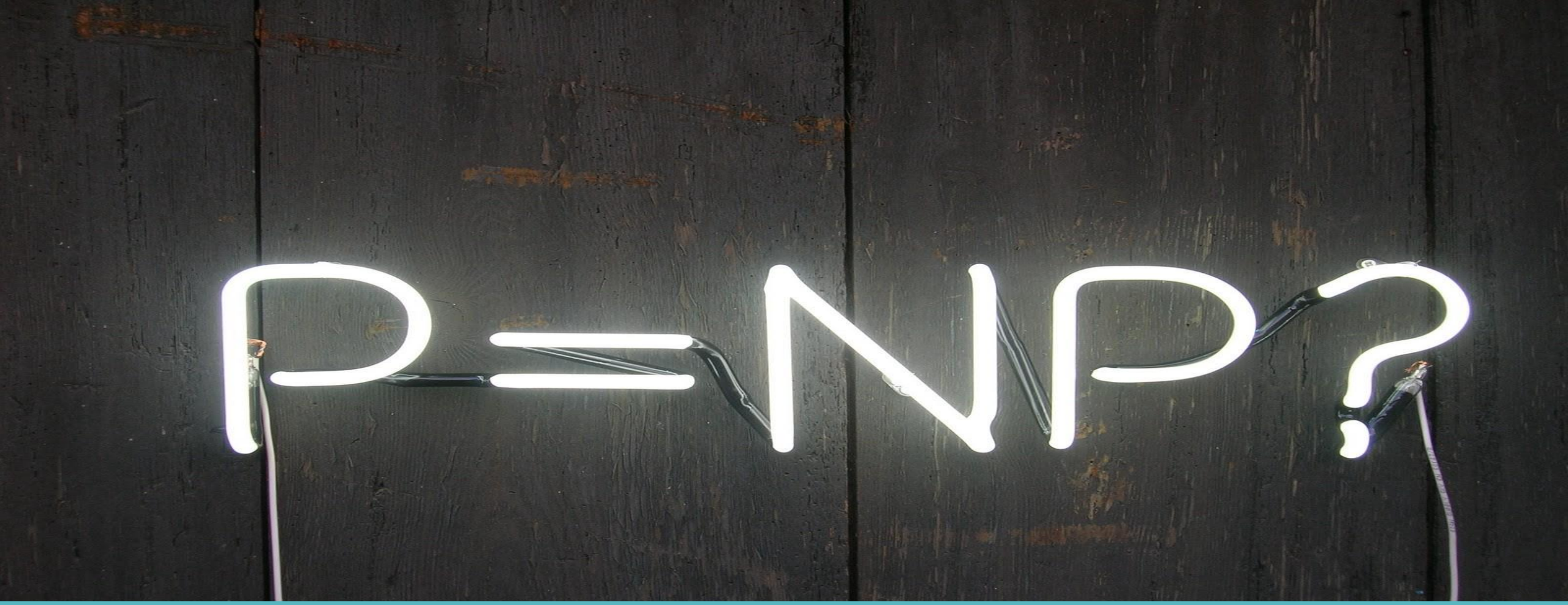


Algorithm Design and Analysis
NP Completeness (1)

http://ada.miulab.tw

Yun-Nung (Vivian) Chen

國立臺灣大學
National Taiwan University

Outline

- Decision Problems v.s. Optimization Problems
- Complexity Classes
  - P v.s. NP
  - NP, NP-Complete, NP-Hard
- Polynomial-Time Reduction

Algorithm Design & Analysis

- Design Strategy
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy Algorithms
  - Graph Algorithms
- Analysis
  - Amortized Analysis
  - NP-Completeness

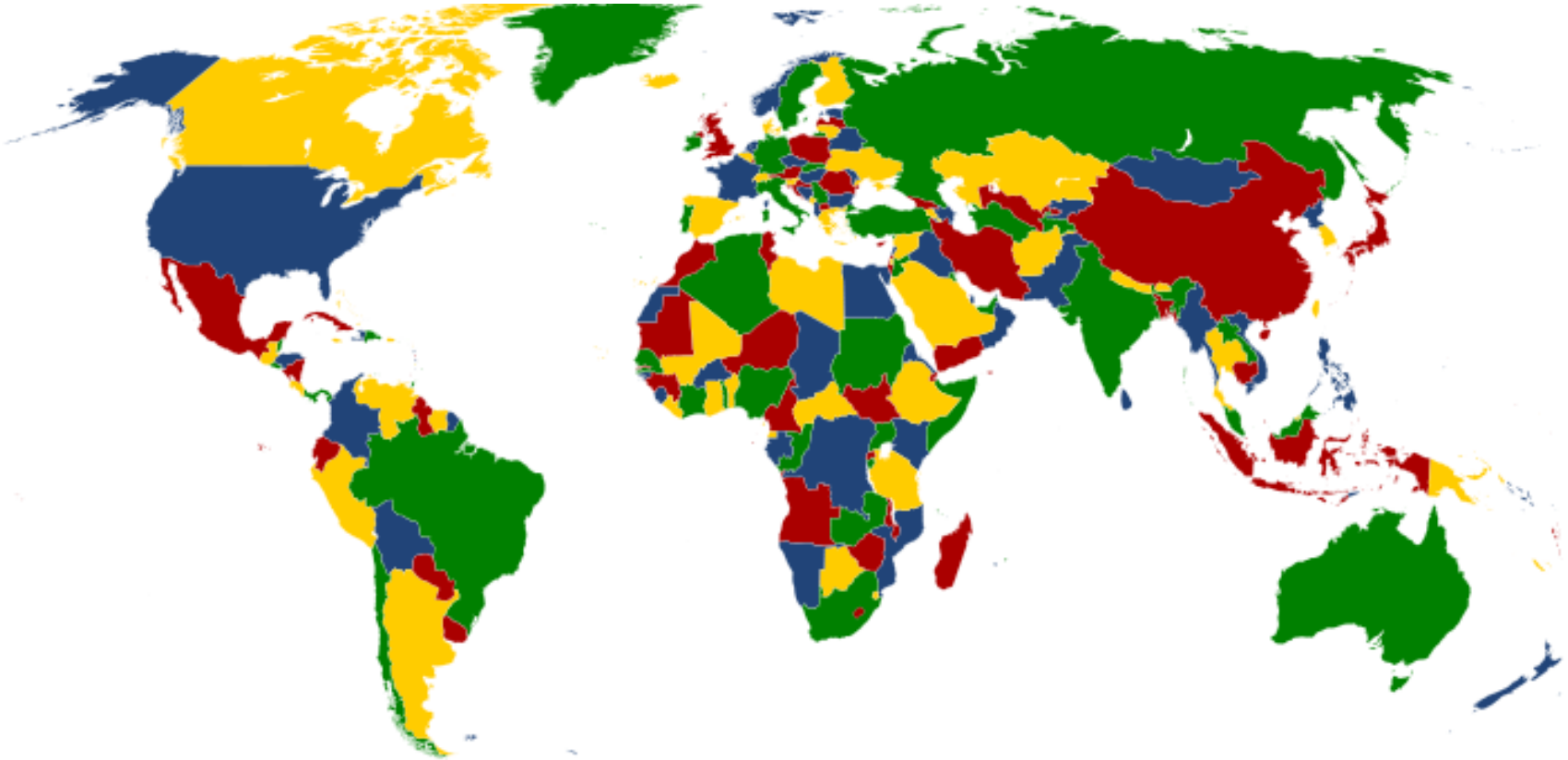
Polynomial Time Algorithms

- For an input with size *n*, the worst-case running time is $O(n^k)$ for some constant *k*

- Problems that are solvable by polynomial-time algorithms as being *tractable*, *easy*, *or efficient*
- Problems that require superpolynomial time as being *intractable*, or *hard*, *or inefficient*

Four Color Problem

- Use total four colors s.t. the neighboring parts have different colors



Four Color Problem (after 100 yrs)

- Finally proven (with the help of computers) by Kenneth Appel and Wolfgang Haken in 1976
  - Their algorithm runs in $O(n^2)$ time
- First major theorem proved by a computer
- Open problems remain...
  - Linear time algorithms to find a solution
  - Concise, human-checkable, mathematical proofs

Planar *k*-Colorability

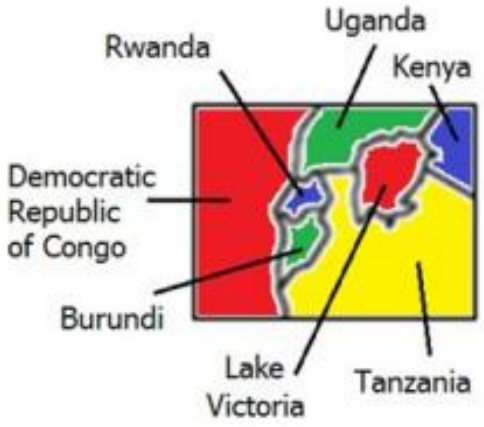
- Given a planar graph *G* (e.g., a map), can we color the vertices with *k* colors such that no adjacent vertices have the same color?
- $k = 1$?
- $k = 2$?
- $k = 3$?
- $k \geq 4$?

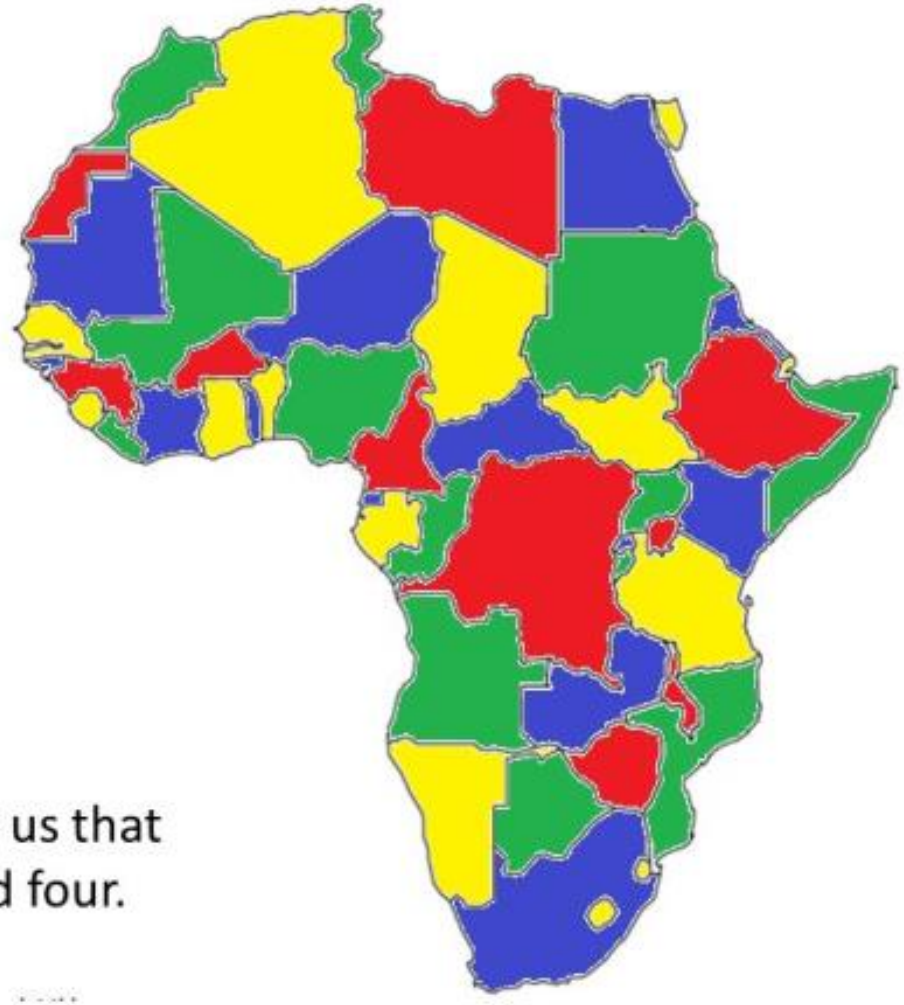
我想想~

How hard is it when $k = 3$?
Can we know its level of difficulty before solving it?

Planar *k*-Colorability

It turns out that
FOUR will do.



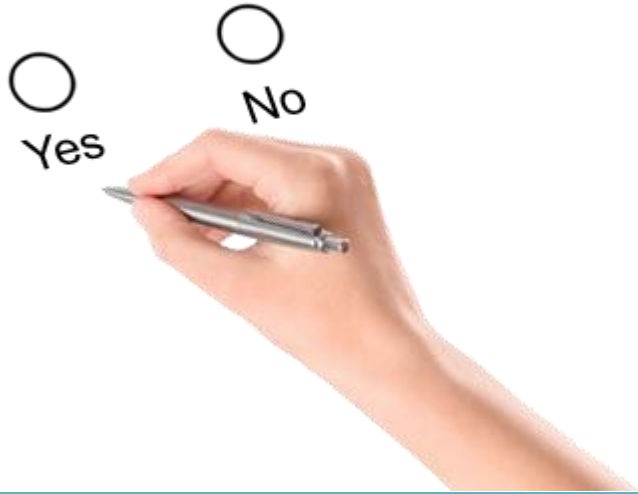


This section shows us that
we certainly need four.

Decision Problems v.s. Optimization Problems

Decision Problems

- Definition: the answer is simply “yes” or “no” (or “1” or “0”)
  - MST: Given a graph $G = (V, E)$ and a bound K, is there a spanning tree with a cost at most K?
  - KNAPSACK: Given a knapsack of capacity C, a set of objects with weights and values, and a target value V, is there a way to fill the knapsack with at least V value?



Optimization Problems

- Definition: each feasible solution has an associated value, and we wish to find a feasible solution with the best value (maximum or minimum)
  - MST-OPT: Given a graph $G = (V, E)$, find the minimum spanning tree of G
  - KNAPSACK-OPT: Given a knapsack of capacity C and a set of objects with weights and values, fill the knapsack so as to maximize the total value

Which is Easier? Why?

How to convert an optimization problem to a related decision problem?

