NP-completeness

a glimpse into Structural Complexity

Polynomial-time algorithms

- All algorithms we have seen so far (and more) run in $O(n^c)$ time
  - graph traversals, Dijkstra, Bellman-Ford, all-pairs shortest paths, minimum spanning trees, minimum flow
  - interval scheduling, LCS, edit distance, sequence alignment
  - Viterbi algorithm in HMM
  - …

Polynomial-time algorithms

- All algorithms we have seen so far (and more) run in $O(n^c)$ time
  - graph traversals, Dijkstra, Bellman-Ford, all-pairs shortest paths, minimum spanning trees, minimum flow
  - interval scheduling, LCS, edit distance, sequence alignment
  - Viterbi algorithm in HMM
  - …
- … but also
  - solving systems of linear equations over reals (Gauss elimination)
  - linear programming over reals (ellipsoid method by L.Khachiyan, 70s)
  - primality testing (Agrawal-Kayal-Saxena, 2002)
  - …

"Efficiency assumption"

- Practical algorithms = polynomial-time algorithms
- Tractable problems = those which admit polynomial-time algorithms *(Cobham-Edmonds thesis)*

- *Corollary*: untractable problems = those for which there is no polynomial-time algorithm

"Efficiency assumption"

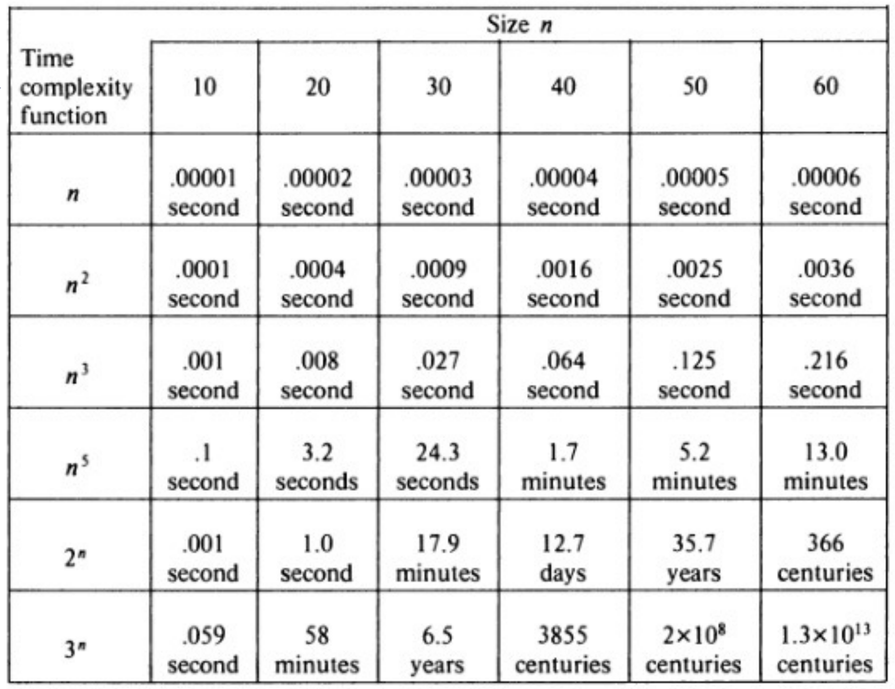
- Practical algorithms = polynomial-time algorithms
- Tractable problems = those which admit polynomial-time algorithms *(Cobham-Edmonds thesis)*

- *Corollary*: untractable problems = those for which there is no polynomial-time algorithm

- There are many problems for which we not to know polynomial-time algorithms, but for many of them, we cannot *prove* that there is none
- *Provably* untractable time problems exist, e.g. (generalized) chess, checkers or GO ($n \times n$ board)
- … up to *undecidable problems* (e.g. Halting problem, Post correspondence problem, …)

Polynomial vs exponential

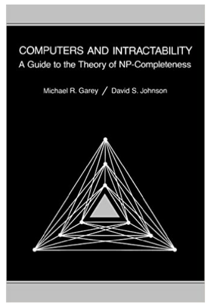
in milliseconds



Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions.

Taken from :
Garey&Johnson, Computers and Intractability, 1979



COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

М.Гэри, Д.Джонсон
ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ И ТРУДНОРЕШАЕМЫЕ ЗАДАЧИ
Издательство «Мир»

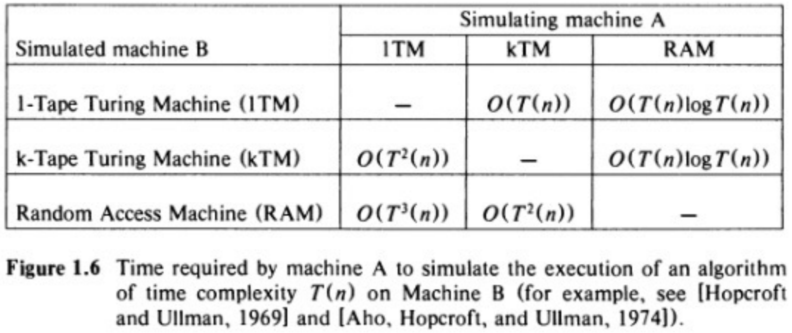
Measuring the complexity of algorithms

- RAM = Random Access Machine (*unit-cost* or *log-cost*)
- TM = Turing machine

Measuring the complexity of algorithms

- RAM = Random Access Machine (*unit-cost* or *log-cost*)
- TM = Turing machine

- RAM and TM are "polynomially equivalent"



