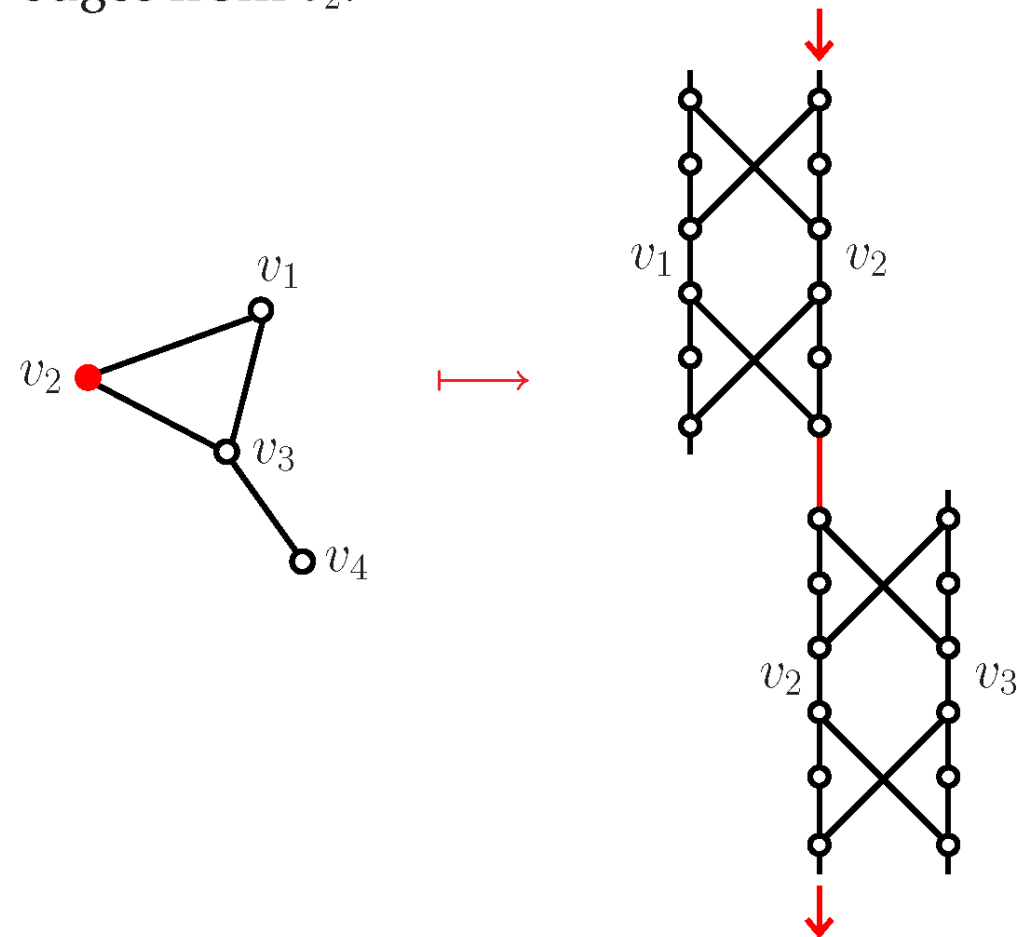
A Hamiltonian path can visit the vertices in the edge gadget in one of three ways:



We want this to correspond to guards being placed on v_1 or v_2 or both v_1 and v_2 , respectively.