Imposing a (lower or upper) bound on the value to be optimized

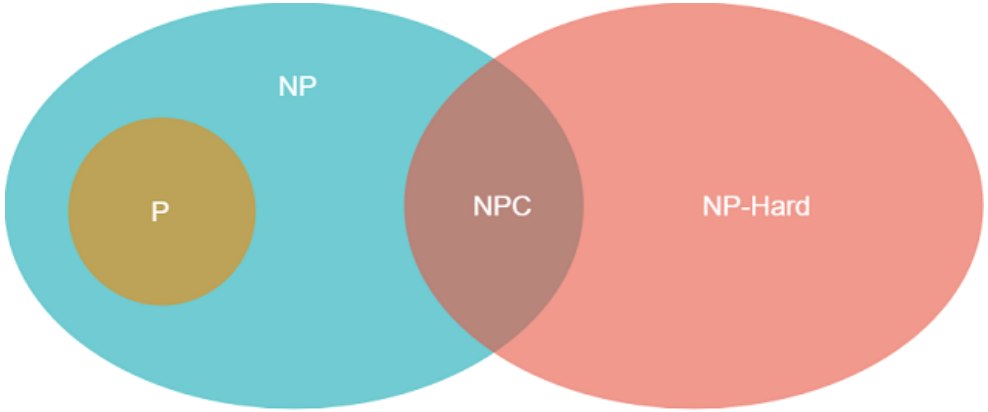
Difficulty Levels

- Every optimization problem has a decision version that is no harder than the optimization problem.

A_{opt}: given a graph, find the length of the shortest path

A_{dec}: given a graph, determine whether there is a path ≤ k

- ➡ Using A_{opt} to solve A_{dec}
  - check if the optimal value ≤ k, constant overhead
- ⬅ Using A_{dec} to solve A_{opt}
  - apply binary search on the value range, logarithmic overhead



P v.s. NP

Textbook Chapter 34 – NP-Completeness

Algorithm Design

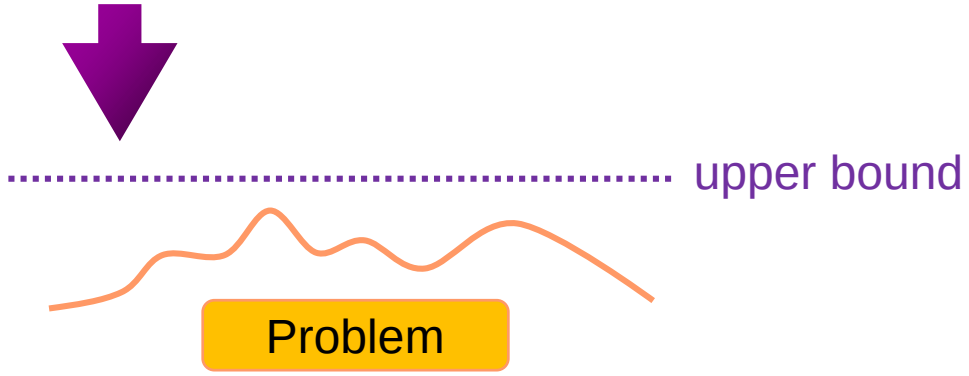
- Algorithmic design methods to solve problems efficiently (polynomial time)
  - Divide and conquer
  - Dynamic programming
  - Greedy
- “Hard” problems without known efficient algorithms
  - Hamilton, knapsack, etc.

Complexity Classes

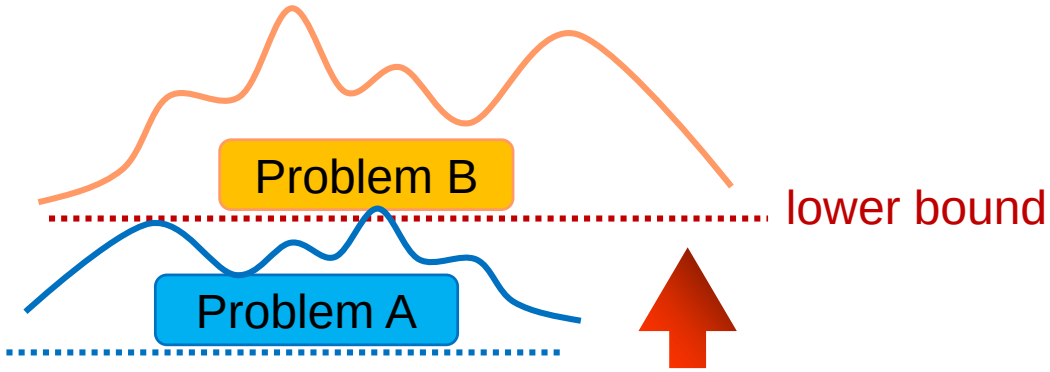
- Can we decide whether a problem is “too hard to solve” before investing our time in solving it?
- Idea: decide which *complexity classes* the problem belongs to via reduction
  - 已知問題A很難。若能證明問題B至少跟A一樣難，那麼問題B也很難。

To Solve v.s. Not to Solve

- Algorithm design
  - Design algorithms to solve computational problems
  - Mostly concerned with *upper bounds* on resources



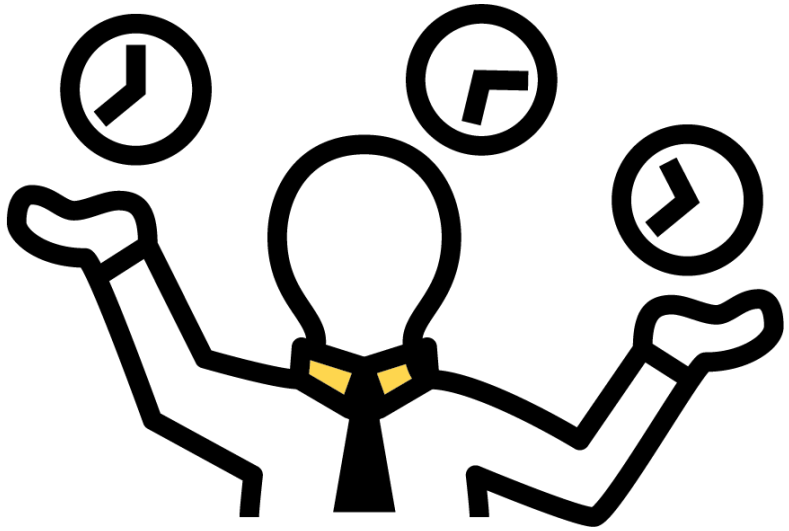
- Complexity theory
  - Classify problems based on their difficulty and identify relationships between classes
  - Mostly concerned with *lower bounds* on resources



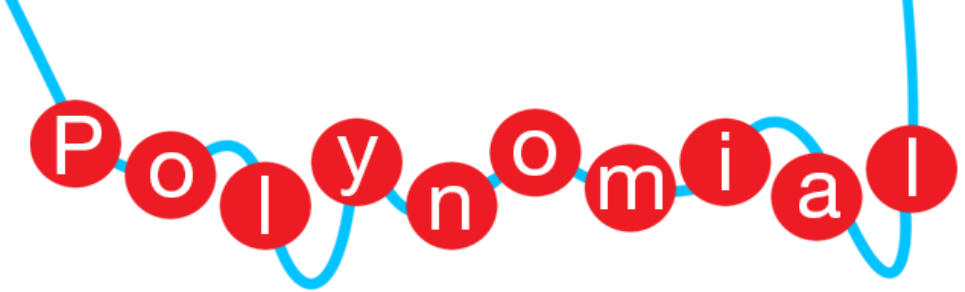
Problem B is no easier than A

Complexity Classes

- A complexity class is “a set of problems of related resource-based complexity”
  - Resource = time, memory, communication, ...
- Focus: *decision problems* and the resource of *time*

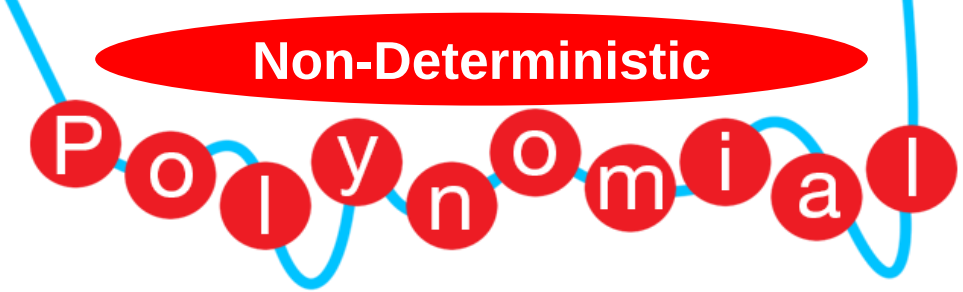


P



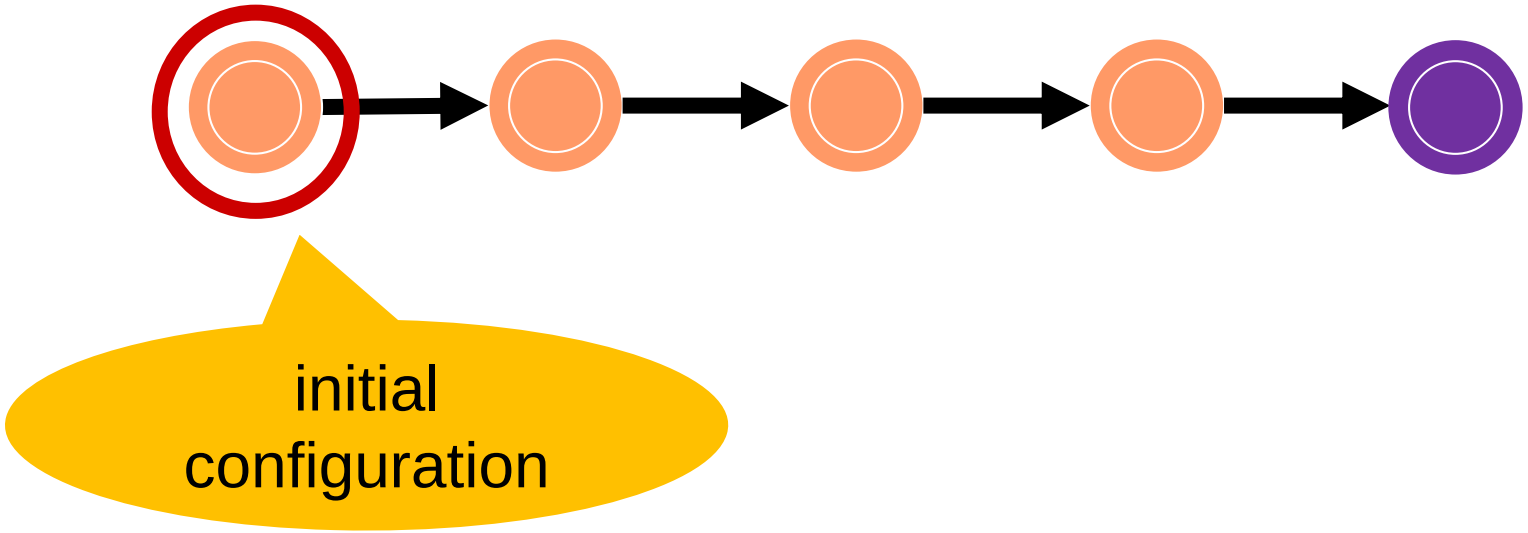
- The class **P** consists of all the problems that can be solved in *polynomial time*.
  - Sorting
  - Exact string matching
  - Primes
  - …
- Polynomial time algorithm
  - For inputs of size *n*, their worst-case running time is $O(n^k)$ for some constant *k*

NP

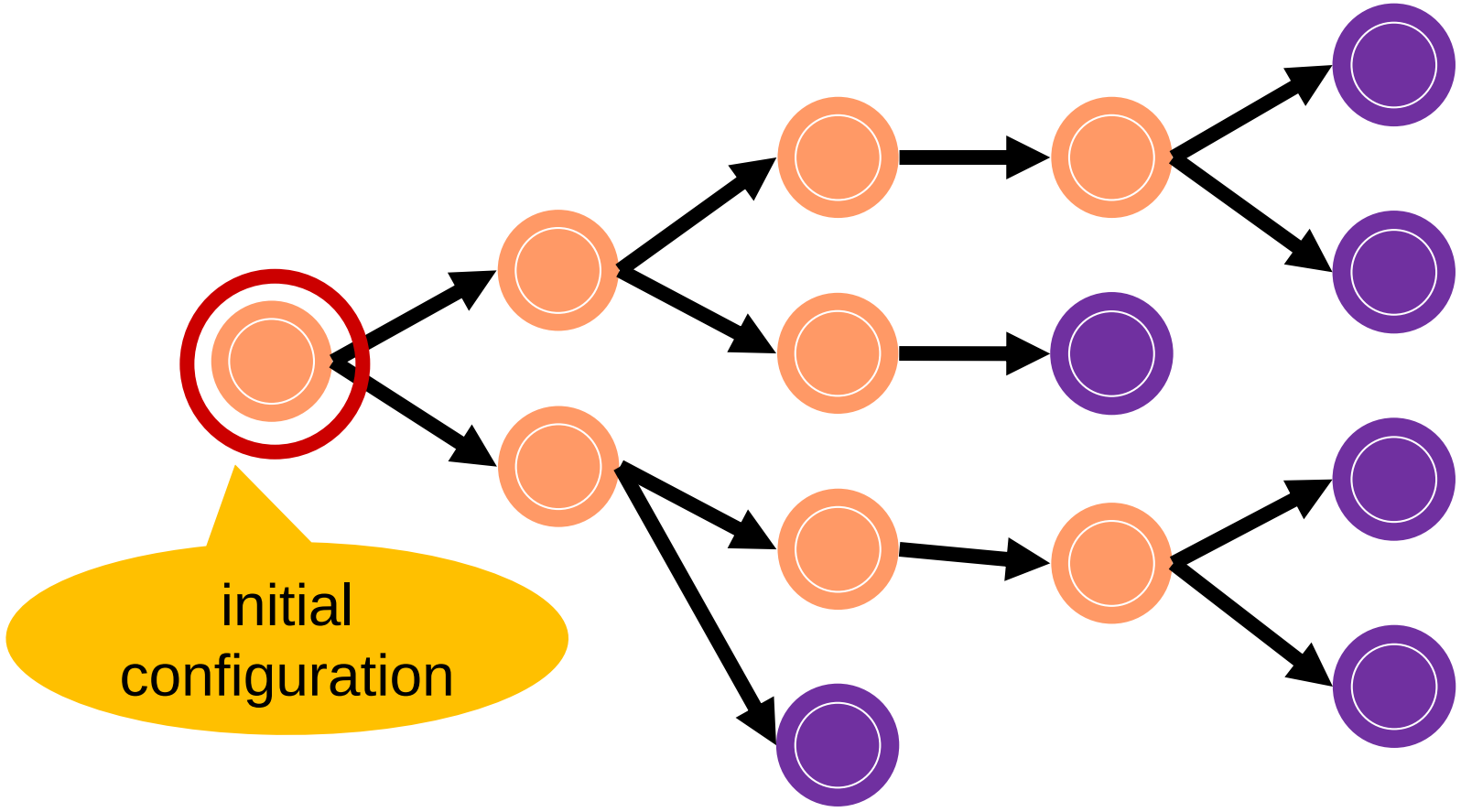


- NP consists of the problems that can be solved in *non-deterministically polynomial time*.
- NP consists of the problems that can be "verified" in polynomial time.
- P consists of the problems that can be solved in (deterministically) polynomial time.