Simulated machine B	Simulating machine A		
	1TM	kTM	RAM
1-Tape Turing Machine (1TM)	–	$O(T(n))$	$O(T(n)\log T(n))$
k-Tape Turing Machine (kTM)	$O(T^2(n))$	–	$O(T(n)\log T(n))$
Random Access Machine (RAM)	$O(T^3(n))$	$O(T^2(n))$	–

Figure 1.6 Time required by machine A to simulate the execution of an algorithm of time complexity $T(n)$ on Machine B (for example, see [Hopcroft and Ullman, 1969] and [Aho, Hopcroft, and Ullman, 1974]).

from :
Garey&Johnson, Computers and Intractability, 1979

- … provided (for unit-cost RAM) that all operands of arithmetic operations are polynomially bounded (i.e. fit a const number of computer words)

Measuring the complexity: input encoding

- Time complexity is a ***function*** of input data size
- For the Turing machine:
  - size of input = number of tape cells of the input encoding
  - time = number of steps (transitions)
- Inside the "polynomial world" the way of encoding the input is important
- Here we assume that all encodings are "polynomially equivalent" ⇒ numbers are ***not*** encoded in unary and the encoding of data is "compact »
  - *Ex 1*: $O(kN)$-time algorithm for coin changing is ***not*** polynomial
  - *Ex 2*: primality testing can be done in $O((\log N)^c)$

Decision vs optimization problems

- Computational problems can be of different nature
- Optimization problem: Compute an object of optimum (minimum or maximum) "size" (e.g. minimum spanning tree, shortest path, etc.)
- Decision problem: Does there exist an object of "size" $\leq k$?

- Decision problems are formalized as language membership problem

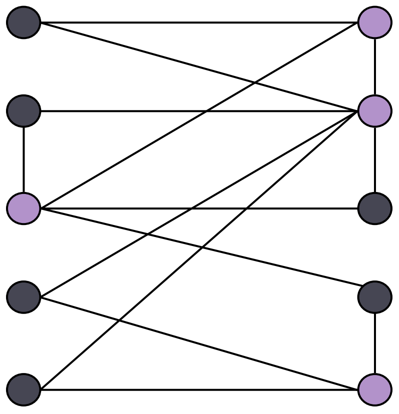
- Obviously, solving an optimization problem implies solving the corresponding decision problem
- Very often, the inverse is true: an opimization problem is "polynomially reducible" (*cf below*) to the decision problem
- Decision and optimization problems are usually "polynomially equivalent". We focus on decision problems.

Example: minimum vertex cover

- Definition: Given a graph $G = (V, E)$, a vertex cover of G is a subset of vertices S ⍰ V such that for each edge, at least one of its endpoints is in S?

- MINIMUM VERTEX COVER (decision problem): Given a graph $G = (V, E)$ and an integer k, is there a vertex cover S such that $|S| \leq k$?
- MINIMUM VERTEX COVER (optimization problem): Given a graph $G = (V, E)$, find a vertex cover S of minimum cardinality.

- To find min vertex cover:
  - (Binary) search for cardinality k^* of min vertex cover
  - Find a vertex v such that $G - \{v\}$ has a vertex cover of size $\leq k^* - 1$ (any vertex in any min vertex cover will have this property)
  - Include v in the vertex cover
  - Recursively find a min vertex cover in $G - \{v\}$
  - $T_{opt}(n) = \log n \cdot T_{dec}(n) + n^2 \cdot T_{dec}(n)$



● vertex cover

Classes **P** and **NP**

- Class **P**:
  - *formal definition*: class of decision problems (languages) that can be solved on Turing machine in time $\leq p(n)$ for a fixed polynome p (n size of the problem)
  - *informal*: Class of problems that have polynomial time algorithms solving them
- Class **NP** ("Non-deterministic P"):
  - *formal definition*: class of decision problems that can be solved on a **non-deterministic** Turing machine in time $\leq p(n)$ for a fixed polynome p (n size of the problem)
  - *equivalent definition*: class of decision problems for which a solution can be **verified** in polynomial time (insures brute-force ('перебор') solutions)

A Survey of Russian Approaches to *Perebor* (Brute-Force Search) Algorithms

B. A. TRAKHTENBROT

Concerns about computational problems requiring brute-force or exhaustive search methods have gained particular attention in recent years because of the widespread research on the "P = NP?" question. The Russian word for "brute-force search" is "perebor." It has been an active research area in the Soviet Union for several decades. Disputes about approaches to perebor had a certain influence on the development, and developers, of complexity theory in the Soviet Union. This paper is a personal account of some events, ideas, and academic controversies that surrounded this topic and to which the author was a witness and—to some extent—a participant. It covers a period that started in the 1950s and culminated with the discovery and investigation of nondeterministic polynomial (NP)-complete problems independently by S. Cook and R. Karp in the United States and L. Levin in the Soviet Union.

*Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]—graph and tree search strategies; K.2 [**History of Computing**]—people, software*
General Terms: Algorithms, Theory, Verification
Additional Key Words and Phrases: brute-force search algorithms, perebor

Introduction

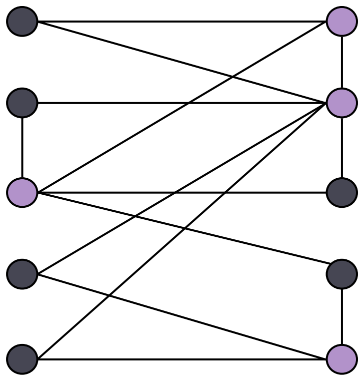
A *perebor* algorithm, or *perebor* for short, is Russian for what is called in English a "brute-force" or "exhaustive" search method. Other combinations of words also occur in translations from Russian, such as "successive trials," "sequential searching," and "thorough searching." To keep the historical flavor, I

case of an affirmative answer to (1), an n-tuple should be produced.