vertices \mapsto **how gadgets are connected**

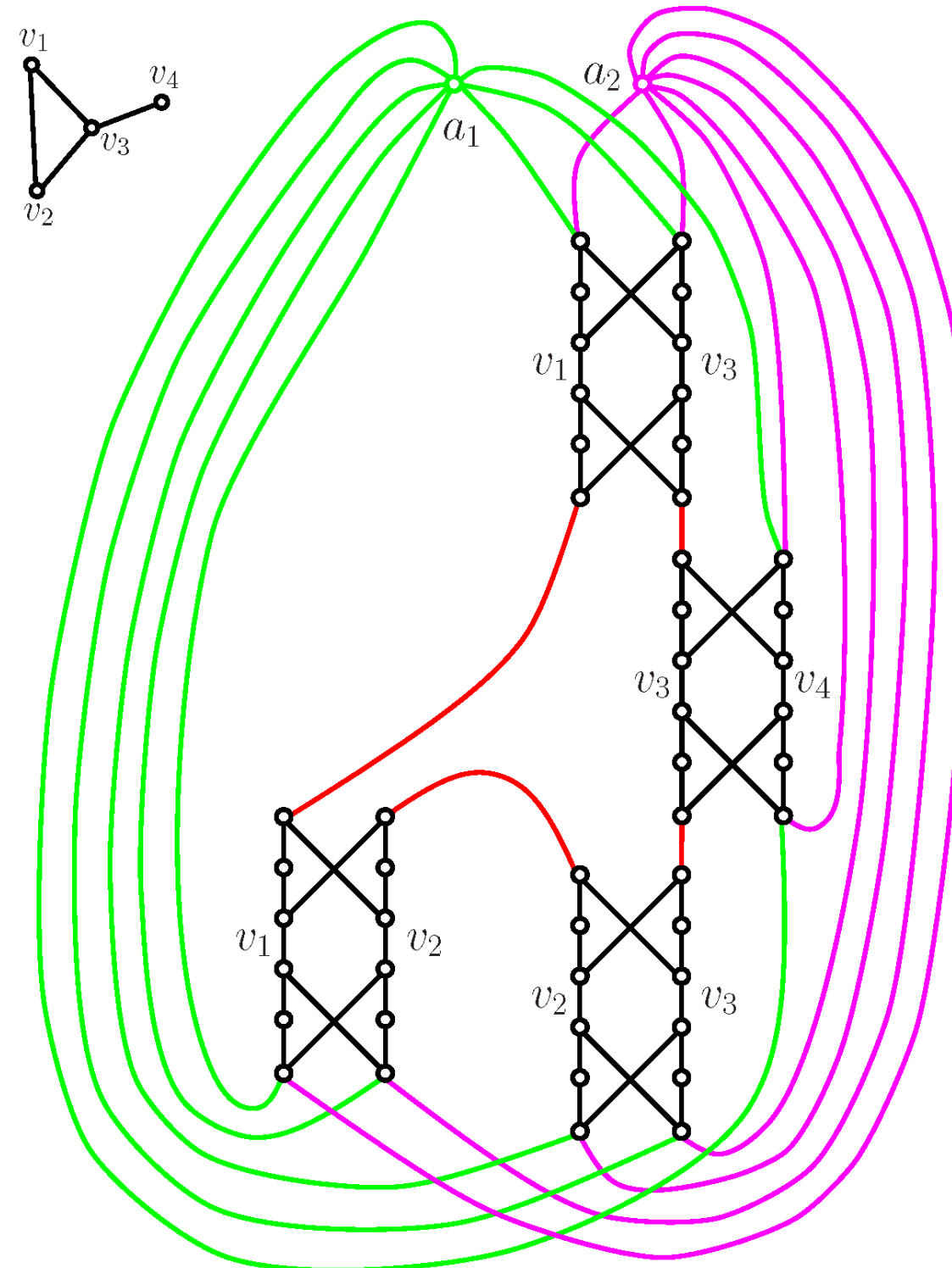
For each vertex v_2 , we connect together **in serial** all edge gadgets corresponding to edges from v_2 :



- 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 .

