COMP 382: Reasoning about Algorithms

P, NP, NP-Hardness, NP-Completeness

Prof. Maryam Aliakbarpour

co-instructors: Prof. Anjum Chida & Prof. Konstantinos Mamouras

November 17, 2025

Today's Lecture

1. **What Is NP-Hardness?**

Reading:

- Chapter 19 of [Roughgarden, 2022]

Content adapted from the same reference.

What Is NP-Hardness?

The Core Problem: Selection Bias

- Introductory algorithm books suffer from **selection bias**.
- They focus on problems with clever, fast algorithms (e.g., sorting, shortest paths, MSTs).

The Core Problem: Selection Bias

- Introductory algorithm books suffer from **selection bias**.
- They focus on problems with clever, fast algorithms (e.g., sorting, shortest paths, MSTs).
- Many important problems have **no fast algorithms known**.
- These problems are deemed "intractable."

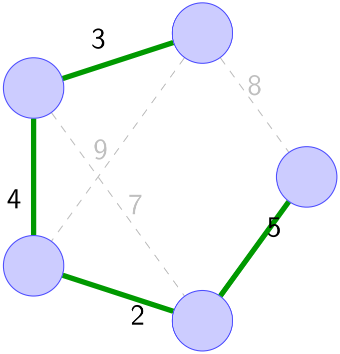
MST vs TSP

An Algorithmic Mystery

“Easy”: Minimum Spanning Tree (MST)

Problem: Find a spanning tree (a subset of edges that connects all vertices without cycles) of minimum total edge cost.

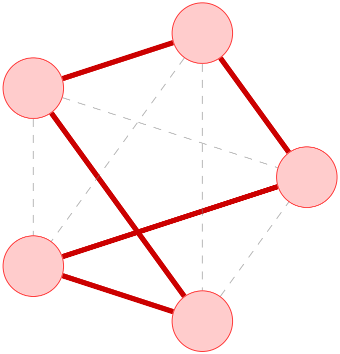
- Solvable by blazingly fast algorithms:
  - Prim’s
  - Kruskal’s
- **Running Time:** $O((m+n)\log n)$.
- This is a **computationally easy** problem.



"Hard": Traveling Salesman Problem (TSP)

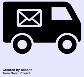
Problem: Find a tour (a cycle visiting every vertex exactly once) of minimum total edge cost.

- The definition looks deceptively similar to MST.
- No fast algorithm is known.
- Exhaustive search is $O(n!)$, which is **infeasible**.
- This is **computationally hard**.



Why TSP Matters: Real-World Intractability

TSP is a powerful template for many practical optimization problems.

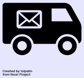


Mail Deliveries

finding the shortest route for deliveries.

Why TSP Matters: Real-World Intractability

TSP is a powerful template for many practical optimization problems.



Mail Deliveries

finding the shortest route for deliveries.

Genome Sequencing

Finding the most plausible ordering of overlapping gene fragments.

Why TSP Matters: Real-World Intractability

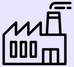
TSP is a powerful template for many practical optimization problems.

Mail Deliveries

finding the shortest route for deliveries.

Genome Sequencing

Finding the most plausible ordering of overlapping gene fragments.



Factory Assembly

Minimizing setup costs between assembling different car models.

