Design and Analysis of Algorithms

Approximation Algorithms – Part 1

Pan Peng

School of Computer Science and Technology
University of Science and Technology of China

Outline

Complexity theory:

- Introduction: (In-)Tractability
- Polynomial time reductions
- Examples of reductions
- The complexity class NP
- NP-completeness

Introduction: (In-)Tractability

Tractable (efficiently solvable) problems

Normally, we say that a problem is efficiently solvable, if it can be solved in polynomial time, i.e. for such a problem there is a $O(n^c)$ algorithm, where:

- $n =$ input size (e.g. number of bits)
- $c =$ constant

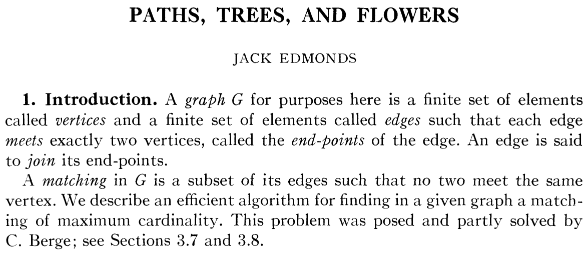
Efficiently solvable problems are also referred to as tractable problems.

Cobham–Edmonds Assumption:

- the assumption that tractability can be considered the same as solvability in polynomial time was pioneered by Alan Cobham and Jack Edmonds, who proposed this assumption in the 1960's.
- The assumption has become the predominant assumption in computer science over the last 50 years.

Definition: We use P to denote the class of all polynomial time solvable problems.

Edmonds 1963, Cobham 1965

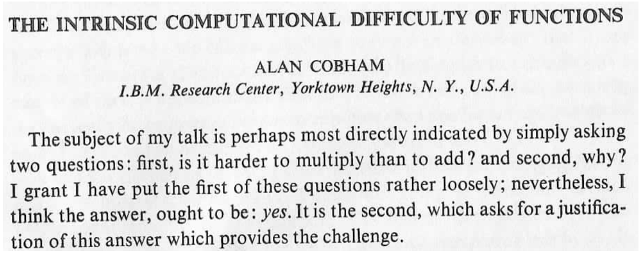


PATHS, TREES, AND FLOWERS

JACK EDMONDS

1. Introduction. A *graph* G for purposes here is a finite set of elements called *vertices* and a finite set of elements called *edges* such that each edge *meets* exactly two vertices, called the *end-points* of the edge. An edge is said to *join* its end-points.

A *matching* in G is a subset of its edges such that no two meet the same vertex. We describe an efficient algorithm for finding in a given graph a matching of maximum cardinality. This problem was posed and partly solved by C. Berge; see Sections 3.7 and 3.8.



THE INTRINSIC COMPUTATIONAL DIFFICULTY OF FUNCTIONS

ALAN COBHAM

I.B.M. Research Center, Yorktown Heights, N. Y., U.S.A.

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? I grant I have put the first of these questions rather loosely; nevertheless, I think the answer, ought to be: *yes*. It is the second, which asks for a justification of this answer which provides the challenge.

Discussion

Justification:

- In practice, polynomial time algorithms usually have small constants and exponents (e.g., linear, quadratic, or cubic).
- Overcoming the exponential bound of brute-force algorithms usually goes hand in hand with significant structural and combinatorial insights into the problem.

Exceptions and critism:

- Some polynomial time algorithms have huge constants and exponents and are considered useless in practice.
- Some exponential time algorithms are considered highly useful in practice since their worst-case behaviour only occurs rarely or the problem instances are sufficiently small.

Classifying Problems: P or not P?

Goal: classification of problems into those that can and those that cannot be solved in polynomial time ("are contained in the complexity class P").

The following problems can provably not be solved in polynomial time:

- Does a given Turing machine (algorithm) halt after k steps?
- Given a board position of $n \times n$ generalized chess, is one of the players guaranteed to win?

Classifying Problems: P or not P?

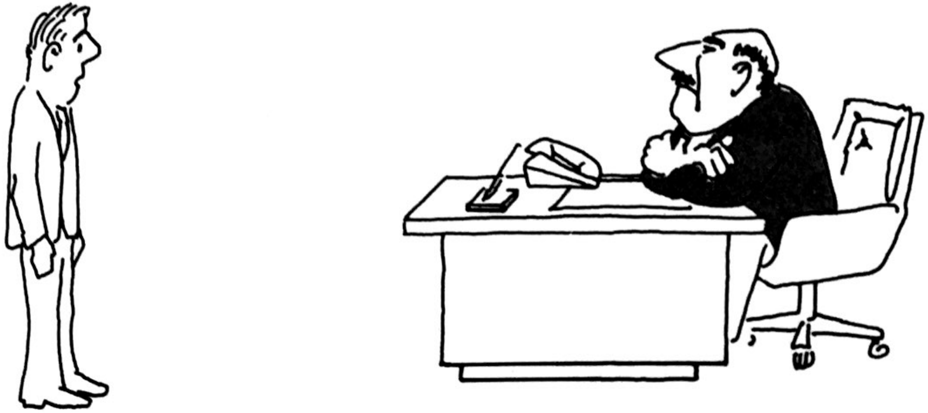
For the following problems neither a polynomial time algorithm nor a proof that they cannot be solved in polynomial time is known:

- Does a given graph contain at least k vertices that are pairwise non-adjacent?
- Can the vertices of a given graph be colored with 3 colors such that no two adjacent vertices have the same color?
- Is a given propositional formula satisfiable?

For many important fundamental problems no classification (into polynomial time or not polynomial time solvable) has been found thus far, which is a very unsatisfactory situation!

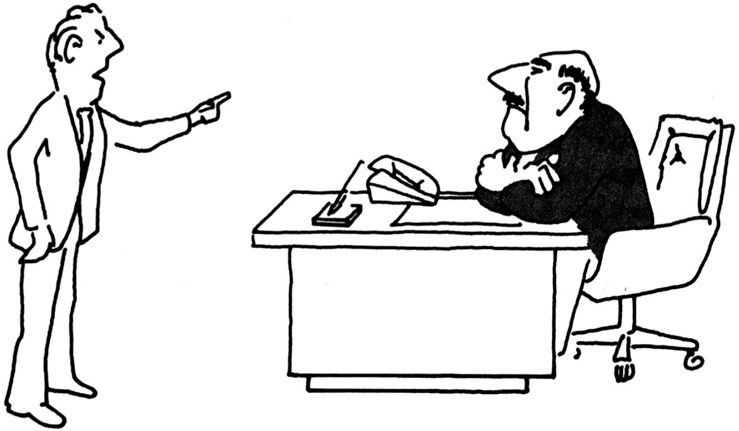
How should we proceed if we cannot solve a problem in polynomial time?