Deterministic Algorithm



Non-Deterministic Algorithm



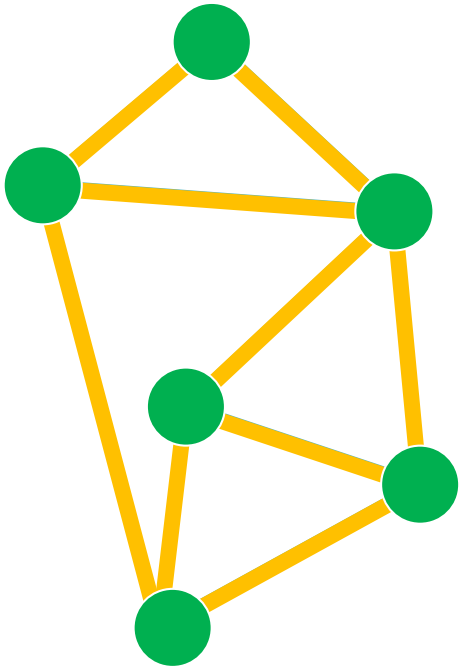
Non-Deterministic Bubble Sort

```
Non-Deterministic-Bubble-Sort(n)
  for i = 1 to n
    for j = 1 to n - 1
      if A[j] < A[i+1] then
        Either exchange A[j] and A[i+1] or do nothing
```

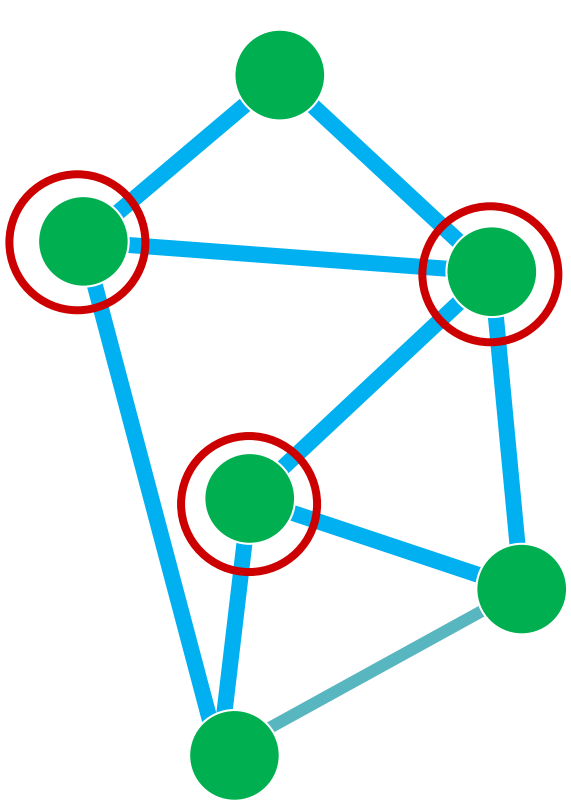
This is not a randomized algorithm.

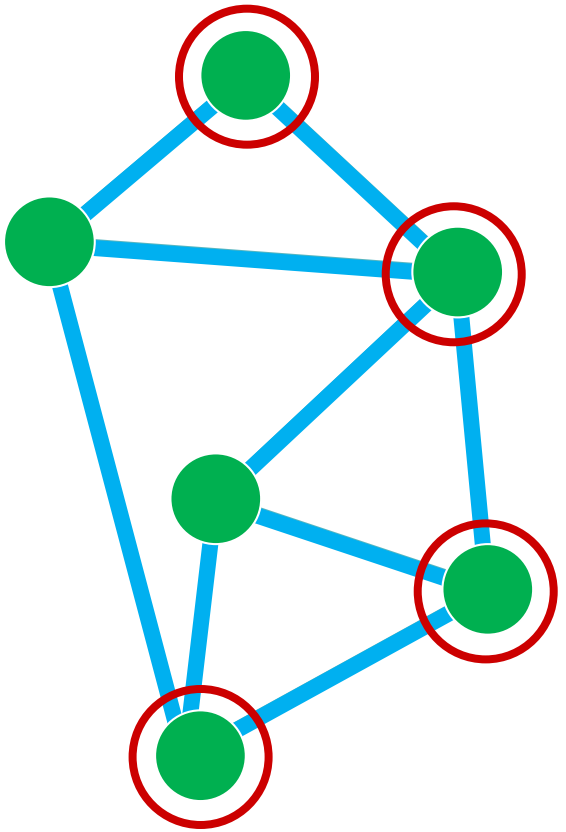
Vertex Cover Problem (路燈問題)

- Input: a graph *G*
- Output: a smallest vertex subset of *G* that covers all edges of *G*.
- Known to be NP-complete



Illustration





Vertex Cover (Decision Version)

- Input: a Graph *G* and an integer *k*.
- Output: Does *G* contain a vertex cover of size no more than *k*?

- Original problem → optimization problem
  - 原先的路燈問題是要算出放路燈的方法
- Yes/No → decision problem
  - 問*k*盞路燈夠不夠照亮整個公園

Non-Deterministic Algorithm

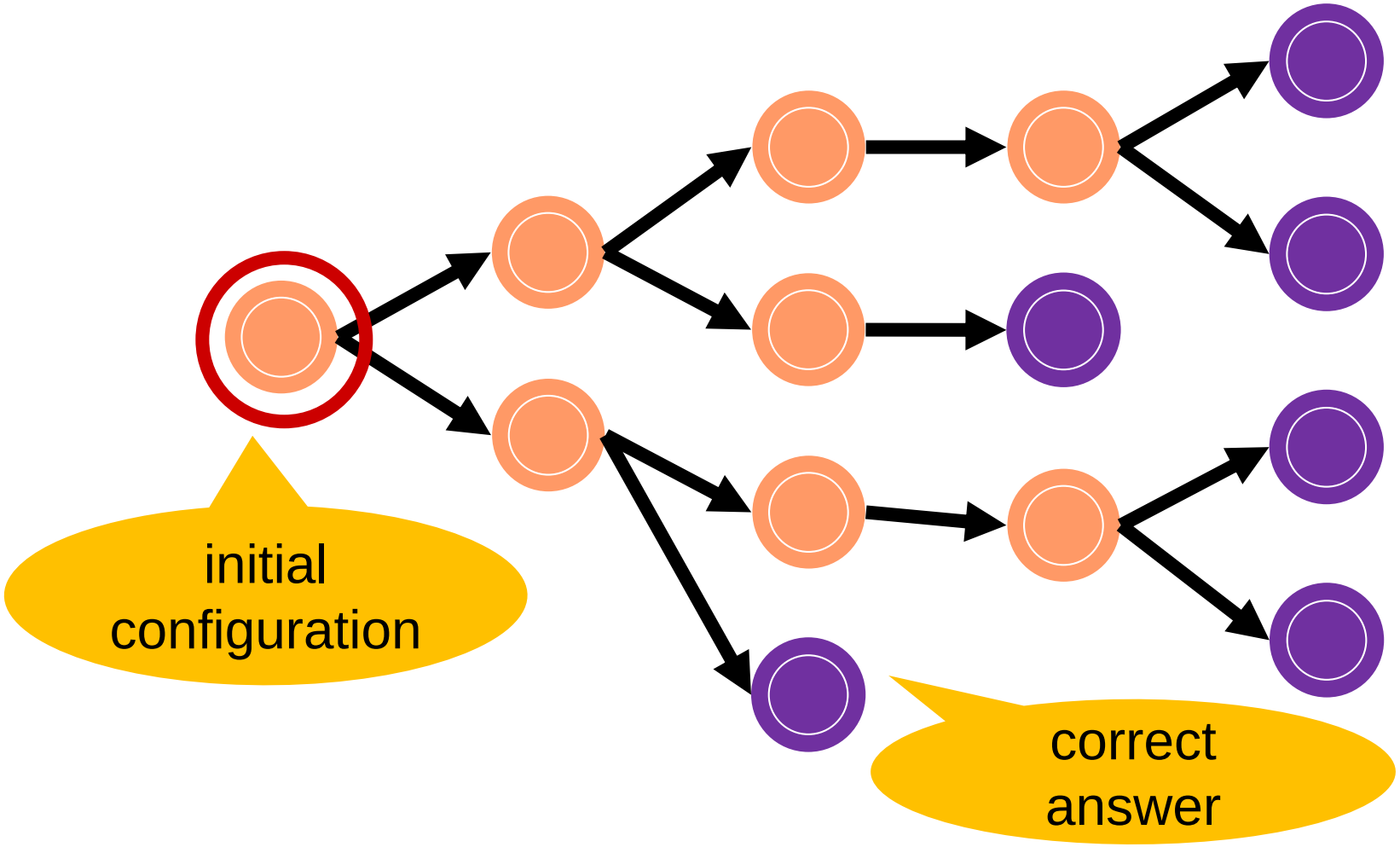
```
Non-Deterministic-Vertex-Cover(G, k)
  set S = {}
  for each vertex x of G
    non-deterministically insert x to S
  if |S| > k
    output no
  if S is not a vertex cover
    output no
  output yes
```

Algorithm Correctness

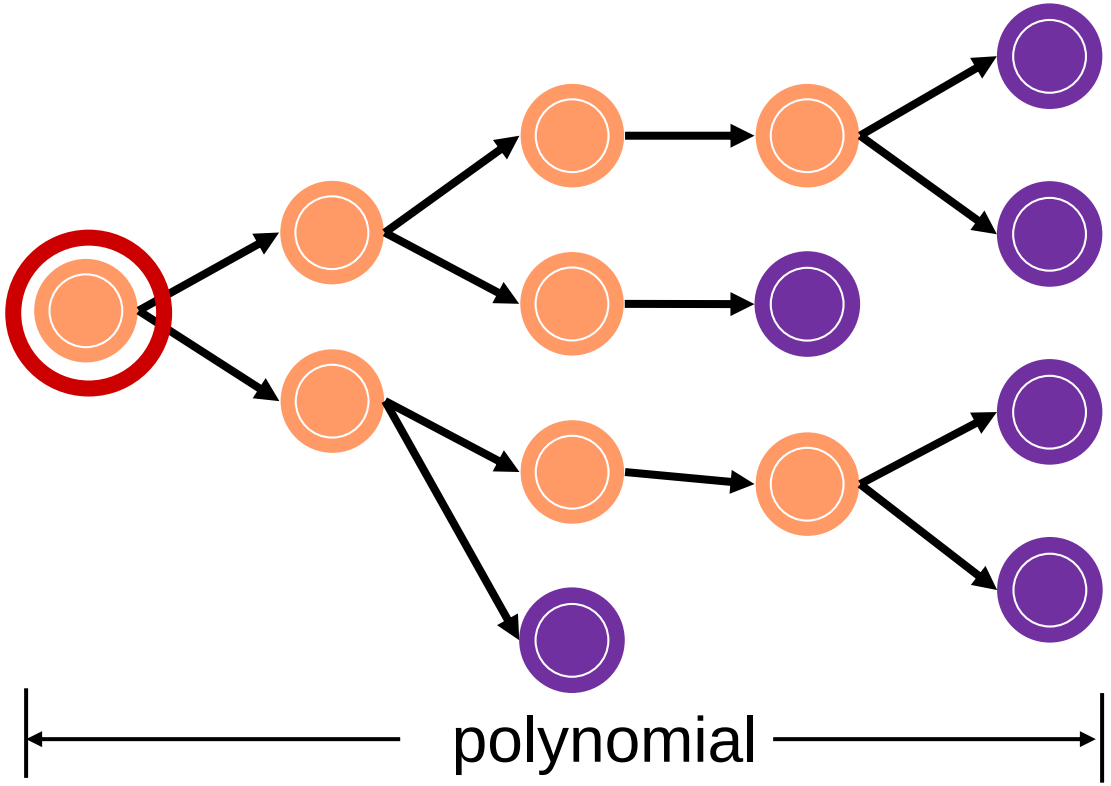
```
Non-Deterministic-Vertex-Cover(G, k)
  set S = {}
  for each vertex x of G
    non-deterministically insert x to S
  if |S| > k
    output no
  if S is not a vertex cover
    output no
  output yes
```

- If the correct answer is *yes*, then there is a computation path of the algorithm that leads to *yes*.
  - 至少有一條路是對的
- If the correct answer is *no*, then all computation paths of the algorithm lead to *no*.
  - 每一條路都是對的

Non-Deterministic Problem Solving



Non-Deterministic Polynomial



“solved” in non-deterministic polynomial time
= “verified” in polynomial time

P ⊆ NP or NP ⊆ P?