Defining “Easy” and “Hard” Problems

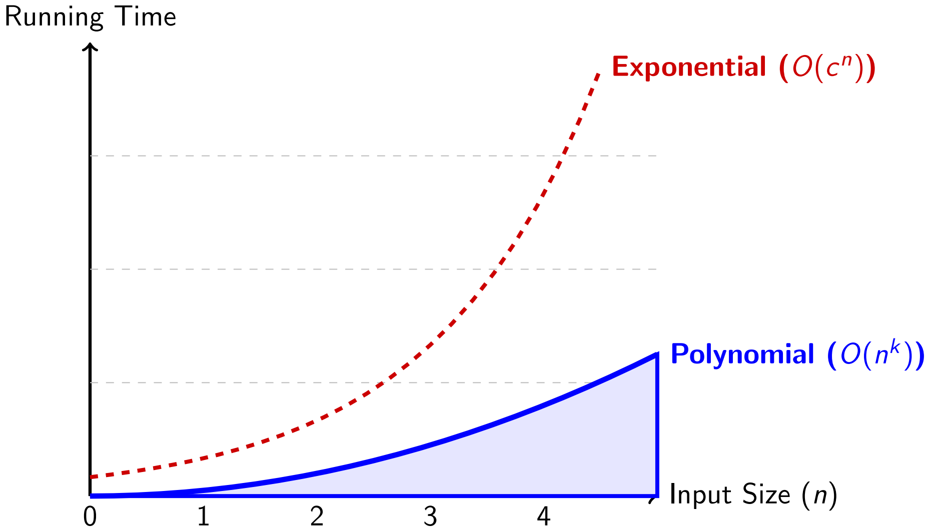
Or, a gentle introduction to complexity classes

Easy and Hard Problems

An oversimplified view:

- **Easy:** can be solved with a polynomial-time algorithm.
- **Hard:** require exponential time in the worst case.

Polynomial vs. Exponential Time

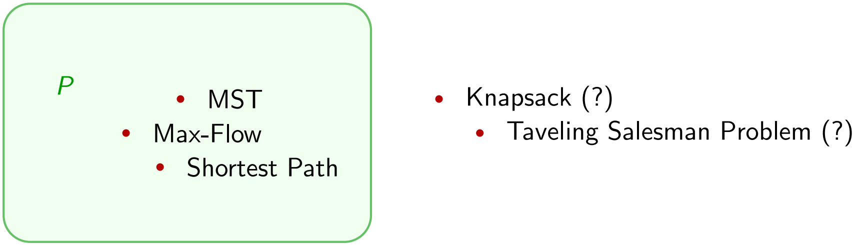


P: Polynomial Time Solvable Problems

- Complexity theory classifies problems based on their *inherent difficulty*;
- Algorithms can be fast or slow, clever or naive, but our statements about the *problem itself*.
- A problem is polynomial time solvable if there is an algorithm that correctly solves it in $O(n^k)$ time, for some constant k, where n is the input length.
- still polynomial even $k = 10^{10}$.
- This is worst-case running time. (maximum running time over all possible inputs of size n)
- **P**: Problems solvable in **P**olynomial time (easy to **solve**).

P: Examples

- Typical examples:
  - Shortest paths (without nasty conditions like negative cycles).
  - Minimum spanning tree, maximum flow, bipartite matching, etc.
- Non-example: the standard dynamic programming for knapsack runs in $\Theta(nW)$ time, where W is the capacity; since the input size is only $\log W$, this is actually **pseudopolynomial**, not polynomial, in the input length.



Decision Problems: The Formal Foundation

- Complexity classes are formally defined using problems that yield a simple **YES or NO** answer.
- This restriction is necessary to create a clean mathematical framework for verification.

Decision Problems: The Formal Foundation

- Complexity classes are formally defined using problems that yield a simple **YES or NO** answer.
- This restriction is necessary to create a clean mathematical framework for verification.
- Optimization problems (finding the minimum or the maximum) are closely connected to their related decision problems (is the minimum $\leq k$?).

Decision Problems: The Formal Foundation

- Complexity classes are formally defined using problems that yield a simple **YES or NO** answer.
- This restriction is necessary to create a clean mathematical framework for verification.
- Optimization problems (finding the minimum or the maximum) are closely connected to their related decision problems (is the minimum $\leq k$?).

Decision

- **MST (Decision):** Is there a spanning tree with total cost $\leq k$?
- **TSP (Decision):** Is there a tour with total cost $\leq k$?

Optimization

- **MST (Optimization):** Find the minimum cost spanning tree.
- **TSP (Optimization):** Find the shortest tour.