K guards $\mapsto K$ selector nodes

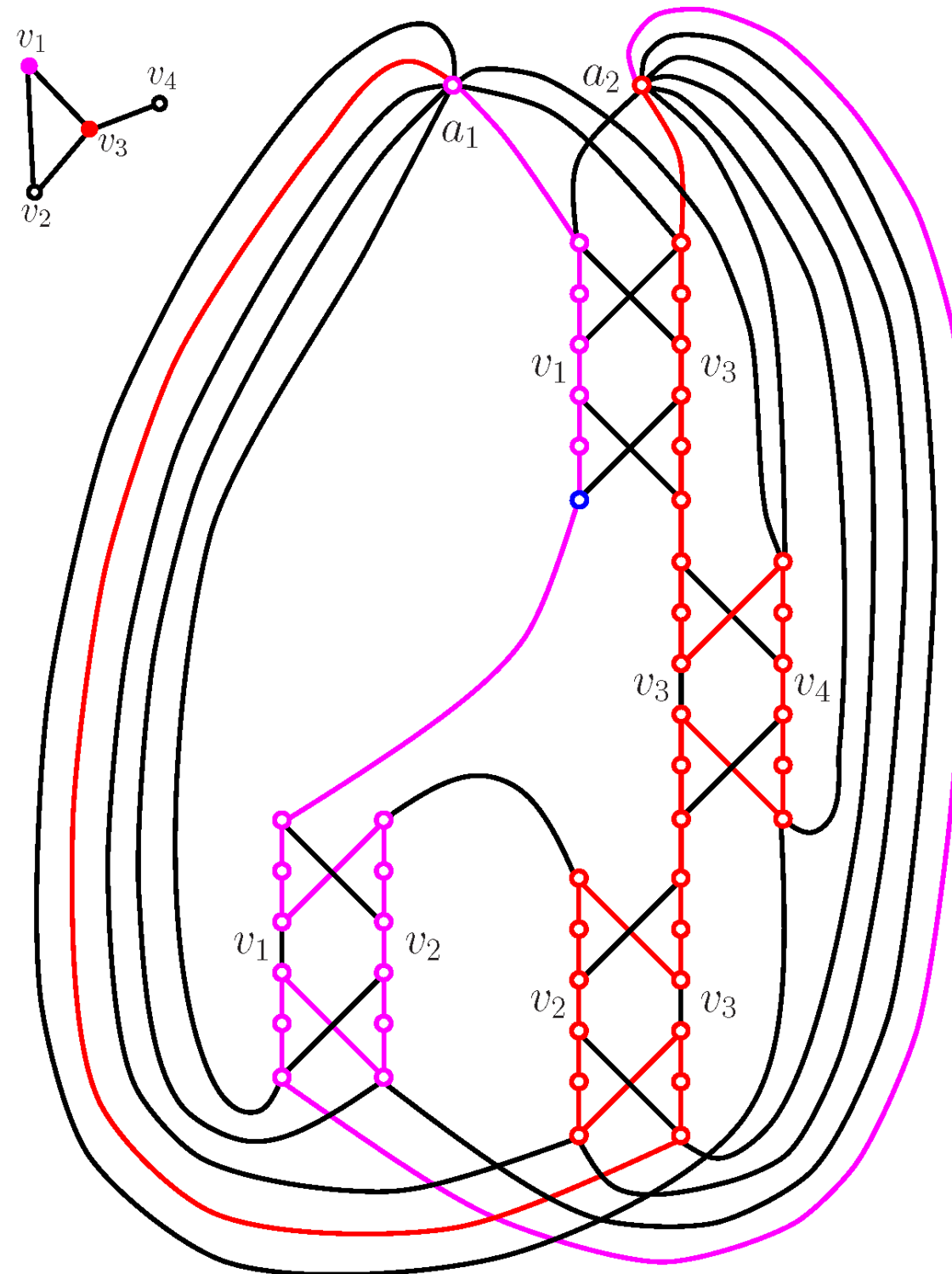
We finish the construction by introducing K selector nodes a_i which are connected with all “loose” edges:



There is a VC
which uses K guards



There is a
Hamiltonian cycle



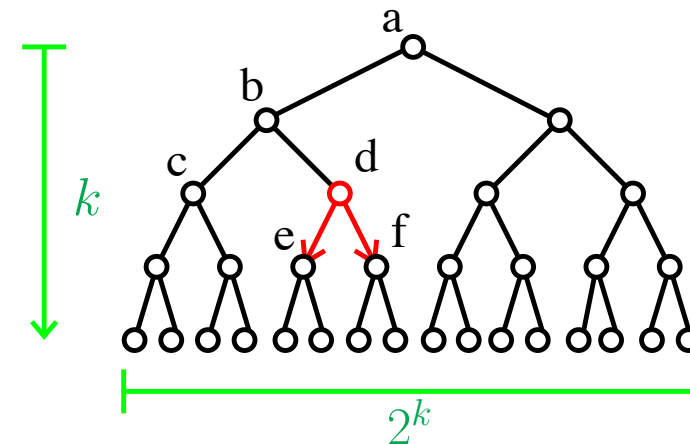
Coping with NP-completeness 1

IN3130 Algorithms and Complexity

Coping with Intractability

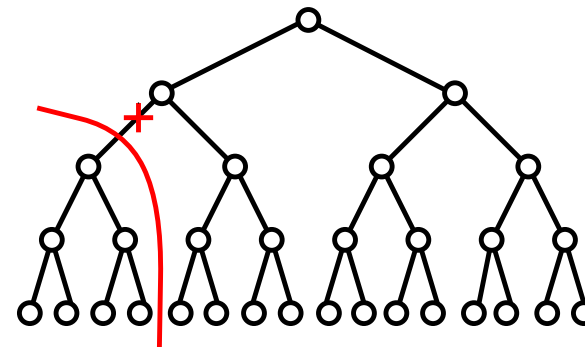
Branch-and-Bound

Branch:



Leaf nodes = possible solutions

Bound:

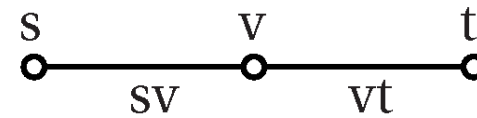


- Backtracking
- Pruning ('avskjæring')



Dynamic Programming

- Building up a solution from solutions from subproblems
- Principle: Every part of an optimal solution must be optimal.