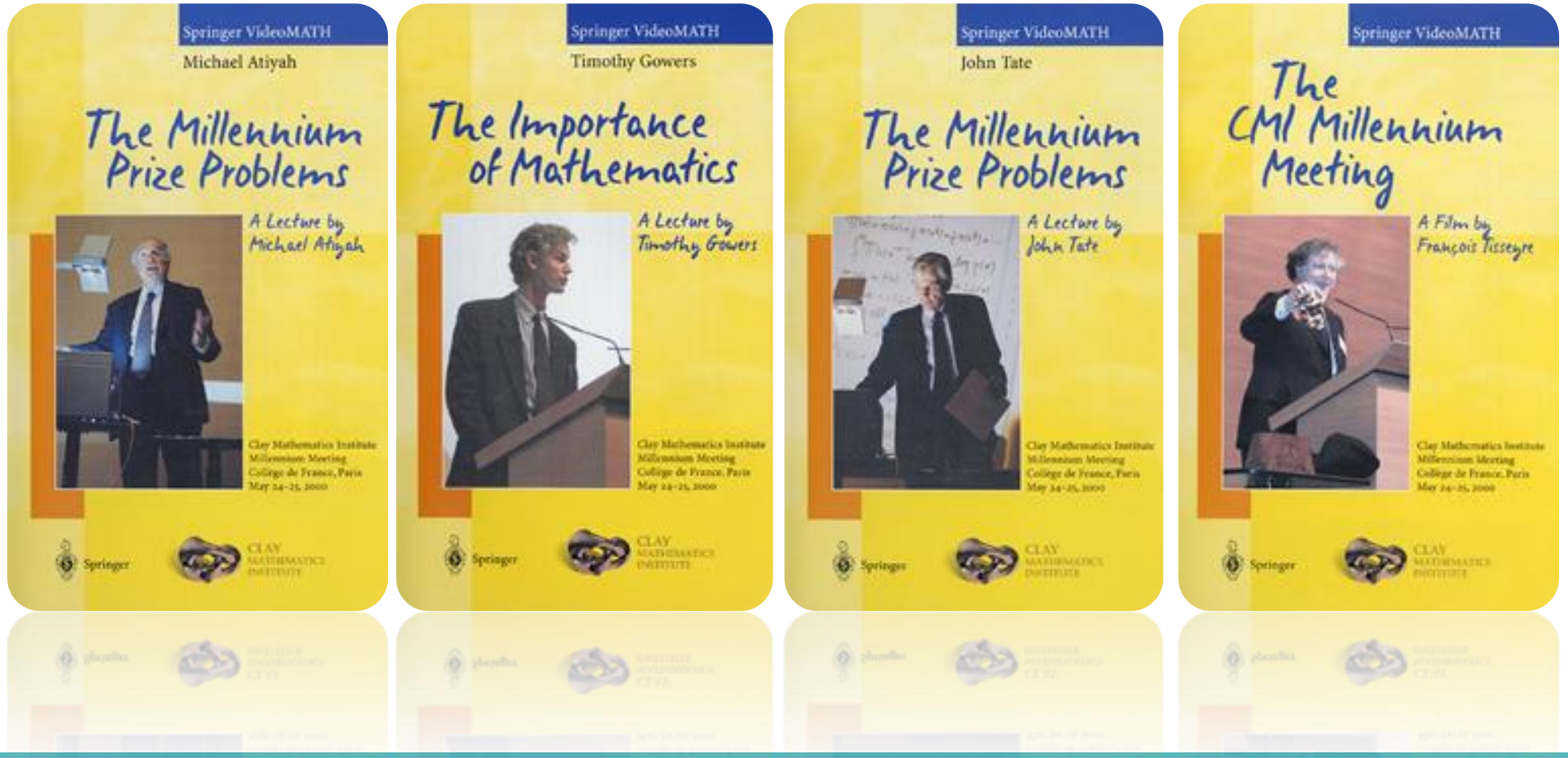
- P ⊆ NP
  - A problem solvable in polynomial time is verifiable in polynomial time as well

- Any NP problem can be solved in (deterministically) exponential time?
  - Yes

- Any NP problem can be solved in (deterministically) polynomial time?
  - Open problem

US$1,000,000 Per Problem

- http://www.claymath.org/millennium-problems



Millennium Problems

- Yang–Mills and Mass Gap
- Riemann Hypothesis
- P vs NP Problem
- Navier–Stokes Equation
- Hodge Conjecture
- Poincaré Conjecture (solved by Grigori Perelman)
- Birch and Swinnerton-Dyer Conjecture

Grigori Perelman
Fields Medal (2006), declined
Millennium Prize (2010), declined

Vinay Deolalikar

- Aug 2010 claimed a proof of P is not equal to NP.

If P = NP

- problems that are verifiable → solvable

- public-key cryptography will be broken

"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss..." – Scott Aaronson, MIT

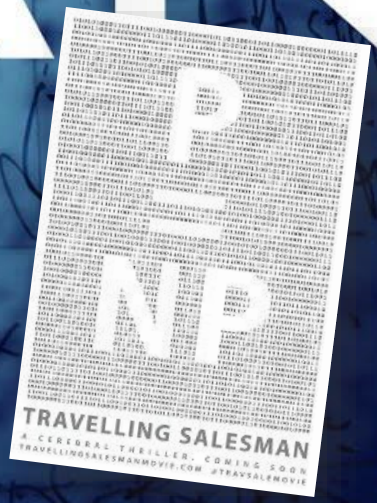
Widespread belief in P ≠ NP

TRAVELLING SALESMAN

Travelling Salesman (2012)

A movie about P = NP

Best Feature Film in Silicon Valley Film Festival 2012



NP, NP-Complete, NP-Hard

NP-Hardness

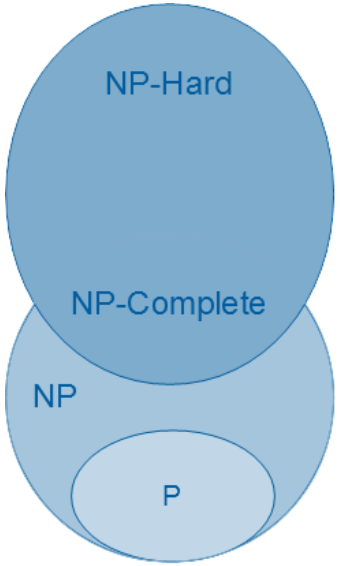
- A problem is NP-hard if it is as least as hard as all NP problems.
- In other words, a problem *X* is NP-hard if the following condition holds:
  - If *X* can be solved in (deterministic) polynomial time, then all NP problems can be solved in (deterministic) polynomial time.

NP-Completeness (NPC)

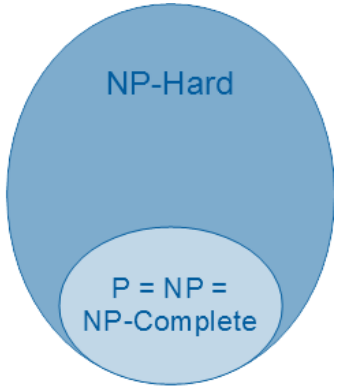
- A problem is NP-complete if
  - it is NP-hard and
  - *it is in NP*.
- In other words, an NP-complete problem is one of the “hardest” problems in the class NP.
- In other words, an NP-complete problem is a hardness representative problem of the class NP.
- Hardest in NP → solving one NPC can solve all NP problems (“complete”)
- It is wildly believed that NPC problems have no polynomial-time solution → good reference point to judge whether a problem is in P
  - We can decide if a problem is “too hard to solve” by showing it is as hard as an NPC problem
  - We then focus on designing approximate algorithms or solving special cases

Complexity Classes

- Class P: class of problems that can be solved in $O(n^k)$
- Class NP: class of problems that can be verified in $O(n^k)$
- Class NP-hard: class of problems that are “at least as hard as all NP problems”
- Class NP-complete: class of problems in both NP and NP-hard

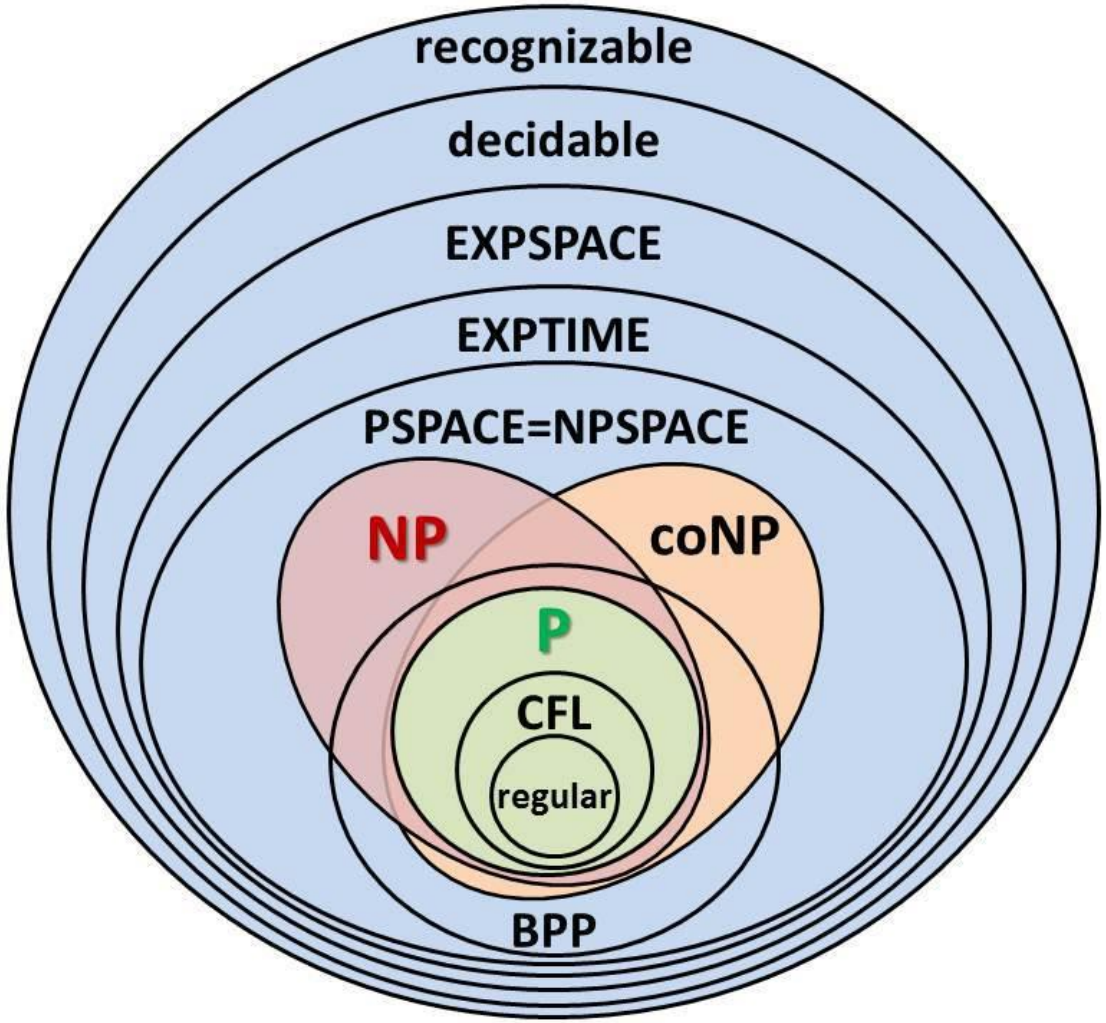


P ≠ NP



P = NP

More Complexity Classes



undecidable: no algorithm; e.g. halting problem

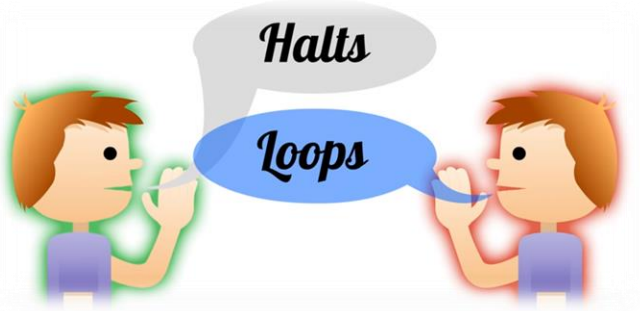
https://www.youtube.com/watch?v=wGLQiHXHWNk

THE HALTING PROBLEM

An Undecidable Problem – Halting Problem

- Halting problem is to determine whether a program p halts on input x
- Proof for undecidable via a counterexample
  - Suppose h can determine whether a program p halts on input x
  - $h(p, x)$ = return (p halts on input x)
  - Define g(p) = if h(p,p) is 0 then return 0 else HANG
  - → g(g) = if h(g,g) is 0 then return 0 else HANG
- Both cases contradict the assumption:

1. g halts on g: then h(g,g)=1, which would make g(g) hang
2. g does not halt on g: then h(g,g)=0, which would make g(g) halt



Complexity Classes

- Which one is in P?

Shortest Simple Path

Longest Simple Path

Euler Tour

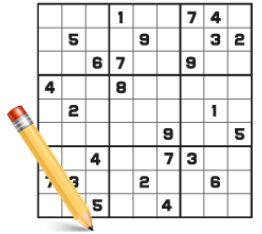
Hamitonian Cycle

LCS with 2 Input Sequences

LCS with Arbitrary Input Sequences

Degree-Constrained Spanning Tree

Minimal Spanning Tree



Candy Crush is NP-Hard

Bejeweled, Candy Crush and other match-three games are (NP-)Hard!

What is this all about?

This is an implementation of the reduction provided in the paper *Bejeweled, Candy Crush and other Match-Three Games are (NP)-Hard* which has been accepted for presentation at the *2014 IEEE Conference on Computational Intelligence and Games (CIG 2014)*. To find more about what NP-Hard means you can read this blog post or the corresponding page on Wikipedia.