Not a Good Solution



“I can’t find an efficient algorithm, I guess I’m just too dumb.”

Would be the best solution, however, often not possible



“I can’t find an efficient algorithm, because no such algorithm is possible!”

Good Compromise

“I can’t find an efficient algorithm, but neither can all these famous people.”

Polynomial time Reductions

Equivalence w. r. t. solvability in polynomial time

We show that many fundamental problems that are unlikely to be solvable in polynomial time are in some sense equivalent and merely different versions of the same unique underlying difficult problem.

For this, we use an important concept called polynomial time reductions, that allow the "translation" between different problems that are equivalent w. r. t. polynomial time solvability.

Decision problems

Simplifying assumption: For simplicity, we restrict ourselves to decision problems, i.e. computational problems whose solution or output is either **YES** or **NO**.

Decision problem:

- a decision problem is a problem whose solution is either **YES** or **NO**.
- in contrast to functional or optimization problems that return a solution or a set of solutions.
- Typically: every functional or optimization problem has a natural corresponding decision problem and vice versa

Decision problems

Simplifying assumption: For simplicity, we restrict ourselves to decision problems, i.e. computational problems whose solution or output is either **YES** or **NO**.

Decision problem:

- a decision problem is a problem whose solution is either **YES** or **NO**.
- in contrast to functional or optimization problems that return a solution or a set of solutions.
- Typically: every functional or optimization problem has a natural corresponding decision problem and vice versa

Example (Minimum Spanning Tree):

- Decision problem: Does a given graph have a spanning tree with cost at most k?
- Functional problem: Find a spanning tree of a given graph with cost at most k.
- Optimization problem: Find a spanning tree of a given graph with minimum cost.

Polynomial time reductions

Question: Suppose we can solve some problem in polynomial time. Can we use this to show that we can also solve other problems in polynomial time?

Question: How can we establish a connection between different problems?

Reduction: Problem X can be polynomially reduced to problem Y.

$$\begin{tikzcd}
\boxed{\text{Problem } X} \arrow[r, "\text{Reduction}"] & \boxed{\text{Problem } Y}
\end{tikzcd}$$

Informal: If X can be polynomially reduced to Y and if we can solve Y in polynomial time, then we can also solve X in polynomial time.

That is, we reduce an instance x of X to an instance y of Y and then solve y. If y is a **YES**-instance of Y, then is x a **YES**-instance of X and vice versa.

Polynomial time reduction

Definition: A polynomial time reduction from a problem X to a problem Y is an algorithm $\mathcal{A}$ that for every instance x of X computes an instance y of Y such that:

1. x is a **YES**-instance of X if and only if y is a **YES**-instance of Y (**equivalence**).
2. $\mathcal{A}$ runs in **polynomial time**, i.e., there is a constant c such that given an instance x of X with size n, $\mathcal{A}$ computes the corresponding instance y of Y in time $O(n^c)$.

Polynomial time reduction

Definition: A polynomial time reduction from a problem X to a problem Y is an algorithm $\mathcal{A}$ that for every instance x of X computes an instance y of Y such that:

1. x is a **YES**-instance of X if and only if y is a **YES**-instance of Y (**equivalence**).
2. $\mathcal{A}$ runs in **polynomial time**, i.e., there is a constant c such that given an instance x of X with size n, $\mathcal{A}$ computes the corresponding instance y of Y in time $O(n^c)$.

Notation: We write $X \leq_P Y$, if there is a polynomial time reduction from X to Y. We also say that "X can be polynomially reduced to Y".

Polynomial time reduction

Definition: A polynomial time reduction from a problem X to a problem Y is an algorithm $\mathcal{A}$ that for every instance x of X computes an instance y of Y such that:

1. x is a **YES**-instance of X if and only if y is a **YES**-instance of Y (**equivalence**).
2. $\mathcal{A}$ runs in **polynomial time**, i.e., there is a constant c such that given an instance x of X with size n, $\mathcal{A}$ computes the corresponding instance y of Y in time $O(n^c)$.

Notation: We write $X \leq_P Y$, if there is a polynomial time reduction from X to Y. We also say that "X can be polynomially reduced to Y".

Remark: If $X \leq_P Y$, then intuitively "Y is at least as hard as X".

Solving problems via reductions

Idea: Show that a problem can be solved in polynomial time by polynomially reducing it to a problem that can be solved in polynomial time.

Lemma: If $X \leq_P Y$ and Y can be solved in polynomial time, then also X can be solved in polynomial time.

Proof:

- Let $\mathcal{A}_{XY}$ be the algorithm that reduces X to Y in time $O(n^a)$ for some a. Let $\mathcal{A}_Y$ be the algorithm that solves Y in time $O(n^b)$ for some b.

Solving problems via reductions

Idea: Show that a problem can be solved in polynomial time by polynomially reducing it to a problem that can be solved in polynomial time.

Lemma: If $X \leq_P Y$ and Y can be solved in polynomial time, then also X can be solved in polynomial time.

Proof:

- Let $\mathcal{A}_{XY}$ be the algorithm that reduces X to Y in time $O(n^a)$ for some a. Let $\mathcal{A}_Y$ be the algorithm that solves Y in time $O(n^b)$ for some b.
- Given an instance x of X of size n, we first apply $\mathcal{A}_{XY}$ to x and obtain in time $O(n^a)$ an (equivalent) instance y of Y.
- The size of y is at most $O(n^a)$, since y was created in time $O(n^a)$.

Solving problems via reductions

Idea: Show that a problem can be solved in polynomial time by polynomially reducing it to a problem that can be solved in polynomial time.

Lemma: If $X \leq_P Y$ and Y can be solved in polynomial time, then also X can be solved in polynomial time.