The obvious *perebor* algorithm that solves both the existential and constructive versions of the problem considers all the n-tuples of truth values in some order (say, lexicographical order). The first time an n-tuple

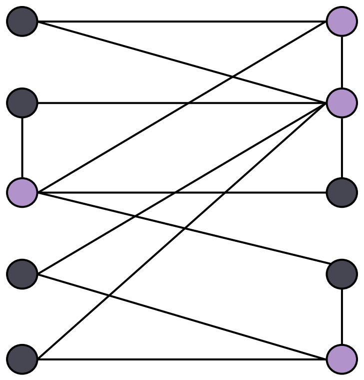
Examples of problems in NP

- Minimum vertex cover
  - easy to verify if a given subset S $\subseteq$V is a vertex cover



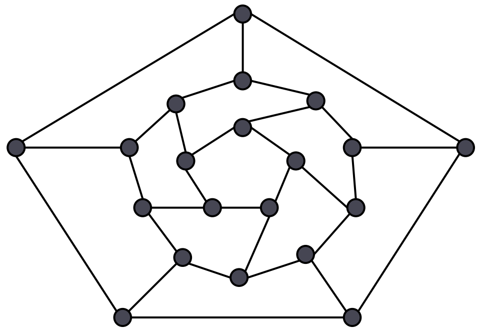
Examples of problems in NP

- Minimum vertex cover
  - easy to verify if a given subset $S \subseteq V$ is a vertex cover



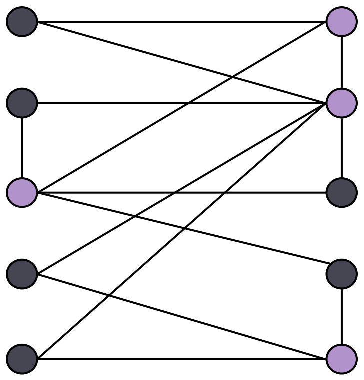
- More examples:
  - Hamiltonian circuit in a graph
  - largest clique in a graph
  - integer factorization $437669 = 541 \cdot 809$
  - longest common subsequence of multiple strings
  - multiset partition $\{3,1,1,2,2,1\} \to \{2,1,1,1\} \cup \{3,2\}$

    (NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
  - …



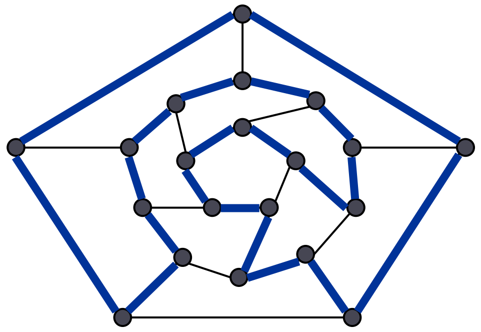
Examples of problems in NP

- Minimum vertex cover
  - easy to verify if a given subset $S \subseteq V$ is a vertex cover



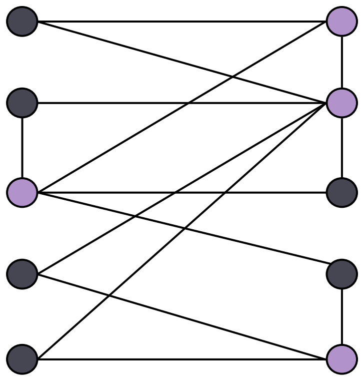
- More examples:
  - Hamiltonian circuit in a graph
  - largest clique in a graph
  - integer factorization $437669 = 541 \cdot 809$
  - longest common subsequence of multiple strings
  - multiset partition $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$

    (NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
  - …



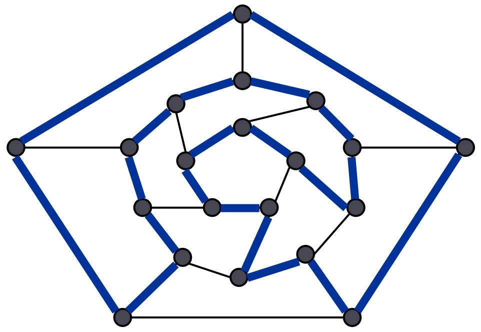
Examples of problems in NP

- Minimum vertex cover
  - easy to verify if a given subset $S \subseteq V$ is a vertex cover



- More examples:
  - Hamiltonian circuit in a graph
  - largest clique in a graph
  - integer factorization $437669=541 \cdot 809$
  - longest common subsequence of multiple strings
  - multiset partition $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$

    (NB: numbers are encoded in binary! otherwise a pseudo-polynomial dynamic programming algorithm exists!)
  - …



- **P⊆NP**

$1,000,000 question

P=NP?