Restricting

- **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
 - ...



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 .



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.



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:

$M_0 : \{0\}$
 $M_1 : \{0, 11\} \quad (0 + 11 = 11)$
 $M_2 : \{0, 11, 18, 29\} \quad (0 + 18 = 18, 11 + 18 = 29)$
 $M_3 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
 $M_4 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
 $M_5 : \{0, 11, 15, 18, 24, 26, 29, 33, 35, 39, 42, 44, 50, 53\}$
 $M_6 : \{0, 7, 11, 15, 18, 22, 24, 25, 26, 29, 31, 33, 35, 36, 39, 40, 42, 44, 46, 49, 50, 51, 53\}$

Theorem 1 *DP is a pseudo-polynomial algorithm. The running time of DP is $\mathcal{O}(nK \log K)$.*

Proof: MAXINT(I) = $K \dots$

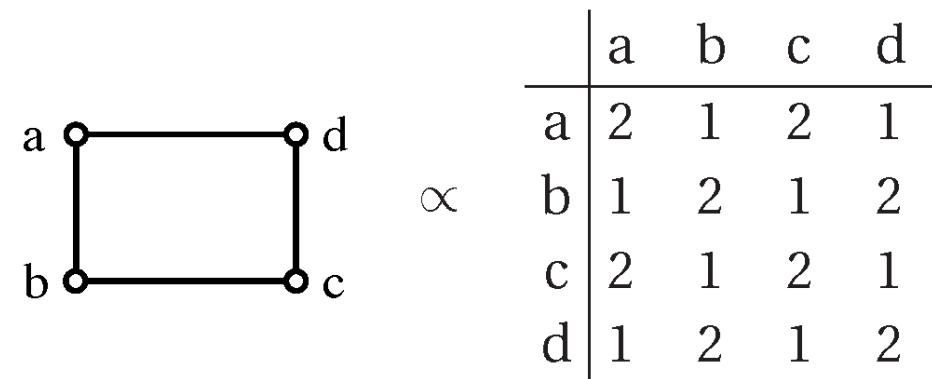


Strong \mathcal{NP} -completeness

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).



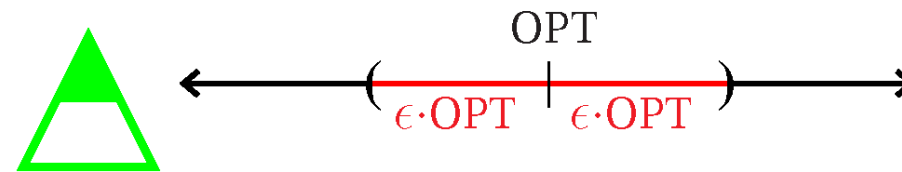
$$K = n(= 4)$$



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

Approximation

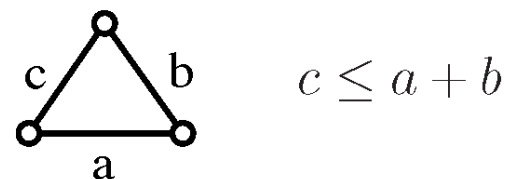


Def. 3 Let L be an optimization problem. We say that algorithm M is a **polynomial-time ϵ -approximation algorithm** for L if M runs in polynomial time and there is a constant $\epsilon \geq 0$ such that M is guaranteed to produce, for all instances of L , a solution whose cost is within an ϵ -neighborhood from the optimum.

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

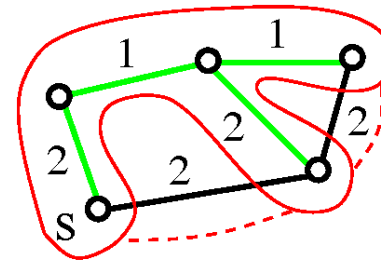




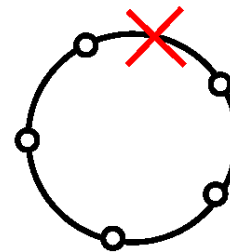
Algorithm TSP- \triangle :

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 $\epsilon = 1$.