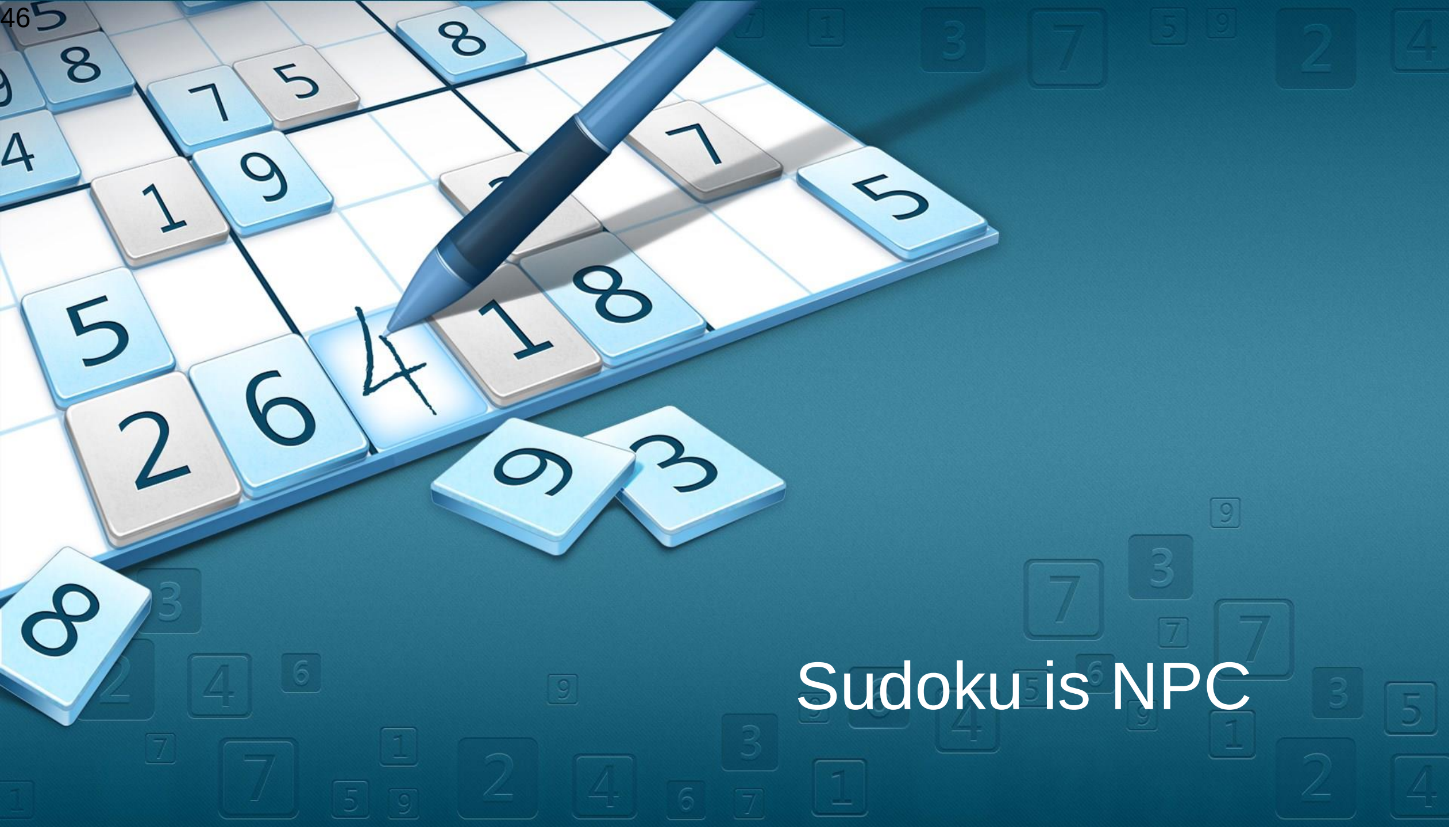
About the authors

We are an Italian group of three people: Luciano Gualà, Stefano Leucci, and Emanuele Natale. We had the weird idea to spend our weekends proving that Candy Crush Saga is NP-Hard. We also thought that it was nice to put online an implementation of our hardness reduction... so here it is!

Rules

Swap two adjacent gems in order to match three or more gems of the same kind. The matched gems will pop, and the gems above will fall. It is possibile to have chains of pops.

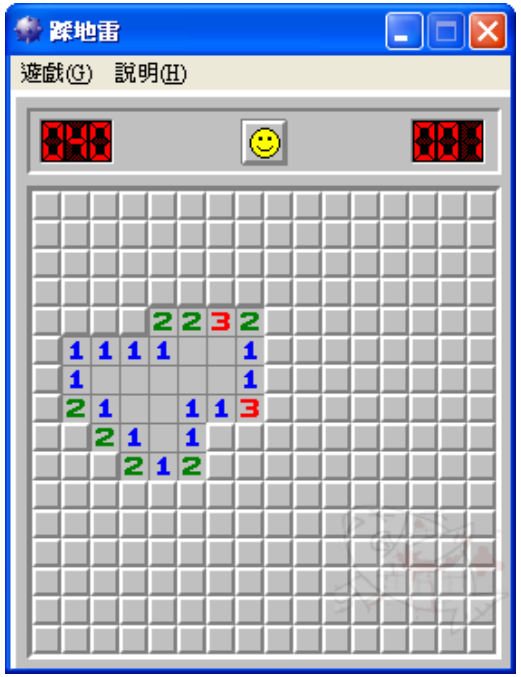
For a complete understanding of what's going on please read the paper on ArXiv.
In a nutshell (for those "tl;dr" folks): you can swap one or two gems on each choice wire from the top one to the bottom one, then you have to traverse the goal wire to reach the goal gem. Popping a wire means setting the corresponing variable to true.

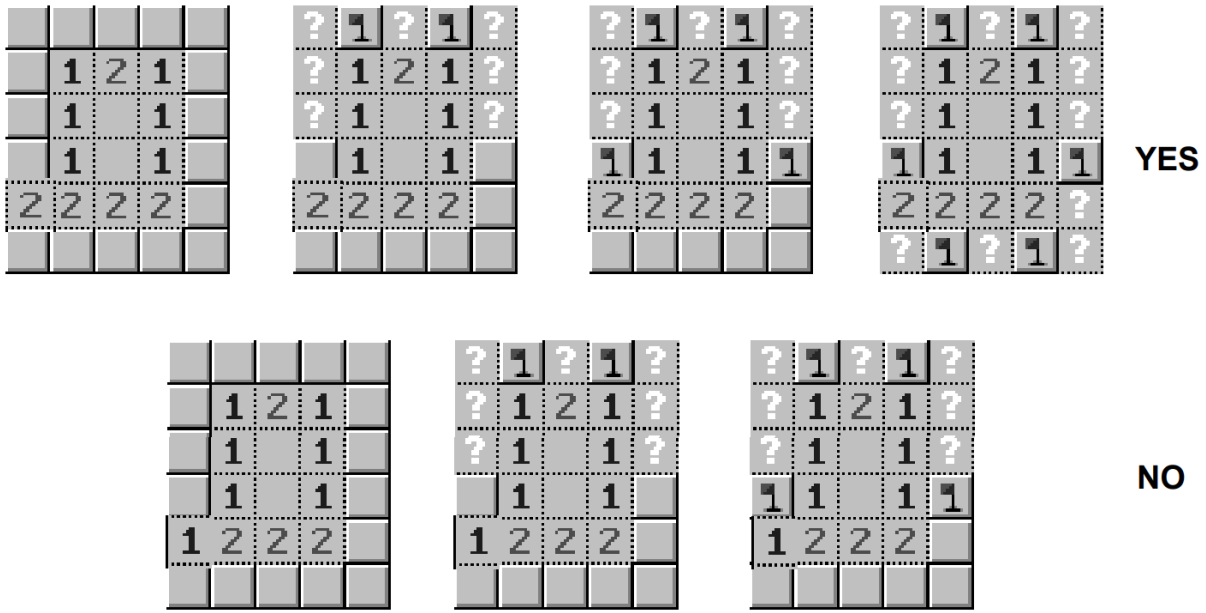


Sudoku is NPC

Minesweeper Consistency is NPC

- Minesweeper Consistency: Given a state of what purports to be a Minesweeper games, is it logically consistent?





Polynomial-Time Reduction

Textbook Chapter 34.3 – NP-completeness and reducibility

First NP-Complete Problem – SAT (Satisfiability)

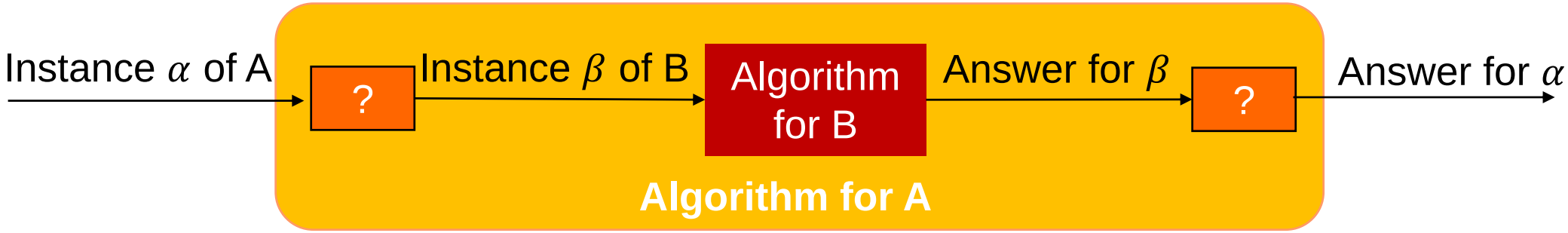
- Input: a Boolean formula with variables
- Output: whether there is a truth assignment for the variables that satisfies the input Boolean formula

$$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y)$$

- Stephan A. Cook [FOCS 1971] proved that
  - SAT can be solved in non-deterministic polynomial time → SAT ∈ NP
  - If SAT can be solved in deterministic polynomial time, then so can any NP problems → SAT ∈ NP-hard

Reduction

- Problem A can be reduced (in polynomial time) to Problem B
  = Problem B can be reduced (in polynomial time) from Problem A
  - We can find an algorithm that solves Problem B to help solve Problem A

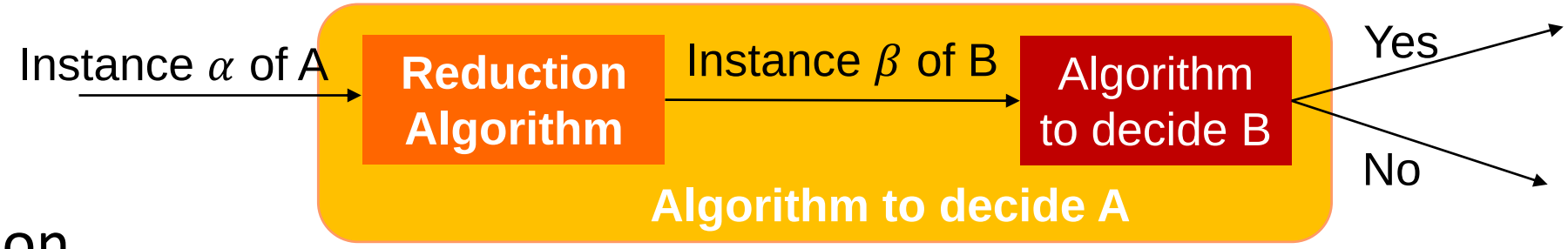


What is the complexity of Algorithm for A?

  - If problem B has a polynomial-time algorithm, then so does problem A
- Practice: design a MULTIPLY() function by ADD(), DIVIDE(), and SQUARE()

Reduction

- A reduction is an algorithm for **transforming a problem instance into another**



- Definition
  - Reduction from A to B implies A is not harder than B
  - $A \leq_p B$ if A can be reduced to B in polynomial time
- Applications
  - Designing algorithms: given algorithm for B, we can also solve A
  - Classifying problems: establish relative difficulty between A and B
  - **Proving limits: if A is hard, then so is B**

This is why we need it for proving NP-completeness!

Questions

- If *A* is an NP-hard problem and *B* can be reduced from *A*, then *B* is an NP-hard problem?
- If *A* is an NP-complete problem and *B* can be reduced from *A*, then *B* is an NP-complete problem?
- If *A* is an NP-complete problem and *B* can be reduced from *A*, then *B* is an NP-hard problem?

Problem Difficulty

- Q: Which one is harder?

KNAPSACK: Given a set $\{a_1, \dots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.

Polynomial-time reducible?

PARTITION: Given a set of n non-negative integers $\{a_1, \dots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

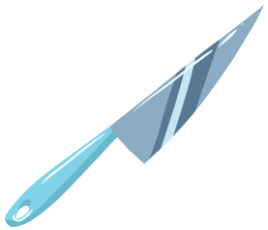
Polynomial-time reducible?

- A: They have equal difficulty.
- Proof:
  - PARTITION $\leq_p$ KNAPSACK
  - KNAPSACK $\leq_p$ PARTITION

Polynomial Time Reduction

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.

Polynomial-time reducible?



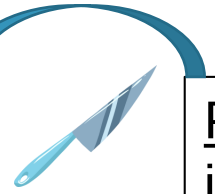
PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

Polynomial-time reducible?

- PARTITION $\leq_p$ KNAPSACK
  - If we can solve KNAPSACK, how can we use that to solve PARTITION?
- KNAPSACK $\leq_p$ PARTITION
  - If we can solve PARTITION, how can we use that to solve KNAPSACK?

PARTITION $\leq_p$ KNAPSACK

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.