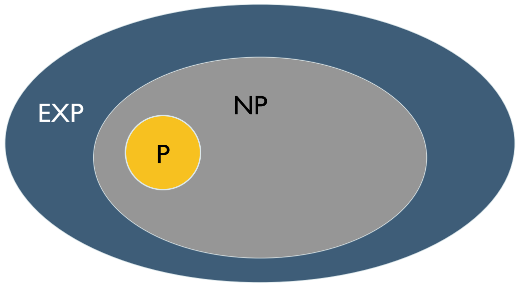
(or can '*перебор*' be eliminated?)

one of Millennium Prize Problems, Clay Mathematics Institute

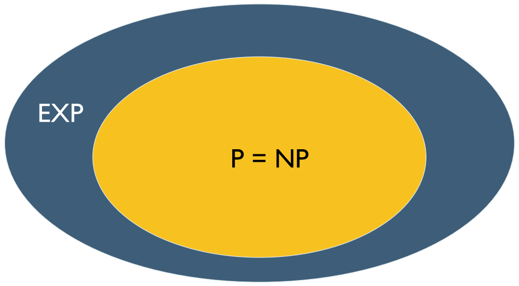
Why is it so important?

would break RSA cryptography
(and potentially collapse economy)

- If yes: Efficient algorithms for 3-COLOR, TSP, FACTOR, SAT, …
- If no: No efficient algorithms possible for 3-COLOR, TSP, SAT, …



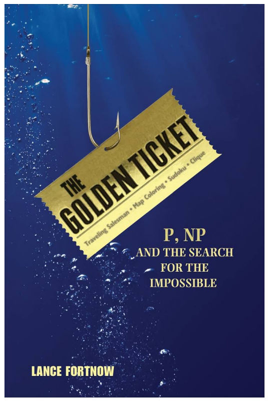
If P ≠ NP



If P = NP

Consensus opinion on P = NP? Probably no.

about consequences of P=NP read L.Fortnow, Golden ticket



Polynomial-Time Reduction

Idea: formalize "*problem X is at least as hard as problem* Y"

Suppose we could solve a problem X in polynomial-time. What else could we solve in polynomial time?

*Reduction**: Problem Y is polynomial-time reducible to problem X if arbitrary instances of problem Y can be solved using:

- Polynomial number of standard computational steps, and
- Polynomial number of calls to oracle that solves problem X

computational model supplemented by special piece of hardware that solves instances of Y in a single step

Notation: $Y \leq_P X$

That is, if we have a code for X, we can obtain a code for Y with only polynomial overhead

(*) called "*polynomial-time Turing reduction*", or "*Cook reduction*" (as opposed to "*Karp(-Levin) reduction*" or "*many-to-one reduction*" or "*polynomial transformation*" which is more restricted)

Polynomial-Time Reduction

Goal: Classify problems according to relative difficulty.

Design algorithms: If $Y \leq_P X$ and X can be solved in polynomial-time, then Y can also be solved in polynomial time. That is, if X is tractable, so is Y.

↑ we have used this earlier in our course

Establish intractability: If $Y \leq_P X$ and Y cannot be solved in polynomial-time, then X cannot be solved in polynomial time. That is, if Y is hard, so is X.

Establish (polynomial-time) equivalence: If $X \leq_P Y$ and $Y \leq_P X$, we write $X \equiv_P Y$

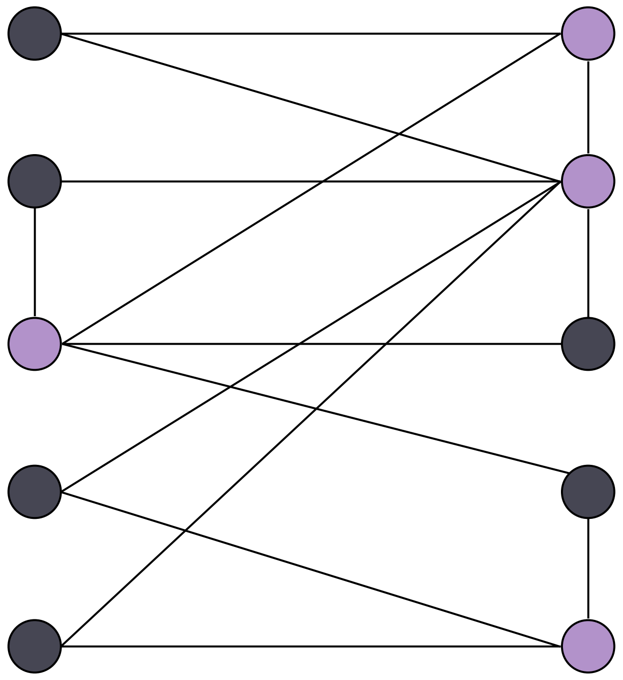
Reduction strategies

- Reduction by simple equivalence
- Reduction from special case to general case
- Reduction by encoding with gadgets

Independent set

MAXIMUM INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

- *Example*: Is there an independent set of size ≥ 6? Yes.
- *Example*: Is there an independent set of size ≥ 7? No.