Proof:

- Let $\mathcal{A}_{XY}$ be the algorithm that reduces X to Y in time $O(n^a)$ for some a. Let $\mathcal{A}_Y$ be the algorithm that solves Y in time $O(n^b)$ for some b.
- Given an instance x of X of size n, we first apply $\mathcal{A}_{XY}$ to x and obtain in time $O(n^a)$ an (equivalent) instance y of Y.
- The size of y is at most $O(n^a)$, since y was created in time $O(n^a)$. We then solve y using $\mathcal{A}_Y$ in time $O((n^a)^b) = O(n^{ab})$.
- Therefore, we solved x in time $O(n^a) + O(n^{ab}) = O(n^{ab})$, which is polynomial in n. $\square$

Showing intractability

Idea: We show that a problem is intractable by reducing to it from some other intractable problem.

Lemma: If $X \leq_P Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

Proof:

- Suppose Y can be solved in polynomial time.

Showing intractability

Idea: We show that a problem is intractable by reducing to it from some other intractable problem.

Lemma: If $X \leq_P Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

Proof:

- Suppose Y can be solved in polynomial time.
- Then, the previous lemma shows that this means that also X can be solved in polynomial time, a contradiction to our assumption that X cannot be solved in polynomial time. $\square$

Equivalence: If $X \leq_P Y$ and $Y \leq_P X$, then we write $X \equiv_P Y$.

🟩 intractable = not solvable in polynomial time

Transitivity

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

Proof:

- Let $\mathcal{A}_{XY}$ be the algorithm that shows $X \leq_P Y$ and uses time $O(n^a)$.
- Let $\mathcal{A}_{YZ}$ be the algorithm that shows $Y \leq_P Z$ and uses time $O(n^b)$.
- Let x be an instance of X of size n.

Transitivity

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

Proof:

- Let $\mathcal{A}_{XY}$ be the algorithm that shows $X \leq_P Y$ and uses time $O(n^a)$.
- Let $\mathcal{A}_{YZ}$ be the algorithm that shows $Y \leq_P Z$ and uses time $O(n^b)$.
- Let x be an instance of X of size n.
- Then, applying $\mathcal{A}_{XY}$ to x takes time $O(n^a)$.
- Moreover, $\mathcal{A}_{XY}$ creates an instance y of size at most $O(n^a)$.
- Therefore, the sub-sequent application of $\mathcal{A}_{YZ}$ on y takes time at most $O(n^{ab})$.
- Hence, applying $\mathcal{A}_{XY}$ and subsequently $\mathcal{A}_{YZ}$ runs in polynomial time and shows that $X \leq_P Z$. $\square$

Applying polynomial time reductions

In the following we will provide some examples of polynomial time reductions between problems.

When giving polynomial time reductions we need to ensure that:

1. The reduction is correct, i.e. the original instance is a **YES**-instance if and only if the reduced instance is a **YES**-instance.
2. The reduction is polynomial, i.e. the reduction can be applied in polynomial time w. r. t. the size of the original instance.

Remark: In some cases it is easier to show correctness and in other cases it is easier to show that the reduction is polynomial.

Basic strategies for reductions

- Reduction via simple equivalence.
- Reduction from special case to general case.
- Reduction via gadgets.

Independent Set $\equiv_P$ Vertex Cover

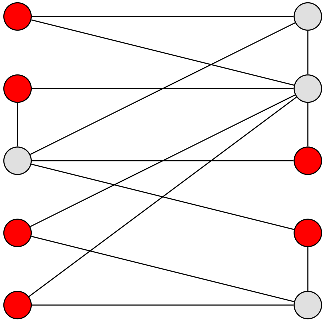
(Reduction via simple equivalence)

Independent Set

Input: A graph G and an integer k.

Question: Does G have an independent set of size at least k?

- Is there an independent set of size ≥ 6? $\Rightarrow$ **YES**.
- Is there an independent set of size ≥ 7? $\Rightarrow$ **NO**.

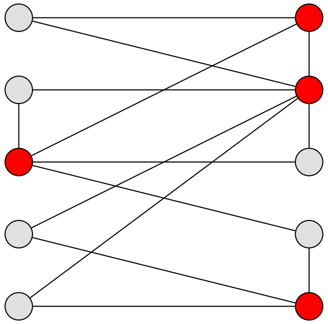


🟩 independent set = a set of vertices that are pairwise not adjacent

Vertex Cover

Input: A graph G and an integer k.

Question: Does G have a vertex cover of size at most k?.

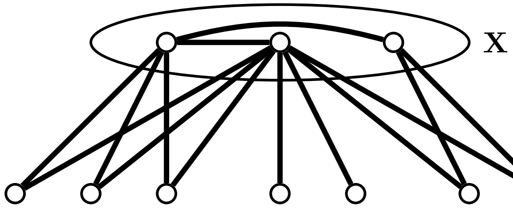


- Is there a vertex cover of size ≤ 4? $\Rightarrow$ **YES**.
- Is there a vertex cover of size ≤ 3? $\Rightarrow$ **NO**.

□ vertex cover $=$ a set of vertices that contains at least one endpoint from every edge

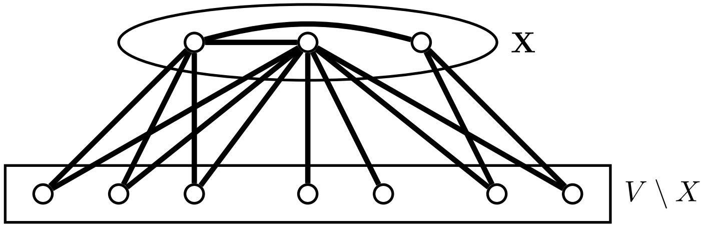
Vertex cover vs. independent set

Let X be a vertex cover of a graph $G = (V, E)$:



Vertex cover vs. independent set

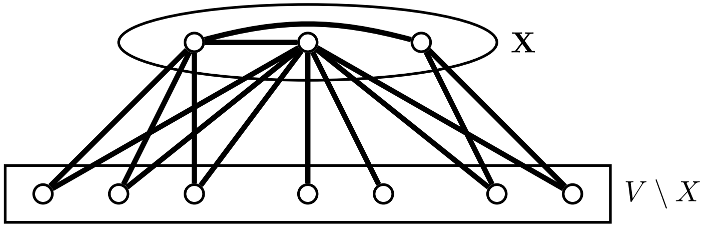
Let X be a vertex cover of a graph $G = (V, E)$:



Then $V \setminus X$ is an independent set!