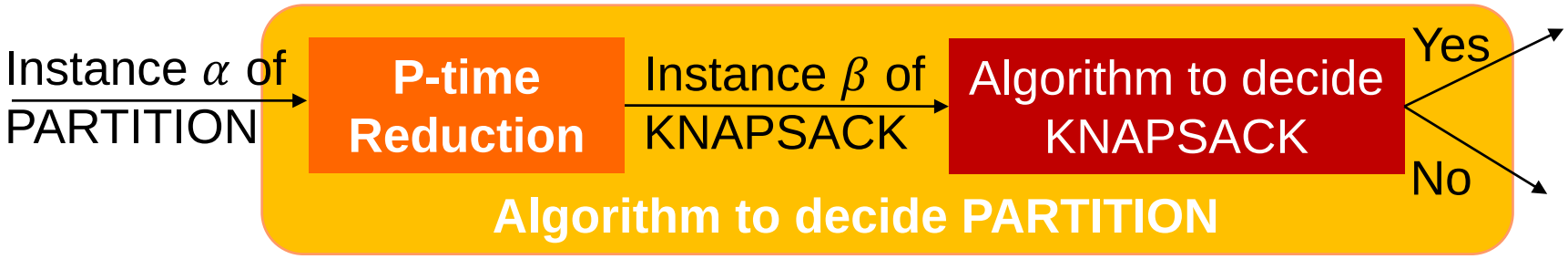
PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction
  - Set $K = \frac{1}{2} \sum_{i=1}^{n} a_i$

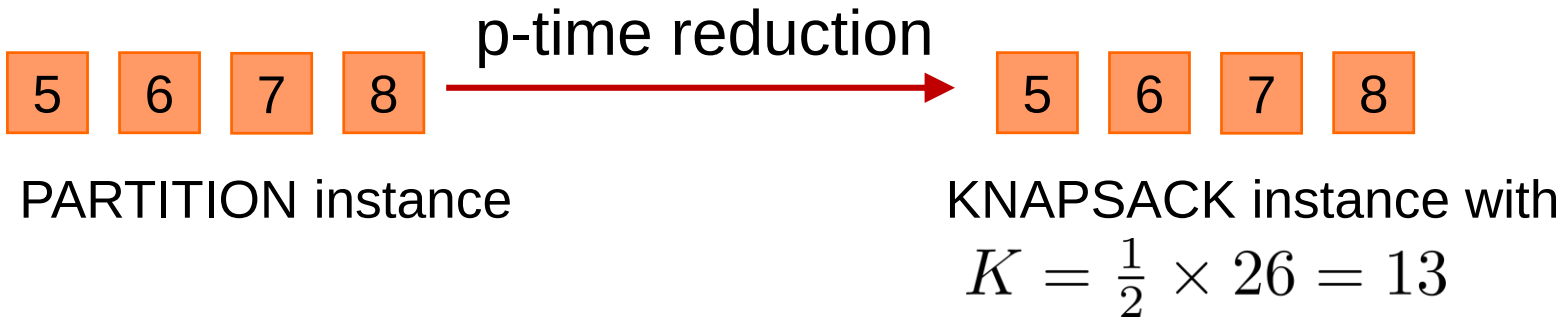
PARTITION instance

KNAPSACK instance with $K = \frac{1}{2} \times 26 = 13$

PARTITION $\leq_p$ KNAPSACK



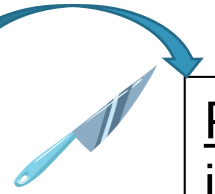
- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction
  - Set $K = \frac{1}{2}\sum_{i=1}^{n} a_i$



- Correctness proof: KNAPSACK returns yes if and only if an equal-size partition exists

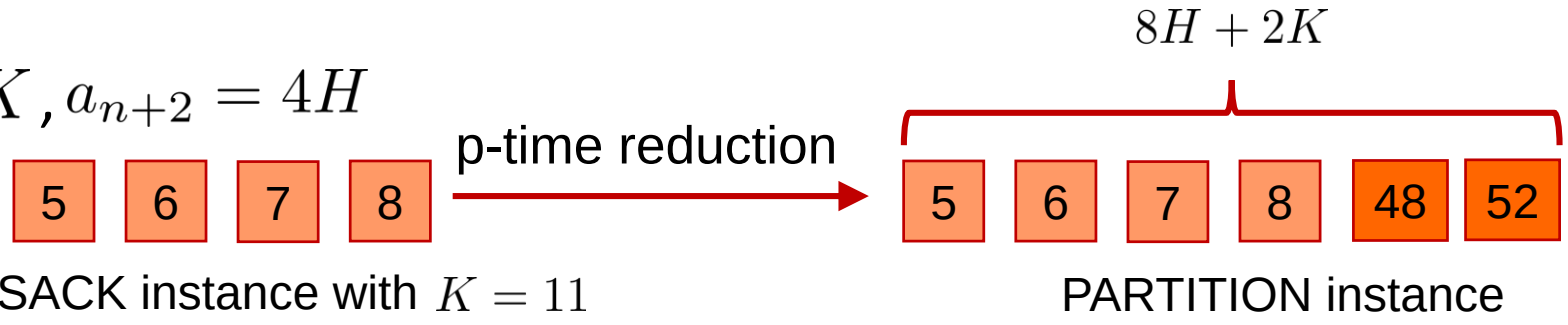
KNAPSACK $\leq_p$ PARTITION

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.



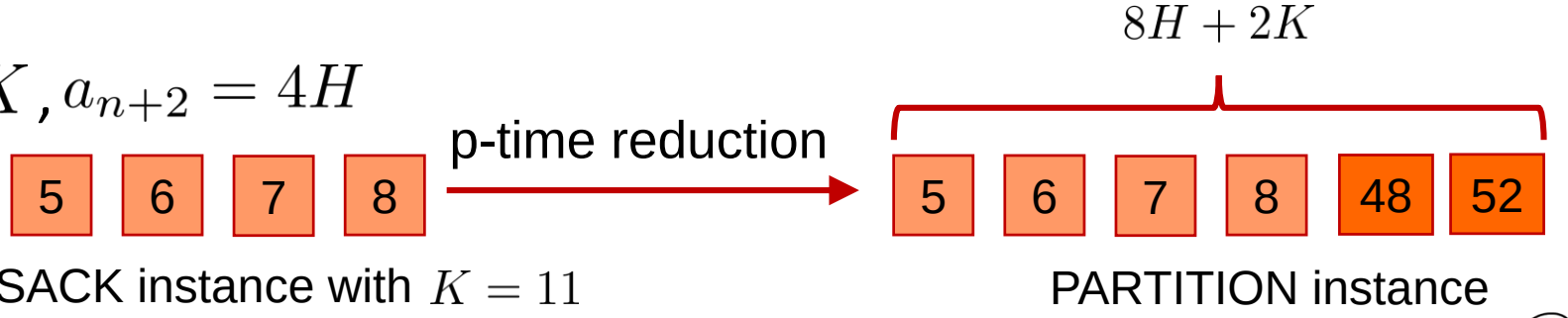
PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction
  - Set $H = \frac{1}{2} \sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$



KNAPSACK $\leq_p$ PARTITION

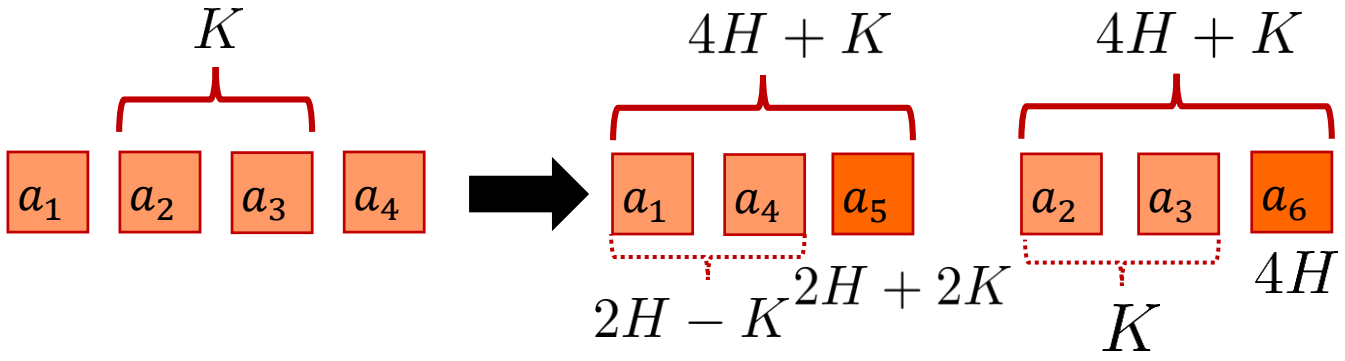
- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$



- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$

KNAPSACK $\leq_p$ PARTITION

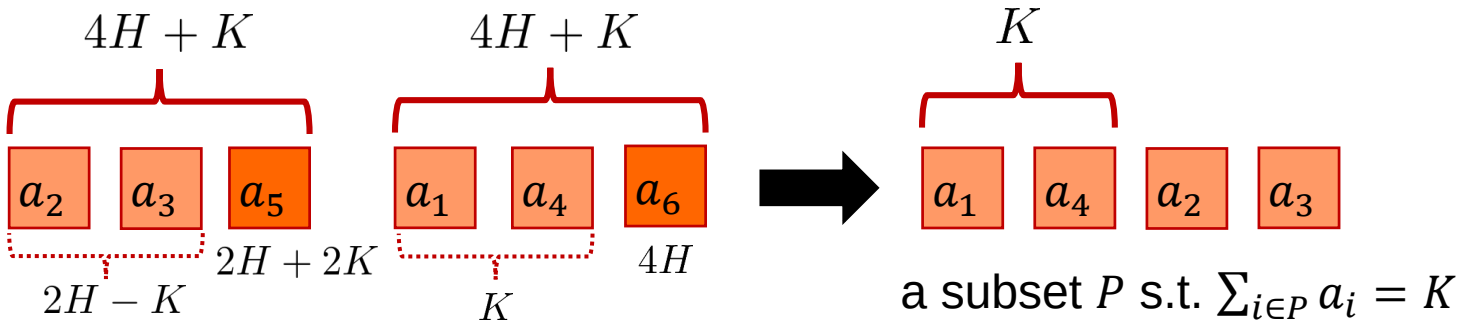
- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$
  - "if" direction



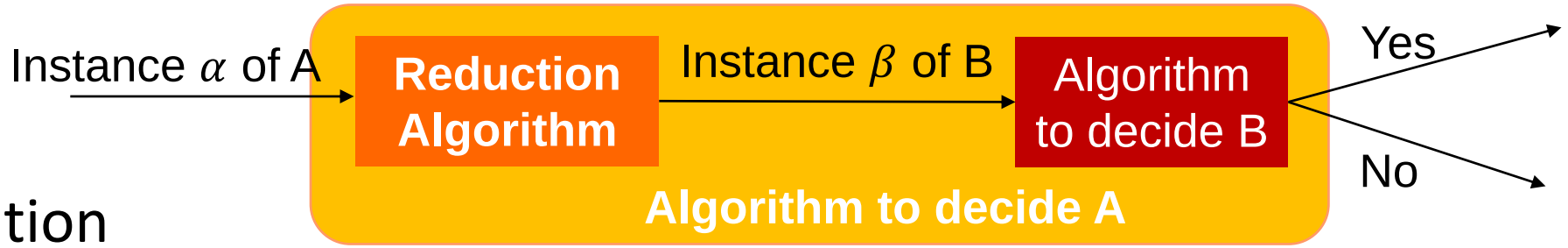
PARTITION returns yes!

KNAPSACK $\leq_p$ PARTITION

- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$
  - "only if" direction
    - Because $\sum_{i=1}^{n+2} a_i = 8H + 2K$, if PARTITION returns yes, each set has $4H + K$
    - $\{a_1, \ldots, a_n\}$ must be divided into $2H - K$ and K



Reduction for Proving Limits



- Definition
  - Reduction from A to B implies A is not harder than B
  - $A \leq_p B$ if A can be reduced to B in polynomial time
- NP-completeness proofs
  - Goal: prove that B is NP-hard
  - Known: A is NP-complete/NP-hard
  - Approach: construct a polynomial-time reduction algorithm to convert α to β
  - Correctness: if we can solve B, then A can be solved → $A \leq_p B$
  - B is no easier than A → A is NP-hard, so B is NP-hard

If the reduction is not p-time, does this argument hold?

Proving NP-Completeness

Formal Language Framework

- Focus on decision problems
- A language L over $\sum$ is any set of strings made up of symbols from $\sum$
- Every language L over $\sum$ is a subset of $\sum^*$

$$\sum^* = \{\epsilon; 0; 1; 10; 11; 101; 111; \cdots\}$$

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them.

- An algorithm A accepts a string $x \in \{0,1\}^*$ if $A(x) = 1$
- The language accepted by an algorithm A is the set of strings

$$L = \{x \in \{0,1\}^* : A(x) = 1\}$$

- An algorithm A rejects a string x if $A(x) = 0$

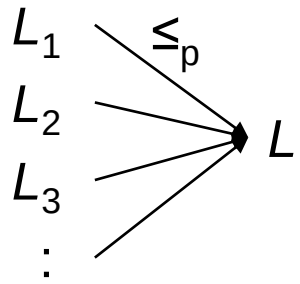
Proving NP-Completeness

- NP-Complete (NPC): class of decision problems in both NP and NP-hard
- In other words, a decision problem L is NP-complete if

  1. $L \in$ NP

  2. $L \in$ NP-hard (that is, $L' \leq_p L$ for every $L' \in$ NP)

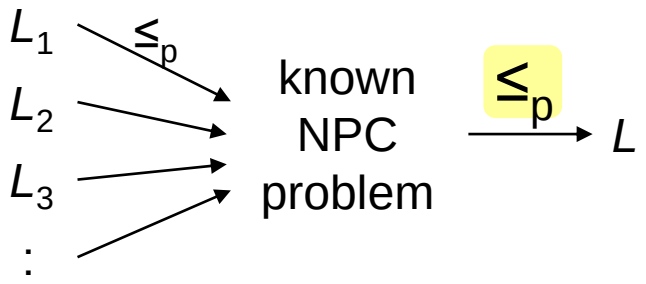
How to prove *L* is NP-hard ?