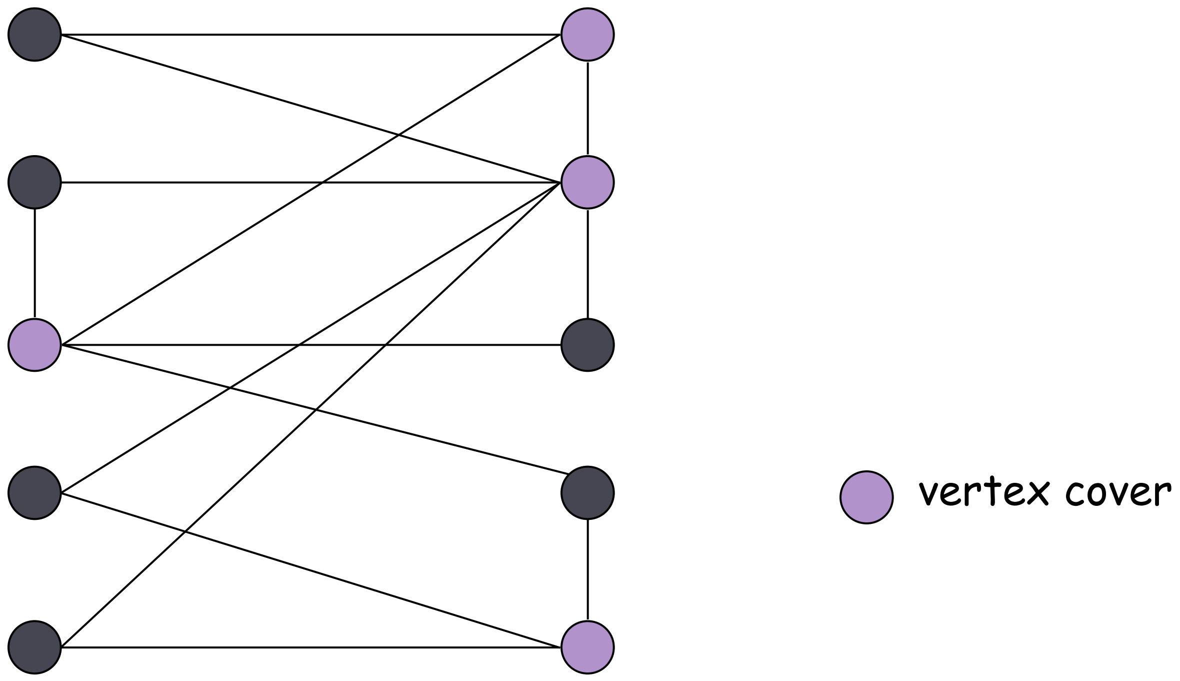


● independent set

Vertex cover

MINIMUM VERTEX COVER: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S?

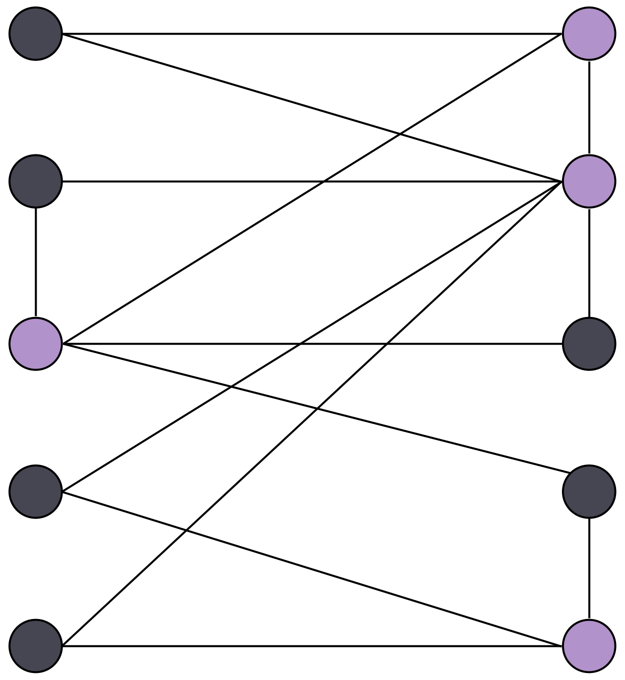
- *Example*: Is there a vertex cover of size ≤ 4? Yes.
- *Example*: Is there a vertex cover of size ≤ 3? No.



Vertex cover and independent set

Claim: VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Proof: S is an independent set iff $V \setminus S$ is a vertex cover.



● independent set

● vertex cover

Clique

CLIQUE: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each pair $x, y \in S$, (x, y) is an edge of E?

Claim: CLIQUE $\equiv_P$ INDEPENDENT-SET.

Proof: S is an independent set of G iff S is a clique of G', where G' is the complement of G: $G' = (V, V^2 - E)$.

Reduction strategies

- Reduction by simple equivalence
- Reduction from special case to general case
- Reduction by encoding with gadgets

Set cover

MINIMUM SET COVER: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k, does there exist a collection of $\leq k$ of these sets whose union is equal to U?

- *Sample application*:
  - m available pieces of software.
  - Set U of n functionalities that we would like our system to have.
  - The i^{th} piece of software provides the set $S_i \subseteq U$ of functionalities.
  - *Goal*: achieve all n functionalities using fewest pieces of software.

- *Example*:

U = { 1, 2, 3, 4, 5, 6, 7 }
k = 2

S_1 = {3, 7}	S_4 = {2, 4}
S_2 = {3, 4, 5, 6}	S_5 = {5}
S_3 = {1}	S_6 = {1, 2, 6, 7}

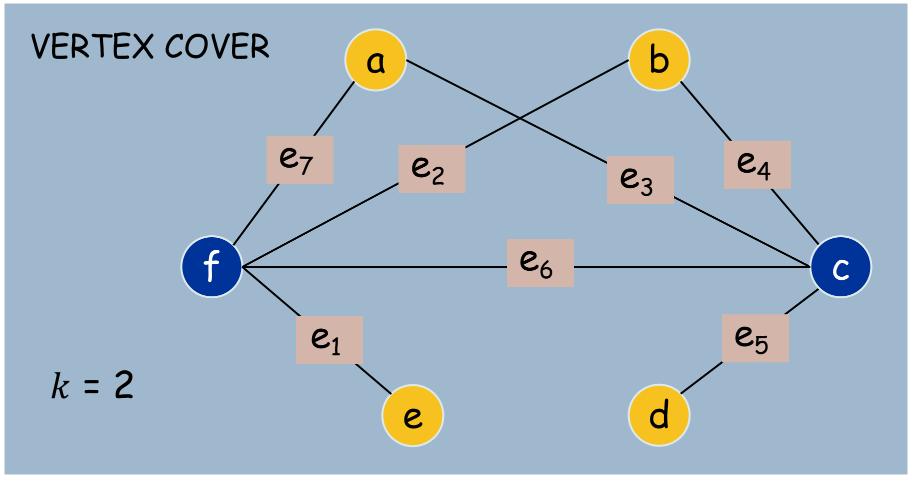
VERTEX-COVER $\leq_P$ SET COVER

Claim: VERTEX-COVER $\leq_P$ SET-COVER

Proof: Given a VERTEX-COVER instance $< G = (V, E), k >$, we construct a SET-COVER instance $< U, S_1, S_2, \ldots, S_m, k' >$ such that the first instance is true **iff** the second instance is true.