Vertex cover vs. independent set

Let X be a vertex cover of a graph $G = (V, E)$:



Then $V \setminus X$ is an independent set!

Moreover, a set X is a vertex cover if and only if $V \setminus X$ is an independent set!

Independent Set $\leq_P$ Vertex Cover

Let (G, k) with $G = (V, E)$ be an instance of Independent Set. Then $(G, |V(G)| - k)$ is an equivalent instance of Vertex Cover.

Proof:
$(\Rightarrow)$ Let S be an independent set of size at least k. Then $V \setminus S$ is a vertex cover of size at most $|V| - k$.

$(\Leftarrow)$ Let S be a vertex cover of size at most k. Then $V \setminus S$ is an independent set of size at least $|V| - k$. $\square$

Independent Set $\leq_P$ Vertex Cover

Let (G, k) with $G = (V, E)$ be an instance of Independent Set. Then $(G, |V(G)| - k)$ is an equivalent instance of Vertex Cover.

Proof:
$(\Rightarrow)$ Let S be an independent set of size at least k. Then $V \setminus S$ is a vertex cover of size at most $|V| - k$.

$(\Leftarrow)$ Let S be a vertex cover of size at most k. Then $V \setminus S$ is an independent set of size at least $|V| - k$. $\square$

Remark: The same reduction also works from Vertex Cover to Independent Set and hence also Vertex Cover $\leq_P$ Independent Set.

Vertex Cover $\leq_P$ Set Cover

(Reduction from special case to general case)

Set cover

Set Cover

Input: A set of elements U, a collection $S_1, \ldots, S_m$ of subsets of U and an integer k.

Question: Does there exist a collection of $\leq k$ of these sets whose union is equal to U?

Example:

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

$$\begin{array}{ll} S_1 = \{3,7\} & S_4 = \{2,4\} \\ S_2 = \{3,4,5,6\} & S_5 = \{5\} \\ S_3 = \{1\} & S_6 = \{1,2,6,7\} \end{array}$$

Set cover

Set Cover

Input: A set of elements U, a collection $S_1, \ldots, S_m$ of subsets of U and an integer k.

Question: Does there exist a collection of $\leq k$ of these sets whose union is equal to U?

Example:

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

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

Set cover: sample applications

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

Vertex cover reduces to set cover

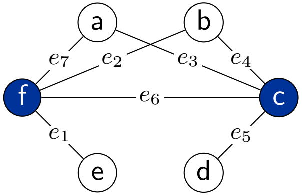
Lemma: Vertex Cover $\leq_P$ Set Cover.

Reduction:

- Given an instance (G, k) of Vertex Cover
- Create an instance of Set Cover by setting:
  - $k = k$,
  - $U = E$,
  - $S_v = \{e \in E : e \text{ incident to } v\}$ for every $v \in V(G)$
- set cover of size $\leq k$ iff vertex cover of size $\leq k$. $\square$

Vertex cover

$k = 2$



Set cover

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

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

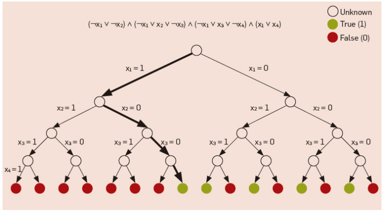
Satisfiability and 3-Satisfiability

Satisfiability (SAT)

- archetypical intractable (decision) problem
- often used to show intractability,
- even very restrictive versions of SAT are intractable,
- there are powerful and highly tuned algorithms (SAT-solvers) to solve SAT

  $\Rightarrow$ SAT is often used to solve other problems via a reduction to SAT!

$$
\begin{aligned}
\Phi := & (x_1 \lor \neg x_2 \lor x_4) \land \\
& (x_2 \lor \neg x_3 \lor \neg x_4) \land \\
& (\neg x_1 \lor x_3 \lor \neg x_4)
\end{aligned}
$$



Satisfiability

Satisfiability (SAT)

Input: A propositional formula in conjunction normal form.

Question: Is Φ satisfiable?

Conjunctive Normal Form (CNF):

- Φ is in conjunctive normal form if it is a conjunction of clauses, i.e.:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

- a clause is a disjunction of literals, i.e.:

$$C_j = l_1 \vee l_2 \vee \ldots \vee l_r$$

- a literal is either a propositional variable (e.g., x) or its negation (e.g., $\neg x$).

Satisfiability: Example

$$\Phi := \underbrace{(x_1 \vee \neg x_2)}_{\text{a clause}} \wedge (\underbrace{x_2}_{\text{a variable}} \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \underbrace{\neg x_4}_{\text{a literal}})$$

Satisfiability: Example

$$\Phi := \underbrace{(x_1 \vee \neg x_2)}_{\text{a clause}} \wedge (\underbrace{x_2}_{\text{a variable}} \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \underbrace{\neg x_4}_{\text{a literal}})$$

Conventions: We will consider Φ as a set of clauses and each clause as a set of literals and denote by $\mathsf{V}(\Phi)$ the set of variables of Φ.

Satisfiability: Example

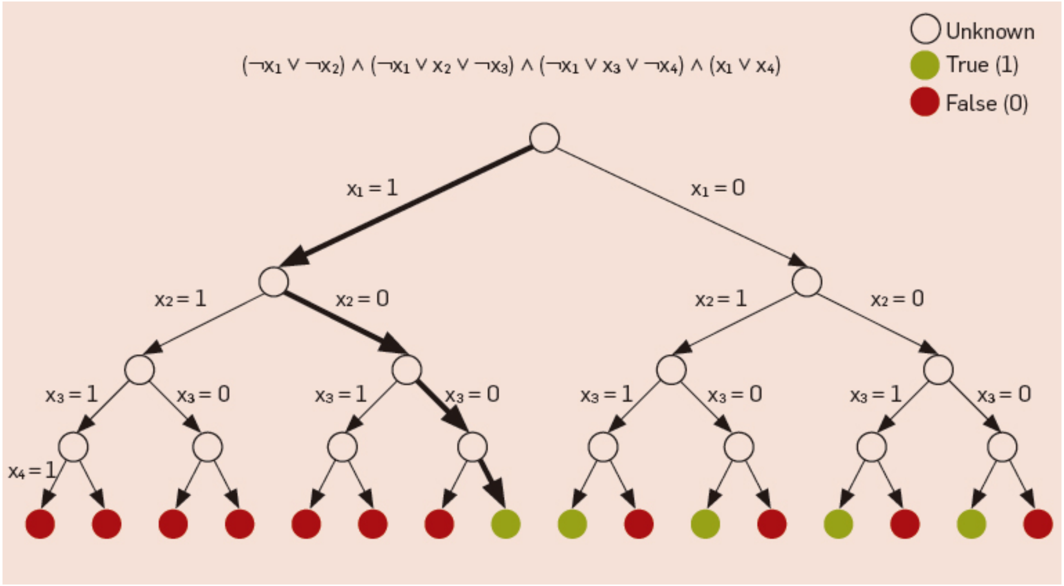
$$\Phi := \underbrace{(x_1 \vee \neg x_2)}_{\text{a clause}} \wedge (\underbrace{x_2}_{\text{a variable}} \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \underbrace{\neg x_4}_{\text{a literal}})$$