all NP problems

held by definition

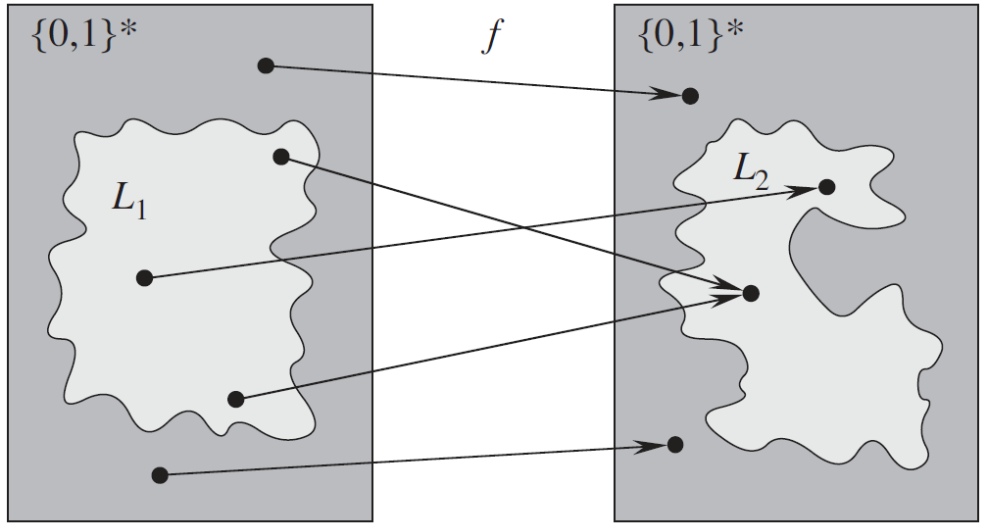
Goal: prove polynomial-time reduction

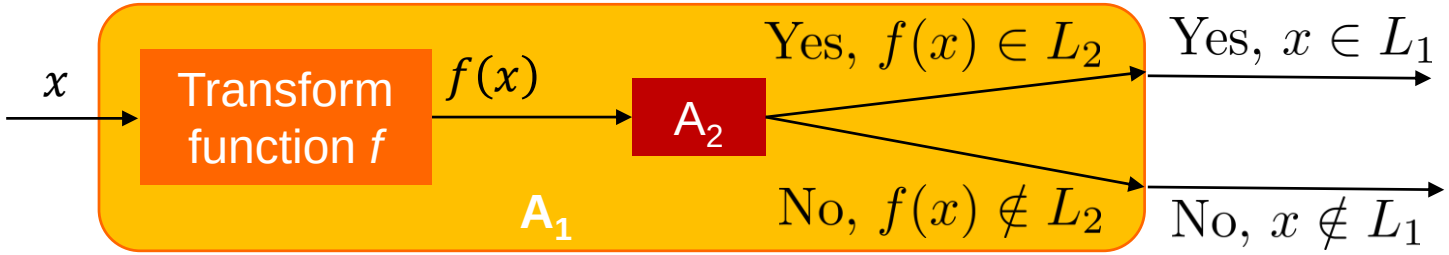


all NP problems

Polynomial-Time Reducible

- If $L_1, L_2 \subset \{0,1\}^*$ are languages s.t. $L_1 \leq_p L_2$, then L2 ∈ P implies L1 ∈ P.



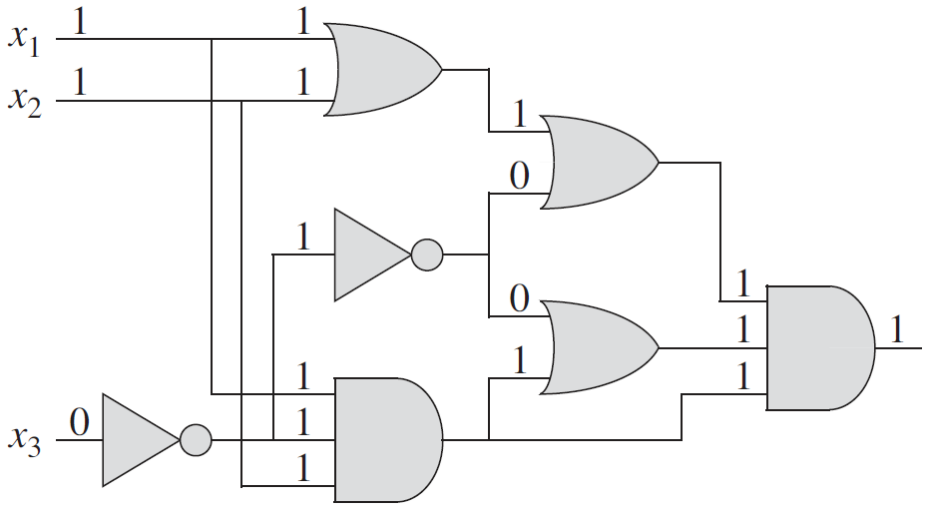


P v.s. NP

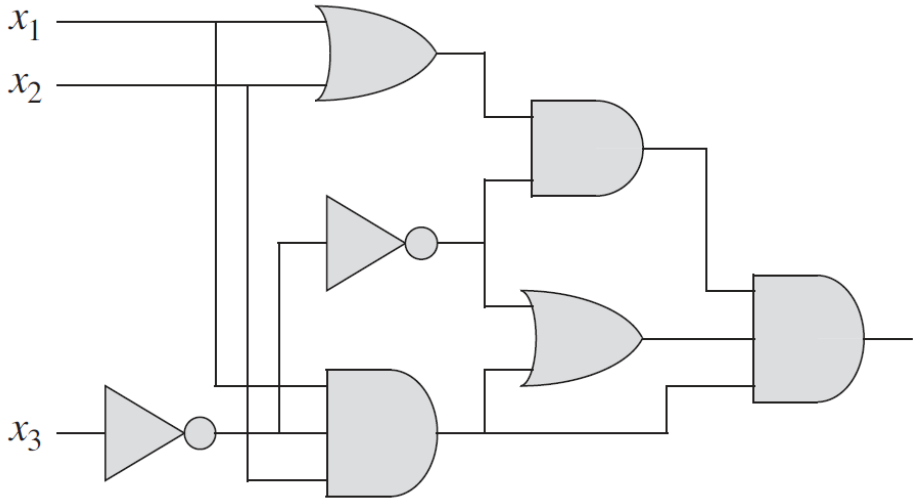
- If one proves that SAT can be solved by a polynomial-time algorithm, then NP = P.
- If somebody proves that SAT cannot be solved by any polynomial-time algorithm, then NP ≠ P.

Circuit Satisfiability Problem

- Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
  - Satisfiable: there exists an assignment s.t. outputs = 1



Satisfiable



Unsatisfiable

CIRCUIT-SAT

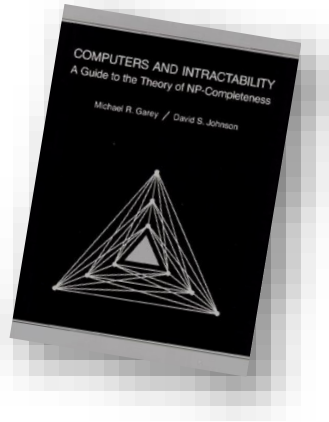
CIRCUIT-SAT = {<C>: C is a satisfiable Boolean combinational circuit}

- CIRCUIT-SAT can be solved in non-deterministic polynomial time

  → ∈ NP
- If CIRCUIT-SAT can be solved in deterministic polynomial time, then so can any NP problems

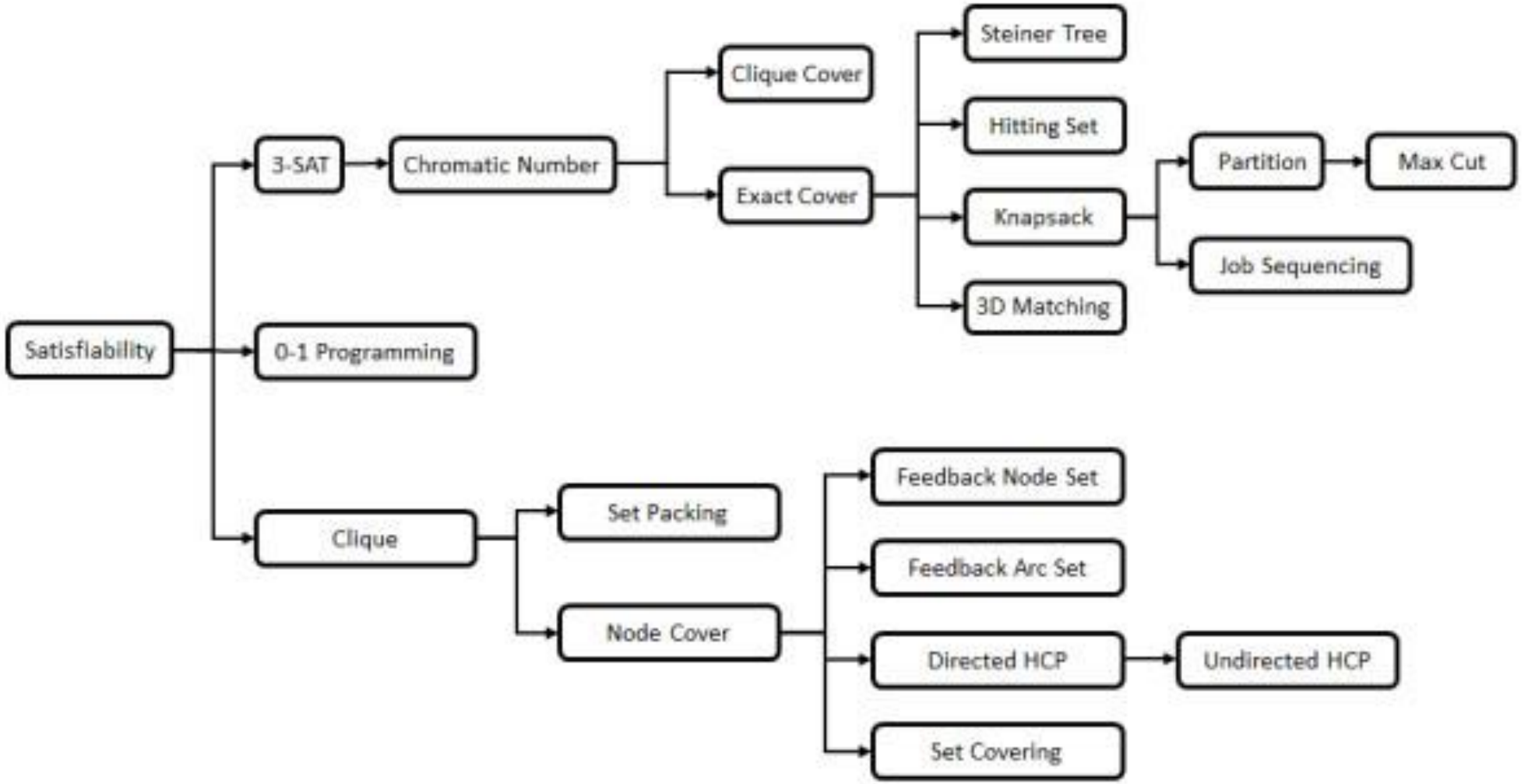
  → ∈ NP-hard
- (proof in textbook 34.3)
- CIRCUIT-SAT is NP-complete

Karp's NP-Complete Problems



1. CNF-SAT
2. 0-1 INTEGER PROGRAMMING
3. CLIQUE
4. SET PACKING
5. VERTEX COVER
6. SET COVERING
7. FEEDBACK ARC SET
8. FEEDBACK NODE SET
9. DIRECTED HAMILTONIAN CIRCUIT
10. UNDIRECTED HAMILTONIAN CIRCUIT
11. 3-SAT
12. CHROMATIC NUMBER
13. CLIQUE COVER
14. EXACT COVER
15. 3-dimensional MATCHING
16. STEINER TREE
17. HITTING SET
18. KNAPSACK
19. JOB SEQUENCING
20. PARTITION
21. MAX-CUT

Karp's NP-Complete Problems



Formula Satisfiability Problem (SAT)

- Given a Boolean formula Φ with variables, is there a variable assignment satisfying Φ

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

  - $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if)
  - Satisfiable: Φ is evaluated to 1

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

$$\begin{aligned}
\phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\
&= (1 \vee \neg((1 \leftrightarrow 1) \vee 1)) \wedge 1 \\
&= (1 \vee \neg(1 \vee 1)) \wedge 1 \\
&= (1 \vee 0) \wedge 1 \\
&= 1 \wedge 1 \\
&= 1
\end{aligned}$$

SAT

SAT = {Φ | Φ is a Boolean formula with a satisfying assignment }

- Is SAT ∈ NP-Complete?
- To prove that SAT is NP-Complete, we show that
  - SAT ∈ NP
  - SAT ∈ NP-hard (CIRCUIT-SAT $\leq_p$ SAT)

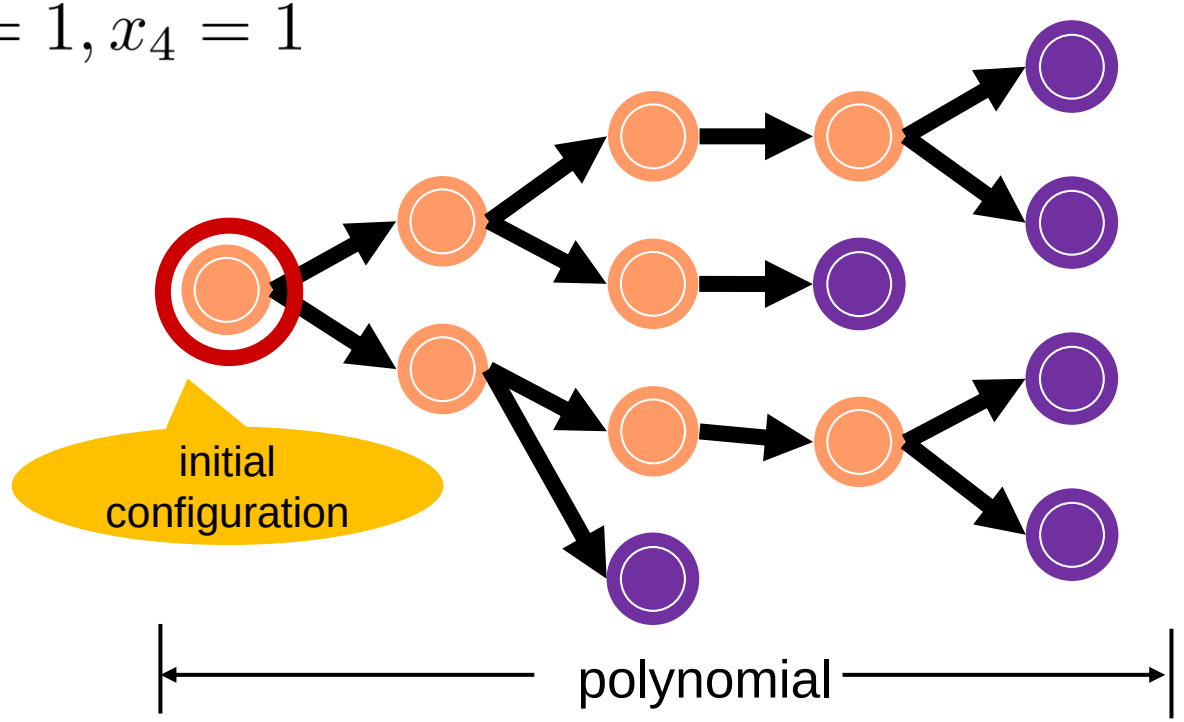
1) CIRCUIT-SAT is a known NPC problem
2) Construct a reduction *f* transforming every CIRCUIT-SAT instance to an SAT instance
3) Prove that x ∈ CIRCUIT-SAT iff *f*(x) ∈ SAT
4) Prove that *f* is a polynomial time transformation