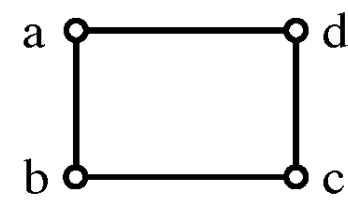
Opt. tour



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.



	a	b	c	d
a	$2+\epsilon n$	1	$2+\epsilon n$	1
b	1	$2+\epsilon n$	1	$2+\epsilon n$
c	$2+\epsilon n$	1	$2+\epsilon n$	1
d	1	$2+\epsilon n$	1	$2+\epsilon n$

$$K = n(= 4)$$

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

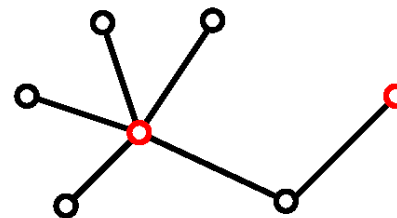
- **Heuristics** are a common way of dealing with intractable (optimization) problems in practice.
- Heuristics differ from algorithms in that they have no performance guarantees, i.e. they don't always find the (best) solution.

A greedy heuristic for VERTEX COVER-opt.:

Heuristic VC-H1:

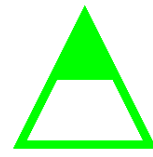
Repeat until all edges are covered:

1. Cover highest-degree vertex v ;
2. Remove v (with edges) from graph;



Theorem 4 *The heuristic VC-H1 is not an ϵ -approximation algorithm for VERTEX COVER-opt. for any fixed ϵ .*

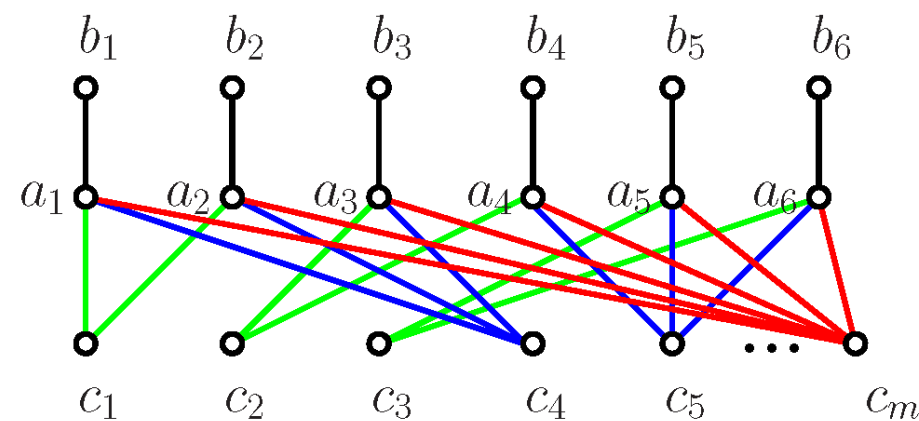
Proof:



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:

$$\begin{aligned}\frac{|VC-H1| - |OPT|}{|OPT|} &= \frac{(n \log n + n) - n}{n} \\ &= \frac{n \log n}{n} = \log n \neq \epsilon\end{aligned}$$

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

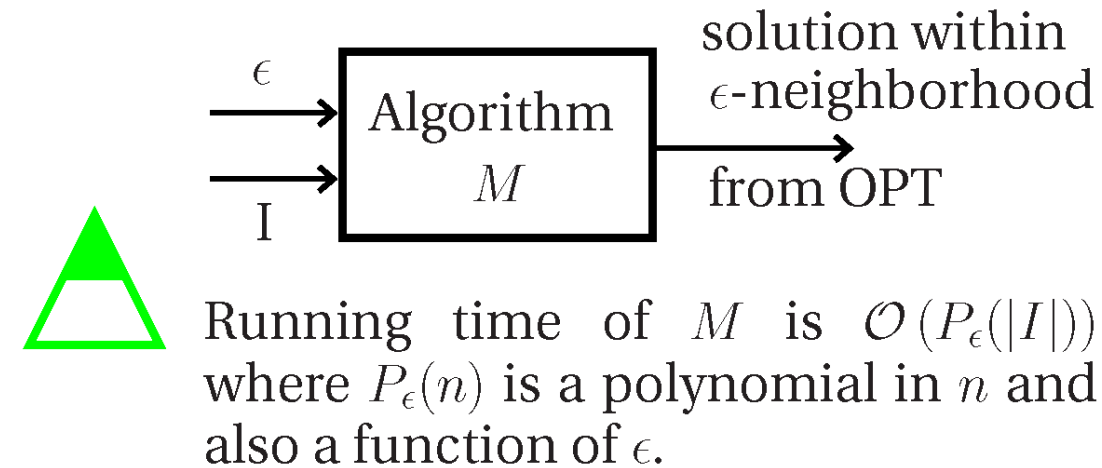
Heuristic VC-H2:

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.

Polynomial-time approximation schemes (PTAS)



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.