Conventions: We will consider Φ as a set of clauses and each clause as a set of literals and denote by $\mathsf{V}(\Phi)$ the set of variables of Φ.

The aim is to decide whether Φ is satisfiable, i.e., whether there is an assignment $\tau : \mathsf{V}(\Phi) \to \{0, 1\}$ satisfying Φ. The assignment τ satisfies Φ if every clause of Φ contains either:

- a variable x with $\tau(x) = 1$ or
- a literal $\neg x$ with $\tau(x) = 0$.

Satisfiability: Example



3-Satisfiability (3-SAT)

It is often more convenient to use the following more restrictive variant of SAT as a starting point for reductions:

3-Satisfiability (3-SAT)

Input: A propositional formula in conjunction normal form having exactly 3 literals per clause.

Question: Is Φ satisfiable?

Remarks:

- A CNF-formula that has exactly 3 literals per clause is usually referred to as 3-CNF formula.
- It was known that SAT $\equiv_P$ 3-SAT. Therefore, both problems have the "same" difficulty.

3-Satisfiability $\leq_P$ Independent Set

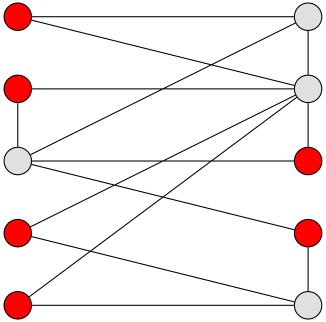
(Reduction via gadgets)

Independent Set

Input: A graph G and an integer k.

Question: Does G have an independent set of size at least k?

- Is there an independent set of size ≥ 6? $\Rightarrow$ **YES**.
- Is there an independent set of size ≥ 7? $\Rightarrow$ **NO**.

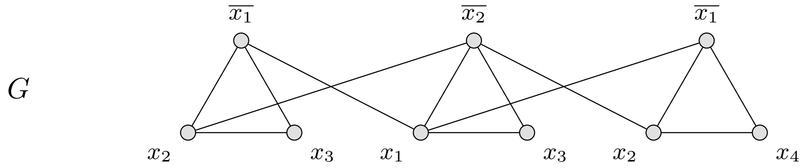


🟩 independent set $=$ a set of vertices that are pairwise not adjacent

The Reduction

Let Φ be an instance of 3-SAT with n variables and m clauses. We construct G and k as follows:

- G contains 3 vertices for each clause of Φ (one for each literal),
- connect the three literals in each clause to a triangle.
- connect each literal with all its negations.
- set $k = m$



$k = 3$

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

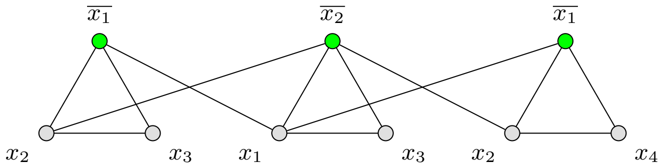
This reduction is clearly polynomial.

Correctness

Correctness: Φ is satisfiable if and only if G has an independent set of size at least k.

Proof: ($\Leftarrow$) Let S be an independent set of size $k = m$.

- S contains exactly one vertex per triangle (at most one because S is independent, at least one because $|S| = k$),
- Moreover, because of the edges between complementary literals S contains at most one literal, i.e., x or $\neg x$, from any variable.



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

Correctness

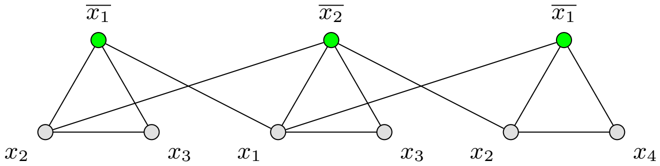
Correctness: Φ is satisfiable if and only if G has an independent set of size at least k.

Proof: ($\Leftarrow$) Let S be an independent set of size $k = m$.

Hence the assignment τ with:

- $\tau(x) = 1$ if $x \in S$,
- $\tau(x) = 0$ if $\neg x \in S$, and
- $\tau(x) \in \{0, 1\}$ arbitrary otherwise

satisfies Φ.



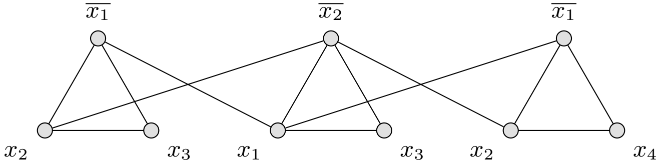
$$\Phi = (\overline{\mathbf{x_1}} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{\mathbf{x_2}} \vee x_3) \wedge (\overline{\mathbf{x_1}} \vee x_2 \vee x_4)$$

Correctness

Correctness: Φ is satisfiable if and only if G has an independent set of size at least k.

Proof: $\Rightarrow$ Let τ be a satisfying assignment for Φ.