Construction:

- Create SET-COVER instance:
  - $U = E,\ S_v = \{e \in E: e \text{ incident to } v\}, k' = k$
- Set-cover of size $\leq k'$ **iff** vertex cover of size $\leq k$. ▪



SET COVER

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$
$k' = 2$

$S_a = \{3, 7\}$	$S_b = \{2, 4\}$
$S_c = \{3, 4, 5, 6\}$	$S_d = \{5\}$
$S_e = \{1\}$	$S_f = \{1, 2, 6, 7\}$

Reduction strategies

- Reduction by simple equivalence
- Reduction from special case to general case
- Reduction by encoding with gadgets - wait a moment :)

NP-hard and NP-complete

NP-hard: A problem X is NP-hard if, for every problem Y in NP, $Y \leq_p X$.

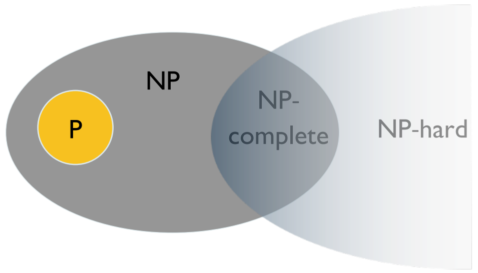
NP-complete: A problem X is NP-complete, if it is NP-hard and in NP.

Theorem: Suppose X is an NP-complete problem. Then X is solvable in poly-time iff P = NP.

Proof: $\Leftarrow$ If P = NP then X can be solved in poly-time since X is in P

$\Rightarrow$ Suppose X can be solved in poly-time.

- Let Y be any problem in NP. Since $Y \leq_p X$, we can solve Y in poly-time. This implies NP $\subseteq$ P.
- We already know P $\subseteq$ NP. Thus P = NP. ▪



If P $\neq$ NP

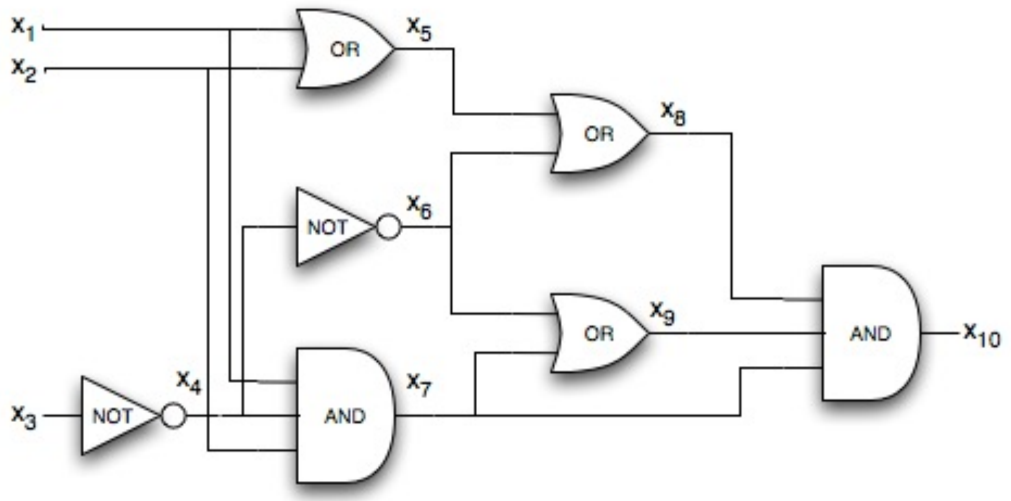
Fundamental question

Do there exist "natural" NP-complete problems?

The "First" NP-Complete Problem

Theorem [Cook 71, Levin 73]: CIRCUIT-SAT is NP-complete.

CIRCUIT-SAT: Given a Boolean circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?



How this was proved?

- from the definition of NP, that is
- by encoding an execution of a non-deterministic Turing machine by a boolean circuit
  - the input problem is solved by YES iff the circuit outputs 1
  - if the execution is polynomial-length then the circuit is polynomial-size

Establishing NP-Completeness

Remark: Once we established the first NP-complete problem, we can "bootstrap" and prove other problems NP-complete by reduction

Universal recipe to establish NP-completeness of problem X

- Step 1. Show that X is in NP.
- Step 2. Choose an NP-complete problem Y.
- Step 3. Prove that $Y \leq_P X$.

Justification: If Y is an NP-complete problem, and X is a problem in NP with the property that $Y \leq_P X$ then X is NP-complete.

Boolean satisfiability (SAT)

Literal: A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause: A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form: A propositional formula Φ that is the conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT: Given CNF formula Φ, does it have a satisfying truth assignment?

3-SAT: SAT where each clause contains exactly 3 literals.

each corresponds to a different variable

Ex: $\left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(x_2 \vee x_3\right) \wedge \left(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}\right)$

Yes: x_1 = true, x_2 = true x_3 = false.