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

$$\sum_{j=1}^n c_j x_j$$

subject to

$$\sum_{j=1}^n w_j x_j \leq K \text{ and } x_j = 0, 1$$

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 .



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.



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 $\sum_{j \in S} c_j = 34$.



The FTPAS 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 $\sum_{j \in S'} c_j = 740$.



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**”:

$$C_m = 53501 \overbrace{87959}^{\text{half of length}} \\ \text{but only } 10^{-5} \text{ of precision}$$



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 the have an upper bound on the **error**:

$$\sum_{j \in S} c_j - \sum_{j \in S'} c_j \leq n \cdot 10^t$$



- 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 $\mathcal{O}\left(\frac{n^3 \log(n C_m W_m)}{\epsilon}\right)$.

Coping with NP-completeness 2

IN3130 Algorithms and Complexity

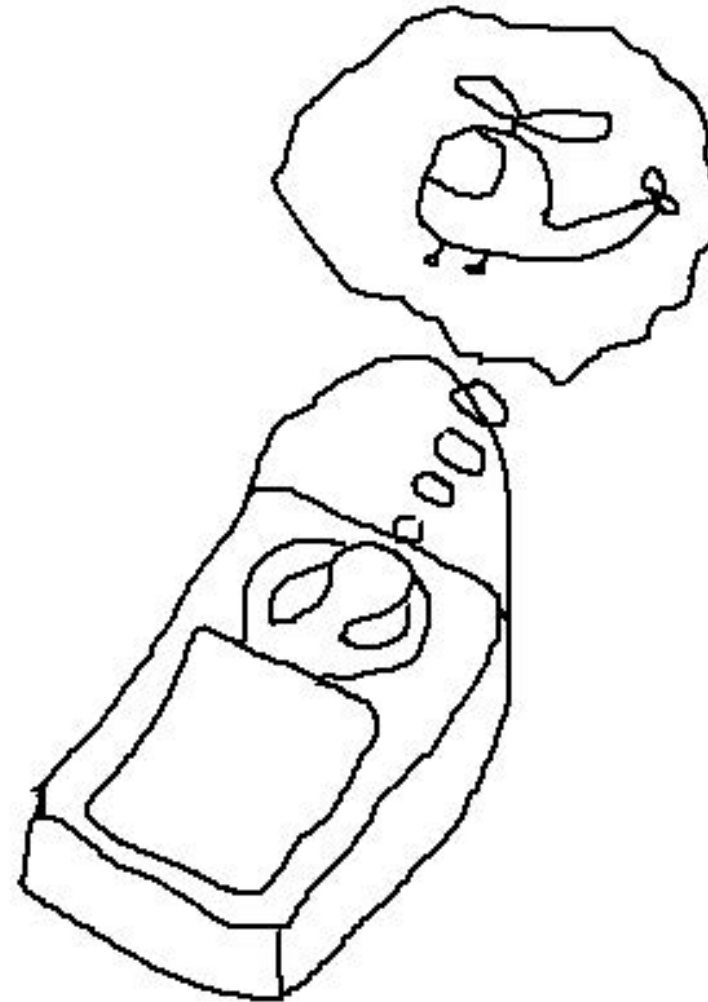


Alternative approaches to algorithm design and analysis

- **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



Average-case analysis & algorithms



~~Worst case~~



- **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).



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



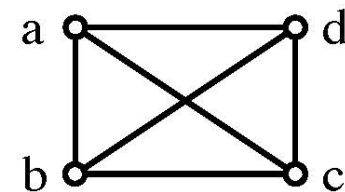
Example: 3-COLORABILITY

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.



- 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):

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



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 $p = 5/n$.



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 ($\lim_{n \rightarrow \infty}$).*

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
$$\begin{aligned} &\leq \mathcal{O}(n) + \mathcal{O}(n^2) P_1(n) + \mathcal{O}(2^{2n}) P_1(n) P_2(n) \\ &= \mathcal{O}(n) \text{ if } P_1(n) \cdot \mathcal{O}(n^2) = \mathcal{O}(n) \\ &\quad \text{and } P_1(n) P_2(n) \cdot \mathcal{O}(2^{2n}) = \mathcal{O}(n) \end{aligned}$$
- 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.