- then τ satisfies at least one literal per clause and
- the satisfying literals are non-complementary,



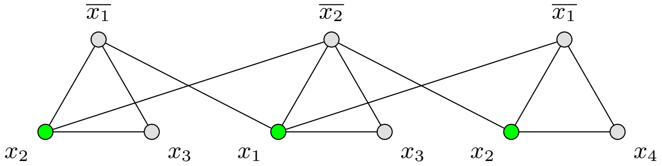
$$\Phi = (\overline{x_1} \vee \mathbf{x_2} \vee x_3) \wedge (\mathbf{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \mathbf{x_2} \vee x_4)$$

Correctness

Correctness: Φ is satisfiable if and only if G has an independent set of size at least k.

Proof: $\Rightarrow$ Let τ be a satisfying assignment for Φ.

- then τ satisfies at least one literal per clause and
- the satisfying literals are non-complementary,
- hence we obtain an independent set S of size $k = m$ by chosing a satisfied literal for each triangle (clause)



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

Recap: Basic Strategies for Reductions

- Simple equivalence:
  Independent Set $\equiv_P$ Vertex Cover.
- Special case to general case:
  Vertex Cover $\leq_P$ Set Cover.
- Coding 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: Combine two algorithms.

Example: 3-SAT $\leq_P$ Independent Set $\leq_P$ Vertex Cover $\leq_P$ Set Cover.

Decision vs. Finding vs. Optimization

Finding a Solution

- Thus far, we only considered decision problems that for every input either output **YES** or **NO**.
- However, in practice one is much more interested in finding a solution rather than merely deciding whether a solution exists.

Example: Vertex cover

Decide Vertex Cover

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

Find Vertex Cover

Input: A graph G and an integer k.

Question: Find a vertex cover of size at most k in G or output **NO** if no such vertex cover exists.

Example: Vertex cover

DECIDE VERTEX COVER

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

FIND VERTEX COVER

Input: A graph G and an integer k.

Question: Find a vertex cover of size at most k in G or output **NO** if no such vertex cover exists.

- Clearly, if we can solve FIND VERTEX COVER, we can also solve DECIDE VERTEX COVER.

Example: Vertex cover

Decide Vertex Cover

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

Find Vertex Cover

Input: A graph G and an integer k.

Question: Find a vertex cover of size at most k in G or output **NO** if no such vertex cover exists.

- Clearly, if we can solve Find Vertex Cover, we can also solve Decide Vertex Cover.
- Does the reverse also hold?

Example: vertex cover

Observation: If G has a vertex cover of size k, then there is a vertex $v \in V(G)$ such that $G - \{v\}$ has a vertex cover of size $k - 1$.

Idea: iterate over all vertices v, and decides if $G - \{v\}$ has a vertex cover of size $k - 1$, and update G and vertex cover correspondingly; Exercise

Example: vertex cover

Observation: If G has a vertex cover of size k, then there is a vertex $v \in V(G)$ such that $G - \{v\}$ has a vertex cover of size $k - 1$.

Idea: iterate over all vertices v, and decides if $G - \{v\}$ has a vertex cover of size $k - 1$, and update G and vertex cover correspondingly; Exercise

Therefore, if we can solve DECIDE VERTEX COVER in polynomial time we can also solve FIND VERTEX COVER in polynomial time.

Optimization vs Decision

- Many problems have natural optimization versions.
- As for finding, in general, optimization and decision versions are polynomial time equivalent

Example: vertex cover

Decide Vertex Cover

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

Minimum Vertex Cover

Input: A graph G.

Question: What is the minimum size of a vertex cover for G?

Example: vertex cover

Decide Vertex Cover

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

Minimum Vertex Cover

Input: A graph G.

Question: What is the minimum size of a vertex cover for G?

- Clearly, if we can solve Minimum Vertex Cover, we can also solve Decide Vertex Cover.

Example: vertex cover

Decide Vertex Cover

Input: A graph G and an integer k.

Question: Decide whether G has a vertex cover of size at most k.

versus:

Minimum Vertex Cover

Input: A graph G.

Question: What is the minimum size of a vertex cover for G?

- Clearly, if we can solve Minimum Vertex Cover, we can also solve Decide Vertex Cover.
- Does the reverse also hold?

Example: vertex cover

Algorithm: run algorithm for the decision version with every possible size $s \in \{0, |V(G)|\}$

- the algorithm works because every graph G has a vertex cover whose size is between 0 and $|V(G)|$
- even better: use binary search between 0 and $|V(G)|$

Therefore, if we can solve Decide Vertex Cover in polynomial time we can also solve Minimum Vertex Cover in polynomial time.

Example: independent set

DECIDE INDEPENDENT SET

Input: A graph G and an integer k.

Question: Decide whether G has an independent set of size at least k.

versus:

MAXIMUM INDEPENDENT SET

Input: A graph G.

Question: What is the maximum size of an independent set for G?

Example: independent set

DECIDE INDEPENDENT SET

Input: A graph G and an integer k.

Question: Decide whether G has an independent set of size at least k.

versus:

MAXIMUM INDEPENDENT SET

Input: A graph G.

Question: What is the maximum size of an independent set for G?

- Clearly, if we can solve MAXIMUM INDEPENDENT SET, we can also solve DECIDE INDEPENDENT SET.

Example: independent set

Decide Independent Set

Input: A graph G and an integer k.

Question: Decide whether G has an independent set of size at least k.

versus:

Maximum Independent Set

Input: A graph G.

Question: What is the maximum size of an independent set for G?

- Clearly, if we can solve Maximum Independent Set, we can also solve Decide Independent Set.
- Does the reverse also hold?

Example: independent set

Algorithm: run algorithm for the decision version with every possible size $s \in \{|V(G)|, 1\}$

- the algorithm works because every graph G has an independent set whose size is between $|V(G)|$ and 1
- even better: use binary search between $|V(G)|$ and 1

Therefore, if we can solve DECIDE INDEPENDENT SET in polynomial time we can also solve MAXIMUM INDEPENDENT SET in polynomial time.

Summary: Decision vs Finding vs Optimization

- in general finding an (optimal) solution and deciding whether a solution exists are polynomial time equivalent
- decision versions of problems are more suitable to show intractability!

The complexity class NP

The Class NP: Polynomial time certifier

Def. Algorithm $C(s,t)$ is a certifier for problem X if for every string s, $s \in X$ iff there exists a string t such that $C(s,t) = \texttt{yes}$.

🟩 *"certificate" or "witness"*

NP. Decision problems for which there exists a poly time certifier.