3-SAT is NP-Complete

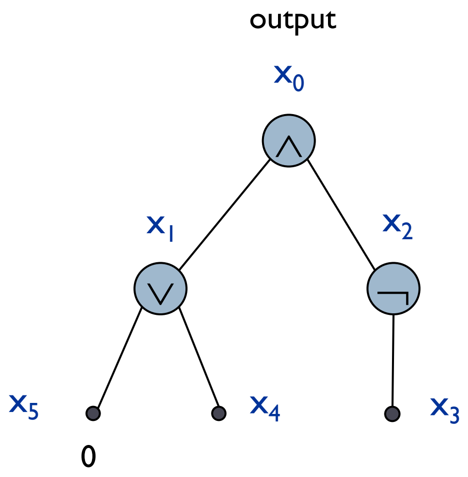
Theorem: 3-SAT is NP-complete.

Proof: Suffices to show that CIRCUIT-SAT $\le_P$ 3-SAT since 3-SAT is in NP.

- Let K be any circuit.
- Create a 3-SAT variable x_i for each circuit element i.
- Make circuit compute correct values at each node:
  - $x_2 = \neg x_3 \Rightarrow$ add 2 clauses: $x_2 \vee x_3$, $\overline{x_2} \vee \overline{x_3}$
  - $x_1 = x_4 \vee x_5 \Rightarrow$ add 3 clauses: $x_1 \vee \overline{x_4}$, $x_1 \vee \overline{x_5}$, $\overline{x_1} \vee x_4 \vee x_5$
  - $x_0 = x_1 \wedge x_2 \Rightarrow$ add 3 clauses: $\overline{x_0} \vee x_1$, $\overline{x_0} \vee x_2$, $x_0 \vee \overline{x_1} \vee \overline{x_2}$
- Constant input values and output value (1)
  - $x_5 = 0 \Rightarrow$ add 1 clause: $\overline{x_5}$
  - $x_0 = 1 \Rightarrow$ add 1 clause: x_0
- *Final step*: turn clauses of length < 3 into clauses of length exactly 3 by introducing new variables:

replace $y \vee z$ by $(y \vee z \vee p) \wedge (y \vee z \vee \bar{p})$

replace y by $(y \vee p \vee q) \wedge (y \vee \bar{p} \vee q) \wedge (y \vee p \vee \bar{q}) \wedge (y \vee \bar{p} \vee \bar{q})$



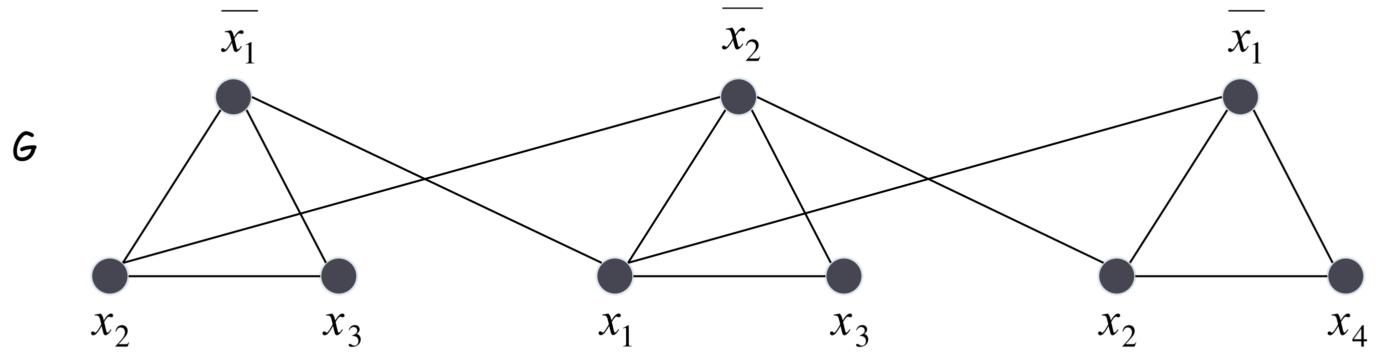
3-SAT Reduces to Independent Set

Theorem: 3-SAT $\leq_P$ INDEPENDENT-SET.

Proof: Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff Φ is satisfiable.

Construction.

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

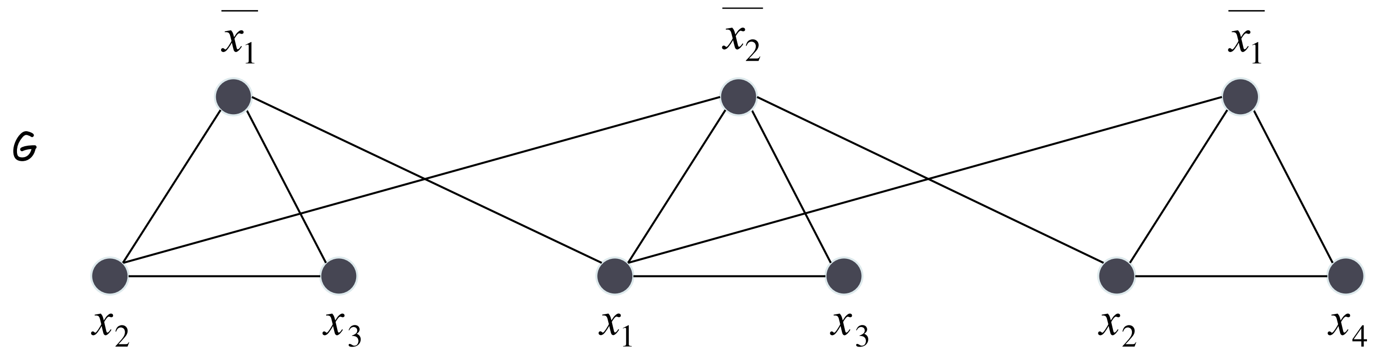
3-SAT Reduces to Independent Set

Claim: G contains independent set of size $k = |\Phi|$ iff Φ is satisfiable.