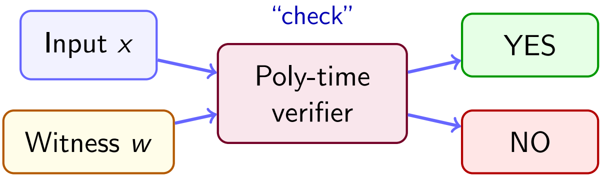
The Class NP

The Class NP

NP is the class of problems for which *solutions can be efficiently recognized*, even if we don't know how to find them efficiently.

A problem is in NP if:

- YES-instances have short **witnesses** (certificates) whose length is polynomial in the input size.
- We can verify a witness in polynomial time.



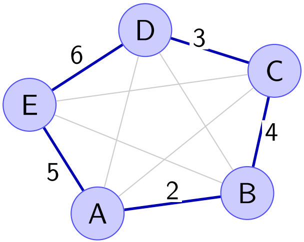
Decision Version of TSP and Its Witness

- **Input:** Complete graph $G = (V, E)$ with edge lengths d_{uv} and a budget k.
- **Question:** Is there a tour (Hamiltonian cycle) of total length $\leq k$?

Decision Version of TSP and Its Witness

- **Input:** Complete graph $G = (V, E)$ with edge lengths d_{uv} and a budget k.
- **Question:** Is there a tour (Hamiltonian cycle) of total length $\leq k$?
- **Example:**

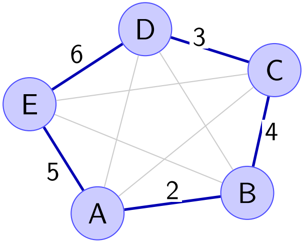
$$\text{witness: } A \to B \to C \to D \to E \to A\,, \qquad k = 25\,.$$



Decision Version of TSP and Its Witness

- **Input:** Complete graph $G = (V, E)$ with edge lengths d_{uv} and a budget k.
- **Question:** Is there a tour (Hamiltonian cycle) of total length $\leq k$?
- **Example:**

$$\text{witness: } A \to B \to C \to D \to E \to A\,, \qquad k = 25\,.$$



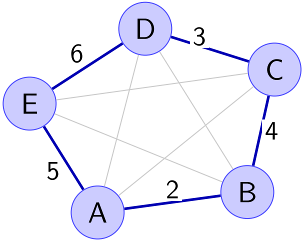
tour 1: $2 + 4 + 3 + 6 + 5 = 20$ Yes!

Solving TSP via Brute-Force Algorithm

1. Enumerate all possible tours (Hamiltonian cycles) on V.
2. For each tour C:
   - Check it visits every vertex exactly once.
   - Compute its total length $L(C)$.
   - If $L(C) \leq k$, **accept**.
3. If no tour passes the test, **reject**.

Solving TSP via Brute-Force Algorithm

1. Enumerate all possible tours (Hamiltonian cycles) on V.
2. For each tour C:
   - Check it visits every vertex exactly once.
   - Compute its total length $L(C)$.
   - If $L(C) \leq k$, **accept**.
3. If no tour passes the test, **reject**.

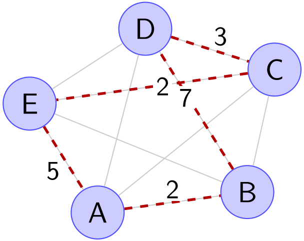


tour 1: $A \to B \to C \to D \to E \to A$

$2 + 4 + 3 + 6 + 5 = 20$

Solving TSP via Brute-Force Algorithm

1. Enumerate all possible tours (Hamiltonian cycles) on V.
2. For each tour C:
   - Check it visits every vertex exactly once.
   - Compute its total length $L(C)$.
   - If $L(C) \leq k$, **accept**.
3. If no tour passes the test, **reject**.

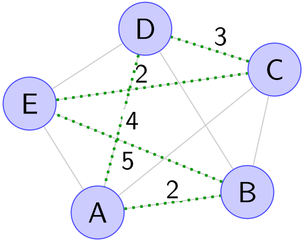


tour 2: $A \to B \to D \to C \to E \to A$

$2 + 7 + 3 + 2 + 5 = 19$ (better)

Solving TSP via Brute-Force Algorithm

1. Enumerate all possible tours (Hamiltonian cycles) on V.
2. For each tour C:
   - Check it visits every vertex exactly once.
   - Compute its total length $L(C)$.
   - If $L(C) \leq k$, **accept**.
3. If no tour passes the test, **reject**.