🟦 $C(s,t)$ *is a poly time algorithm and* $|t| \leq p(|s|)$ *for some polynomial* $p(\cdot)$.

Remark. NP stands for nondeterministic polynomial time.

Intuition:

- P $\approx$ problems that can be solved in polynomial time
- NP $\approx$ problems for which a (polynomial-sized) solution can be verified in polynomial time

The Class NP: Polynomial time Certifier (more formally)

Definition: NP $=$ class of all decision problems for which there exists a polynomial time certifier.

- Let X be a decision problem

The Class NP: Polynomial time Certifier (more formally)

Definition: NP $=$ class of all decision problems for which there exists a polynomial time certifier.

- Let X be a decision problem

- a certifier for X is an algorithm $C(s,t)$ that takes as input:
  - an input string s for X and
  - a certificate or witness t

The Class NP: Polynomial time Certifier (more formally)

Definition: NP $=$ class of all decision problems for which there exists a polynomial time certifier.

- Let X be a decision problem
- a certifier for X is an algorithm $C(s,t)$ that takes as input:
  - an input string s for X and
  - a certificate or witness t

  such that for every input s:

  $s \in X$ (i.e., s is a **YES** instance for X) $\Leftrightarrow$
  there is a witness t such that $C(s,t)$ answers **YES**

The Class NP: Polynomial time Certifier (more formally)

Definition: NP $=$ class of all decision problems for which there exists a polynomial time certifier.

- Let X be a decision problem

- a certifier for X is an algorithm $C(s,t)$ that takes as input:
  - an input string s for X and
  - a certificate or witness t

  such that for every input s:

  $s \in X$ (i.e., s is a **YES** instance for X) $\Leftrightarrow$
  there is a witness t such that $C(s,t)$ answers **YES**

- $C(s,t)$ is a polynomial time certifier for X if:
  - $C(s,t)$ is a certifier for X and
  - $C(s,t)$ runs in polynomial time and
  - $C(s,t)$ only accepts certificates t of polynomial length, i.e., $|t| \leq \text{poly}(|s|)$

Example: Independent Set

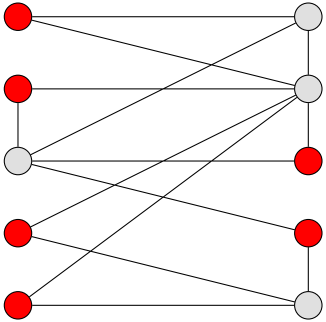
Independent Set

Input: A graph G and an integer k.

Question: Does G have an independent set of size at least k?

independent set $=$ a set of vertices that are pairwise not adjacent,

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



Example: Independent Set

Independent Set

Input: A graph G and an integer k.

Question: Does G have an independent set of size at least k?

independent set $=$ a set of vertices that are pairwise not adjacent,

Certificate/Witness: A set S of vertices of G of size at least k.

Certifier: "tests whether S is an independent set"

Algorithm: for every pair $u, v \in S$, check if u, v are adjacent

Satisfiability is in NP

Certificate. An assignment τ of truth values to the n boolean variables.

Certifier. Substitute the values of τ into Φ and evaluate the formula of constants in linear time.

Ex.

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$$

instance s

$$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1$$

certificate t

$$(0 \vee 1 \vee 0) \wedge (1 \vee 0 \vee 0) \wedge (1 \vee 1 \vee 1) \wedge (0 \vee 1 \vee 0)$$

substitution for s and t

Conclusion. SAT is in NP.

Certifiers and Certificates: Hamiltonian Cycle

Hamiltonian Cycle

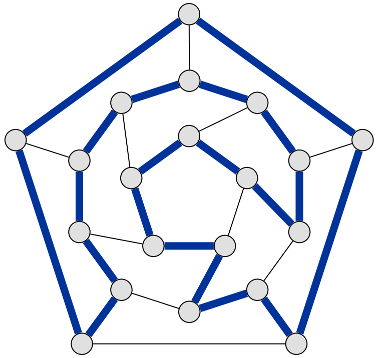
Input: An undirected graph G.

Question: Does there exist a (simple) cycle C that visits every node of G?

Certificate. A permutation of the n nodes.

Certifier. Check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

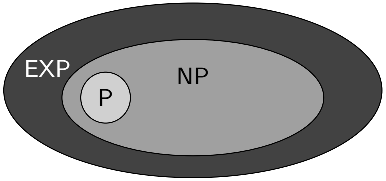
Conclusion. Hamiltonian Cycle is in NP.



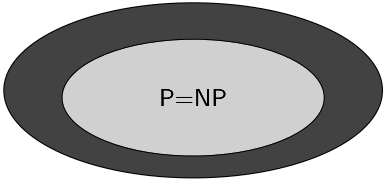
The Main Question: P Versus NP

Is P = NP**?** [Cook 1971, Edmonds, Levin, Yablonski, Gödel]

- Is the decision problem as easy as the certification problem?
- Clay $1 million prize.



If P ≠ NP



If P = NP

If yes: Efficient algorithms for Vertex Cover, Set Cover, SAT, . . .
If no: No efficient algorithms possible for Vertex Cover, Set Cover, SAT, . . .

Consensus opinion on P = NP? Probably no.

NP-**completeness**

NP-Complete

NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

NP-complete problems are in some sense the hardest problems in NP.

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

NP-Complete

NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

NP-complete problems are in some sense the hardest problems in NP.

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

($\Leftarrow$) If P $=$ NP then Y can be solved in poly time since Y is in NP.

NP-Complete

NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

NP-complete problems are in some sense the hardest problems in NP.

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

(⇐) If P $=$ NP then Y can be solved in poly time since Y is in NP.

(⇒) Suppose Y can be solved in poly time.

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

NP-Complete

NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

NP-complete problems are in some sense the hardest problems in NP.

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

($\Leftarrow$) If P $=$ NP then Y can be solved in poly time since Y is in NP.

($\Rightarrow$) Suppose Y can be solved in poly time.

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

Fundamental question. Do there exist “natural” NP-complete problems?

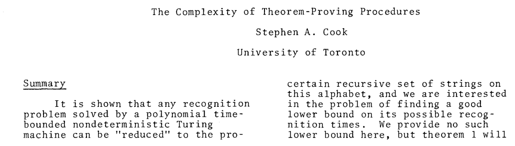
Is there a unique NP-complete problem?