Proof: $\Rightarrow$ Let S be independent set of size k.

- S must contain exactly one vertex in each triangle.
- Set these literals to true. $\longleftarrow$ and any other variables in a consistent way
- Truth assignment is consistent and all clauses are satisfied.

$\Leftarrow$ Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k. ▪



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

Review

- Basic reduction strategies.
  - Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER
  - Special case to general case: VERTEX-COVER $\leq_P$ SET-COVER
  - Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET

Transitivity: If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

Proof idea: Compose the two algorithms.

Example: 3-SAT $\leq_P$ INDEPENDENT-SET $\leq_P$ VERTEX-COVER $\leq_P$ SET-COVER

What we showed so far

All problems below are NP-complete:

$$\begin{tikzcd}
\text{CIRCUIT-SAT} \arrow[d] & \\
\text{3-SAT} \arrow[d] & \\
\text{INDEPENDENT SET} \arrow[d, leftrightarrow] & \\
\text{VERTEX COVER} \arrow[r] \arrow[d, leftrightarrow] & \text{SET COVER} \\
\text{CLIQUE} &
\end{tikzcd}$$

Some more NP-complete problems

- SET-PACKING: max number of mutually disjoint sets
- INT-PROGRAMMING: linear programming on integers
- HAMILTONIAN-CYCLE
- TSP: traveling salesman problem
- 3-COLOR: coloring a graph (even planar!) *NB*: 2-color is in P
- SUBGRAPH-ISOMORPHISM: given two graphs, is the first one a subgraph of the second?
- LCS of multiple strings
- MULTISET-PARTITION: $\{3,1,1,2,2,1\} \rightarrow \{2,1,1,1\} \cup \{3,2\}$
- KNAPSACK: select a subset of "maximal value" fitting a knapsack
  - n objects, weights $w_1, \cdots, w_n$, values $v_1, \cdots, v_n$, knapsack weight capacity W
  - $\max_{S \subseteq 1..n} \{\sum_{i \in S} v_i \mid \sum_{i \in S} w_i \leq W\}$

Are there non-NP-complete problems that are not in P?

- Most of ”natural” NP problems are either in P or NP-complete
- Notable exceptions:
  - INTEGER FACTORIZATION
  - GRAPH-ISOMORPHISM

NP

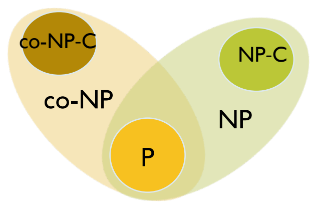
- A slight modification can transform a polynomial-time problem into NP-complete

Polynomial	**NP-complete**
Shortest path	Longest path
2-SAT	3-SAT
2-colorability	3-colorability
Bipartite vertex cover	Vertex cover
Maximum graph matching	3D matching
Minimum cut	Maximum cut
Minimum spanning tree	Degree-constrained spanning tree

NP and co-NP

- **co-NP**: problems whose complement is in NP

- *Examples*:
  - UNSATISFIABILITY (TAUTOLOGY)
  - NO-HAMILTONIAN-CYCLE



If P ≠ NP and NP-complete≠ co-NP-complete

  - INTEGER FACTORIZATION: in NP∩co-NP but not known to be in P

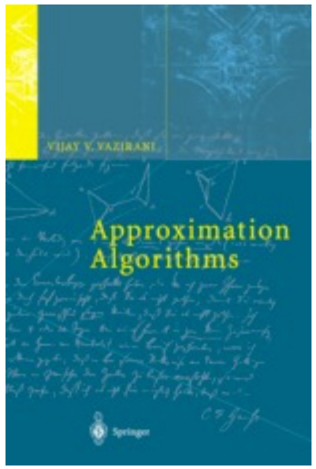
Coping with NP-completeness

- If a problem is NP-complete, you design an algorithm to do at most two of the following three things:
  1. Solve the problem exactly
  2. Guarantee to solve the problem in polynomial time
  3. Solve arbitrary instances of the problem

- 1+2: solving only small instances (e.g. via pseudo-polynomial algorithms)
  - cf. fixed-parameter tractability
  - e.g. Vertex Cover can be solved in $2^k n^{O(1)}$ by simple exhaustive search, where k is the size of Minimum Vertex Cover. Cf https://pacechallenge.org/2019/vc/
- 1+3: improved exponential-time algorithms, e.g.
  - 3-SAT can be solved in $O(1.48^n)$ instead of $O(2^n)$
  - 3-colorability can be solved in $O(1.3289^n)$
- 2+3: **approximation algorithms**, heuristics

- Note that NP-completeness has also advantageous consequences (cryptography)

Approximation algorithms: examples



- VERTEX COVER
  - has an easy 2-approximation algorithm
  - There is no 1.3606-approximation algorithm unless P=NP [Dinur, Safra 2005]
- SET-COVER can be $\log n$-approximated, where n is the size of the set to be covered ("universe")
- KNAPSACK with a running time $O(\frac{n^3}{\varepsilon})$ such that the computed solution verifies $V \geq (1 - \varepsilon) \cdot V^*$, where V is the computed total value and V^* the optimal total value (FPTAS)