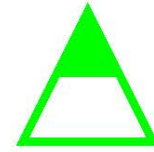
Randomized computing

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

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



Randomized algorithms

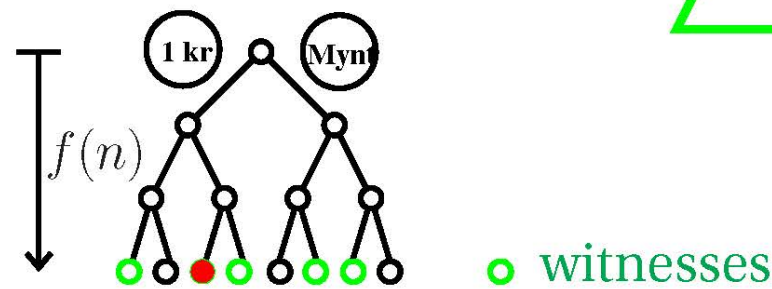


Idea: Toss a coin & simulate non-determinism

Example 1: Proving polynomial non-identities

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

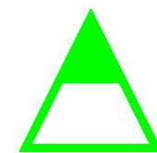
- What is the “classical” complexity of the problem?
- Fast, randomized algorithm:
 - Guess values for x and y and compute left-hand side (LHS) and right-hand side (RHS) of equation.
 - If $\text{LHS} \neq \text{RHS}$, then we know that the polynomials are different.
 - If $\text{LHS} = \text{RHS}$, 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.



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!



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'