General Idea:

- not a priori obvious.
- there could be several hardest NP-complete problems that cannot be reduced to one another.

Theorem: SAT is NP-complete. [Cook 1971, Levin 1973]

Cook 1971, Levin 1973



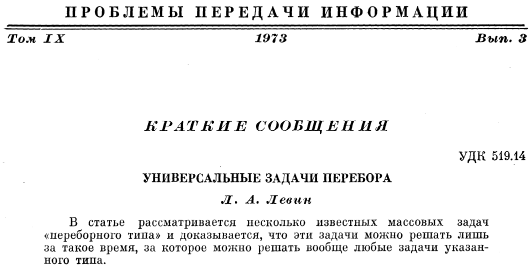
The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the pro- certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will



ПРОБЛЕМЫ ПЕРЕДАЧИ ИНФОРМАЦИИ

Том IX *1973* *Вып. 3*

КРАТКИЕ СООБЩЕНИЯ

УДК 519.14

УНИВЕРСАЛЬНЫЕ ЗАДАЧИ ПЕРЕБОРА

Л. А. Левин

В статье рассматривается несколько известных массовых задач «переборного типа» и доказывается, что эти задачи можно решать лишь за такое время, за которое можно решать вообще любые задачи указанного типа.

Establishing NP-Completeness

Remark. Once we establish first "natural" NP-complete problem, others fall like dominoes.

Recipe to establish NP-completeness of problem Y.

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

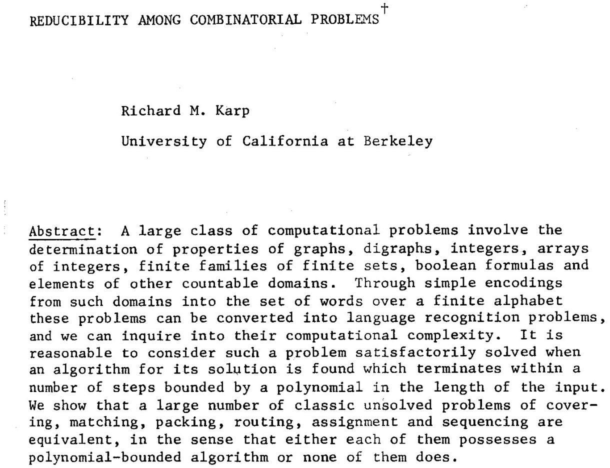
Justification. If X is an NP-complete problem, and Y is a problem in NP with the property that $X \leq_p Y$ then Y is NP-complete.

Pf. Let W be any problem in NP. Then $W \leq_p X \leq_p Y$.

🟩 *by definition of NP-complete* 🟦 *by assumption*

- By transitivity, $W \leq_p Y$.
- Hence Y is NP-complete. □

Karp 1972



REDUCIBILITY AMONG COMBINATORIAL PROBLEMS†

Richard M. Karp

University of California at Berkeley

Abstract: A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.

[Karp 1972] Karp contributed strongly to the establishment of the theory of NP-completeness for which he obtained the Turing award in 1985.

Extent and Impact of NP-Completeness

Extent of NP-completeness. [Papadimitriou 1995]

- Prime intellectual export of CS to other disciplines.
- 6,000 citations per year (title, abstract, keywords).
  - more than "compiler", "operating system", "database"
- Broad applicability and classification power.
- "Captures vast domains of computational, scientific, mathematical endeavors, and seems to roughly delimit what mathematicians and scientists had been aspiring to compute feasibly."

NP-completeness can guide scientific inquiry.

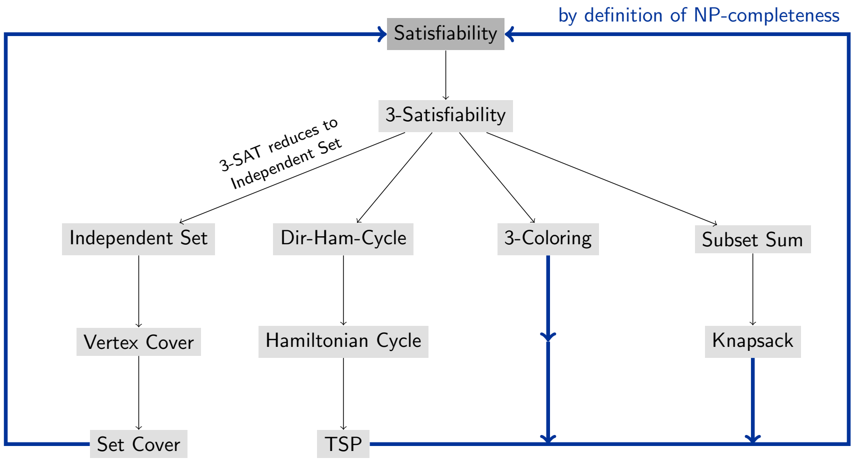
- 1926: Ising introduces simple model for phase transitions.
- 1944: Onsager solves 2D case in tour de force.
- 19xx: Feynman and other top minds seek 3D solution.
- 2000: Istrail proves 3D problem NP-complete.

More Hard Computational Problems

Aerospace engineering: optimal mesh partitioning for finite elements.
Biology: protein folding.
Chemical engineering: heat exchanger network synthesis.
Civil engineering: equilibrium of urban traffic flow.
Economics: computation of arbitrage in financial markets with friction.
Electrical engineering: VLSI layout.
Environmental engineering: optimal placement of contaminant sensors.
Financial engineering: find minimum risk portfolio of given return.
Game theory: find Nash equilibrium that maximizes social welfare.
Genomics: phylogeny reconstruction.
Mechanical engineering: structure of turbulence in sheared flows.
Medicine: reconstructing 3-D shape from biplane angiocardiogram.
Operations research: optimal resource allocation.
Physics: partition function of 3-D Ising model in statistical mechanics.
Politics: Shapley-Shubik voting power.
Pop culture: Minesweeper consistency.
Statistics: optimal experimental design.

NP-Completeness

Observation. All problems below are NP-complete and polynomial reduce to one another!



References

- J. Kleinberg, E. Tardos. Chapter 8. **Algorithm Design**. Addison Wesley, 2005.
- Boaz Barak and Sanjeev Arora. **Computational Complexity: A Modern Approach**. Cambridge University Press. 2009