SAT ∈ NP

- **Polynomial-time verification**: replaces each variable in the formula with the corresponding value in the certificate and then evaluates the expression

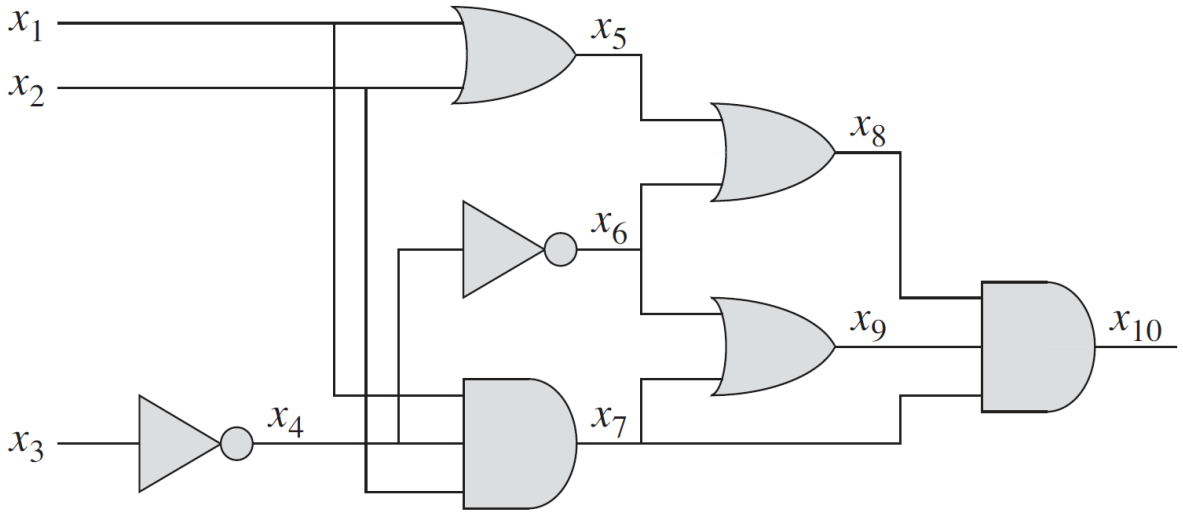
$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

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

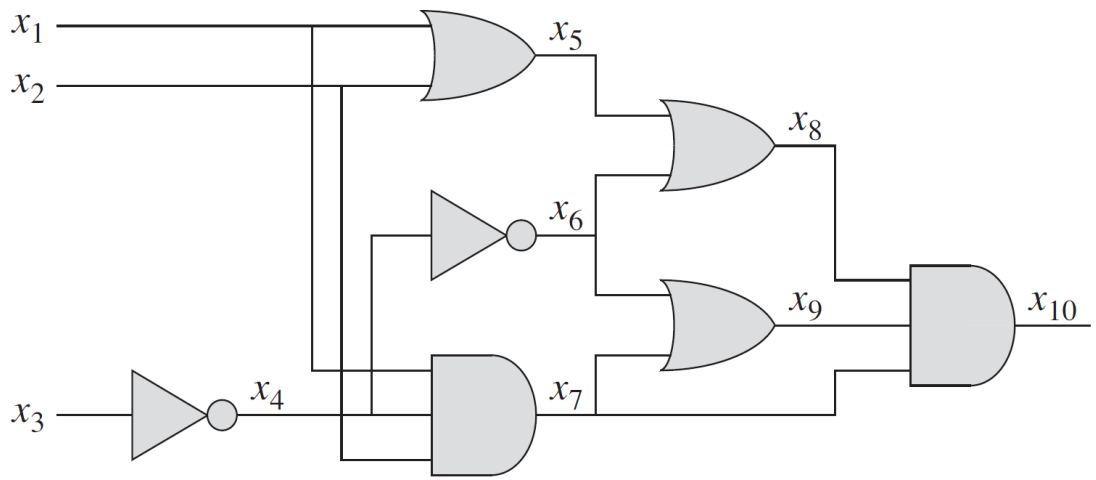


SAT ∈ NP-Hard

1) CIRCUIT-SAT is a known NPC problem
2) Construct a reduction *f* transforming every CIRCUIT-SAT instance to an SAT instance
   - Assign a variable to each **wire** in circuit C
   - Represent the operation of each gate using a formula, e.g.
   - Φ = AND the **output variable** and the **operations of all gates** $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$



SAT ∈ NP-Hard



$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

- Prove that x ∈ CIRCUIT-SAT ↔ f(x) ∈ SAT
  - x ∈ CIRCUIT-SAT → f(x) ∈ SAT
  - f(x) ∈ SAT → x ∈ CIRCUIT-SAT
- f is a polynomial time transformation

CIRCUIT-SAT $\leq_p$ SAT → SAT ∈ NP-hard

3-CNF-SAT Problem

- 3-CNF-SAT: Satisfiability of Boolean formulas in 3-*conjunctive normal form* (3-CNF)

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

  - 3-CNF = AND of clauses, each of which is the OR of exactly 3 distinct literals
  - A literal is an occurrence of a variable or its negation, e.g., x_1 or $\neg x_1$

$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$ → satisfiable

3-CNF-SAT

3-CNF-SAT = {Φ | Φ is a Boolean formula in 3-conjunctive normal form (3-CNF) with a satisfying assignment }

- Is 3-CNF-SAT ∈ NP-Complete?
- To prove that 3-CNF-SAT is NP-Complete, we show that
  - 3-CNF-SAT ∈ NP
  - 3-CNF-SAT ∈ NP-hard (SAT $\leq_p$ 3-CNF-SAT)

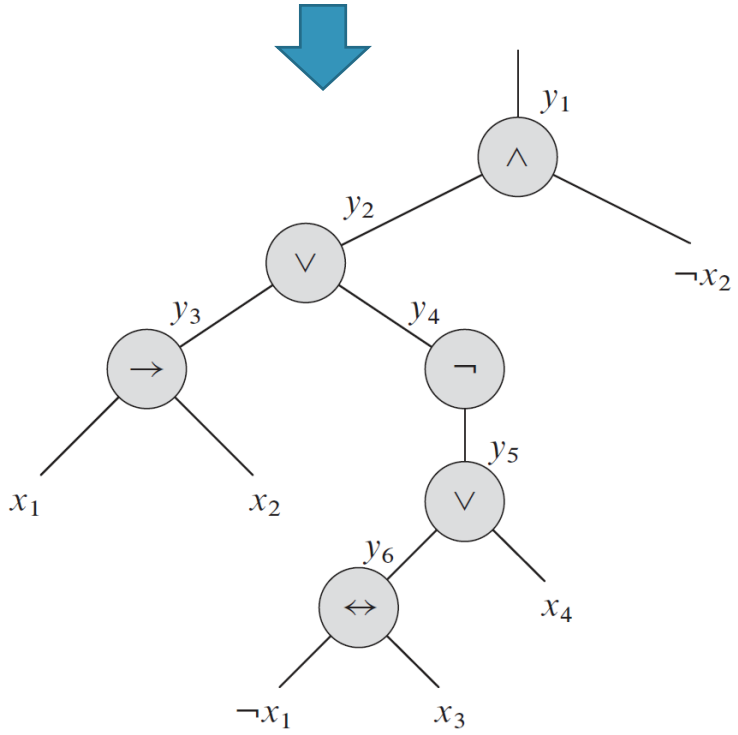
1) SAT is a known NPC problem
2) Construct a reduction *f* transforming every SAT instance to an 3-CNF-SAT instance
3) Prove that x ∈ SAT iff *f*(x) ∈ 3-CNF-SAT
4) Prove that *f* is a polynomial time transformation

We focus on the reduction construction from now on, but remember that a full proof requires showing that all other conditions are true as well

SAT $\leq_p$ 3-CNF-SAT

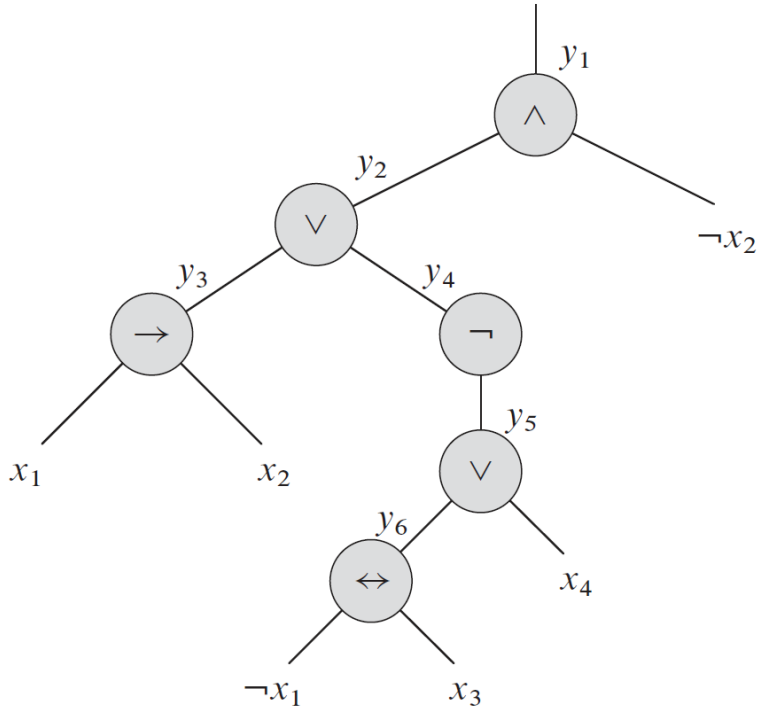
a) Construct a binary parser tree for an input formula Φ and introduce a variable y_i for the output of each internal node

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$



SAT $\leq_p$ 3-CNF-SAT

b)Rewrite Φ as the AND of the root variable and clauses describing the operation of each node



$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \wedge x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \wedge x_3))\end{aligned}$$

SAT $\leq_p$ 3-CNF-SAT

c) Convert each clause Φ_i' to CNF

- Construct a truth table for each clause Φ_i'
- Construct the disjunctive normal form for $\neg\Phi_i'$
- Apply DeMorgan's Law to get the CNF formula Φ_i''

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \wedge x_2)) \wedge (y_4 \leftrightarrow \neg y_5) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \wedge x_3))$$

y_1	y_2	y_2	Φ_1'	$\neg\Phi_1'$
1	1	1	0	1
1	1	0	1	0
1	0	1	0	1
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	1	0
0	0	0	1	0

$$\neg\phi_1' = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

$$\phi_1' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$

SAT $\leq_p$ 3-CNF-SAT

d)Construct Φ’’’ in which each clause C_i exactly 3 distinct literals

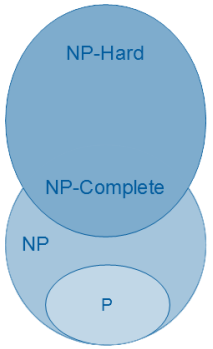
- 3 distinct literals: $C_i = l_1 \vee l_2 \vee l_3$
- 2 distinct literals: $C_i = l_1 \vee l_2$

  $C_i = l_1 \vee l_2 = (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$
- 1 literal only: $C_i = l$

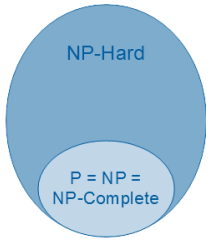
  $C_i = l = (l \vee p \vee q) \wedge (l \vee \neg p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee \neg q)$

- Φ’’’ is satisfiable iff Φ is satisfiable
- All transformation can be done in polynomial time
- → 3-CNF-SAT is NP-Complete

Concluding Remarks

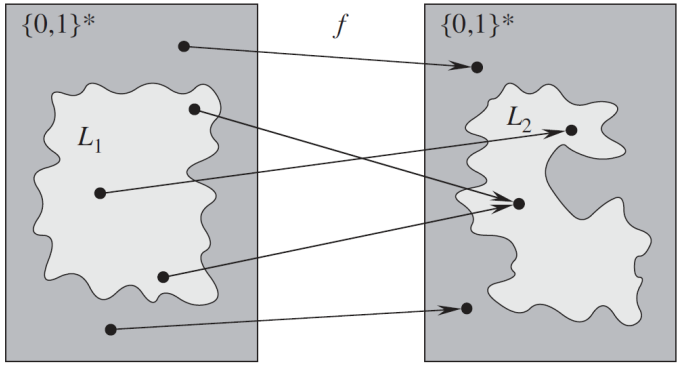


P ≠ NP



P = NP

- Proving NP-Completeness: L ∈ NPC iff L ∈ NP and L ∈ NP-hard
- Step-by-step approach for proving L in NPC:
  - Prove L ∈ NP
  - Prove L ∈ NP-hard
    - Select a known NPC problem C
    - Construct a reduction f transforming every instance of C to an instance of L
    - Prove that $x \in C \iff f(x) \in C, \forall x \in \{0,1\}^*$
    - Prove that f is a polynomial time transformation L ∈ NP



Question?

Important announcement will be sent to @ntu.edu.tw mailbox & post to the course website

Course Website: http://ada.miulab.tw
Email: ada-ta@csie.ntu.edu.tw