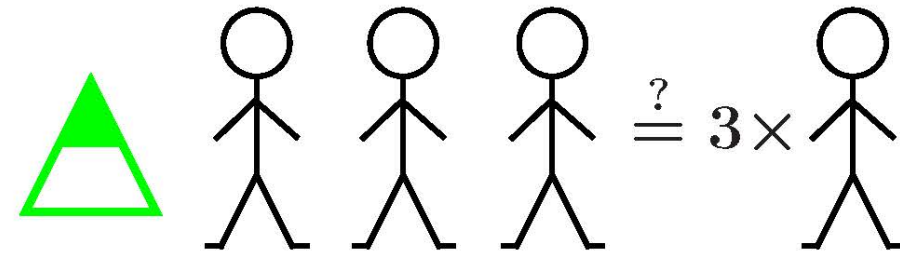
Neural Networks

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

Taboo search

- Analogy with culture
- adaptive memory, responsive exploration

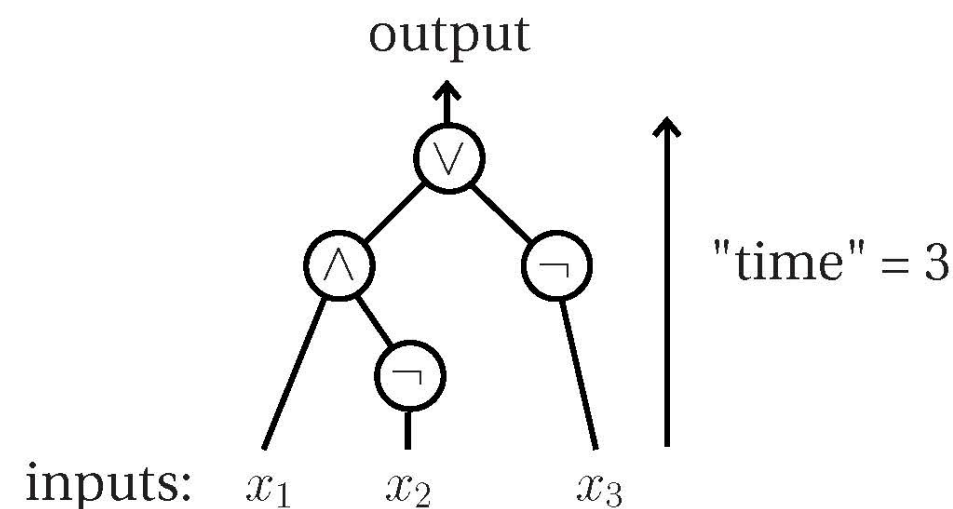
Parallel computing



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

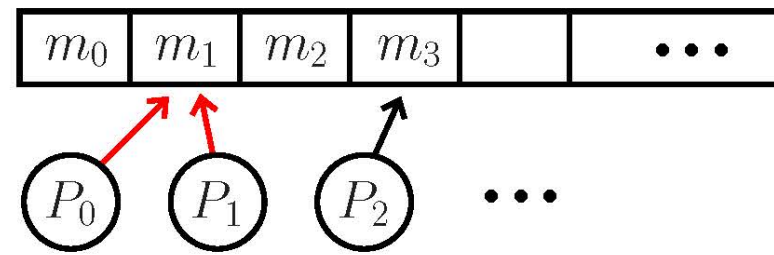
Parallel machine models

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



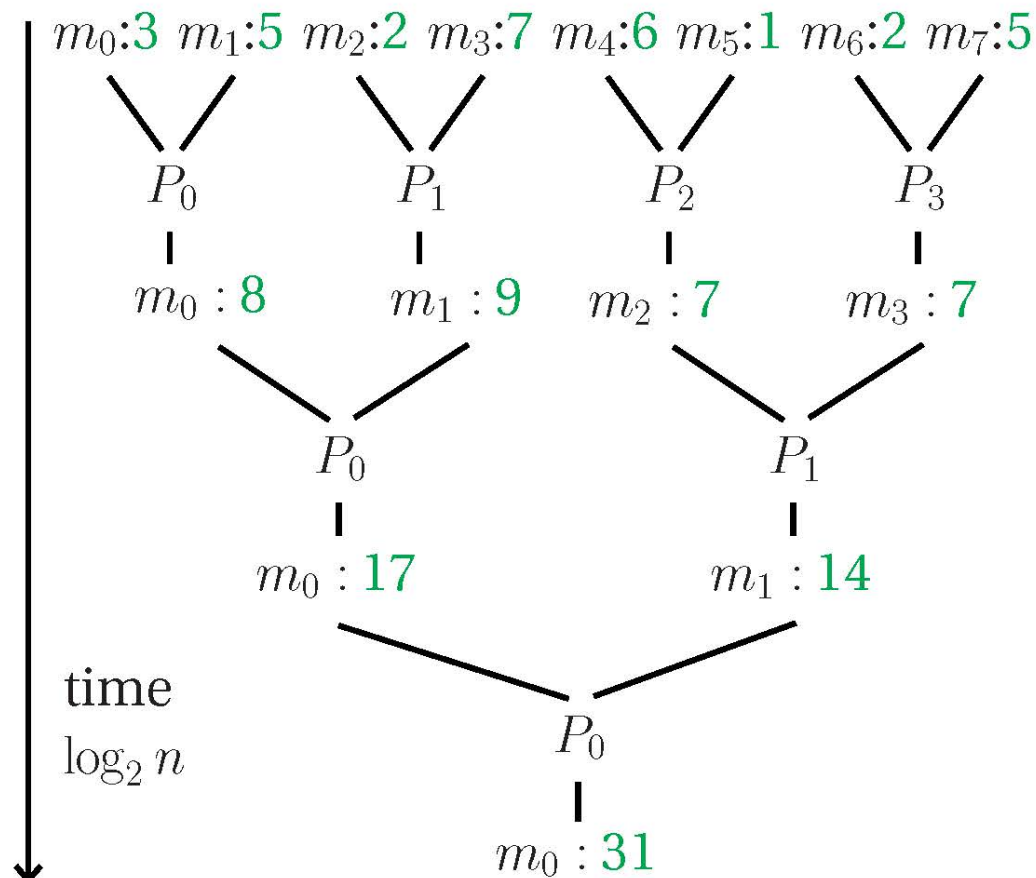
— Boolean Circuit complexity: **"time"** (length of longest directed path) and **hardware** (# of gates)

• Parallel 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 2 *With polynomial many processors*

parallel poly. time = sequential poly. time

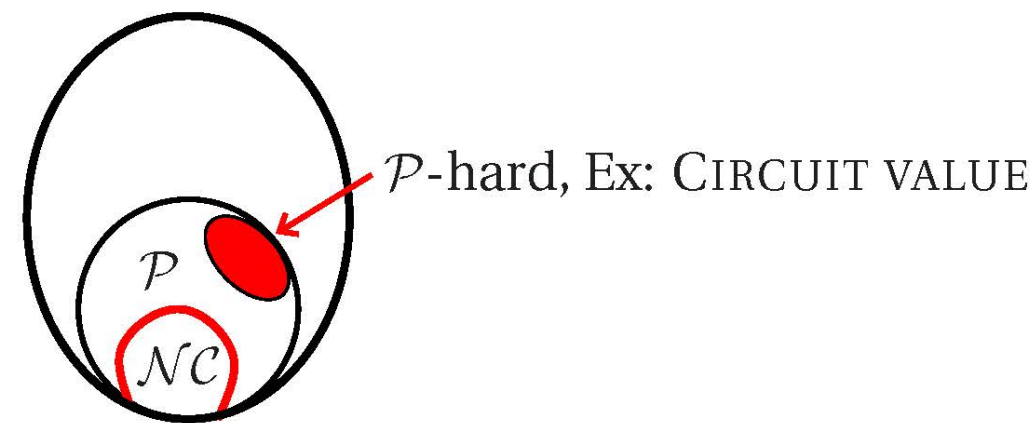
Proof:

- 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 $t_1(m) \cdot t_2(n) = \text{polynomial}$.



Parallel complexity classes

Def. 2 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.



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