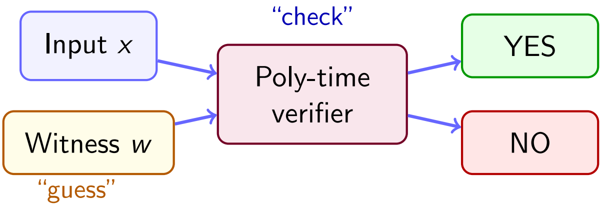


tour 3: $A \to B \to E \to C \to D \to A$

$2 + 5 + 2 + 3 + 5 = 17$ (best)

The Class NP as "Guess and Check"

- For problems in NP, we can always solve them by:
  1. Enumerating all candidate solutions (witnesses) of polynomial length. [guess a solution]
  2. Checking each one using the polynomial-time verifier.
- Number of candidates is typically exponential in input size $\Rightarrow$ exponential-time brute force.
- Vast majority of important natural problems (scheduling, routing, puzzles, many optimization problems) live in NP.



What Does “NP” Stand For?

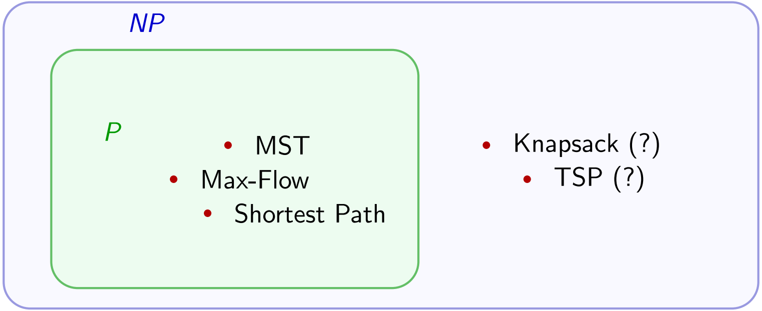
- Common wrong guess: “not polynomial”.
- Correct: **Nondeterministic Polynomial time**.

- Historically defined using *nondeterministic Turing machines*: machines that can “guess” a solution and then verify it in polynomial time.
- Modern viewpoint (equivalent and more intuitive for us): NP is the set of problems with polynomial-time verifiers and polynomial-length witnesses.

Is P $=$ NP?

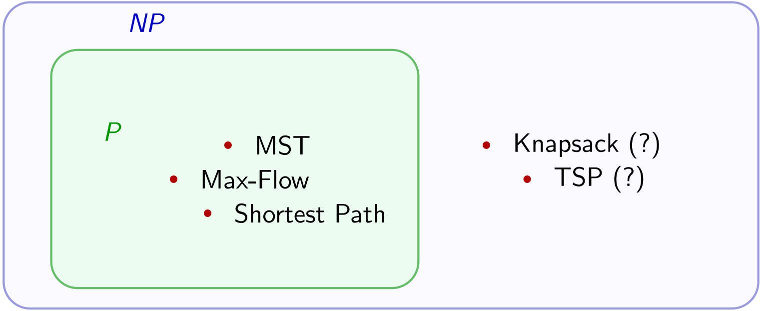
Is P = NP?

- We know that $\mathbf{P} \subseteq \mathbf{NP}$, e.g., MST $\in$ NP.



Is P = NP?

- We know that **P** $\subseteq$ **NP**, e.g., MST $\in$ NP.
- For many problems in NP, no polynomial-time algorithm is known, (e.g., TSP).



The P vs. NP Conjecture

Conjecture: $\mathbf{P} \neq \mathbf{NP}$. Most experts believe this is true.

> *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; everyone who could recognize a good investment strategy would be Warren Buffett. It's possible to put the point in Darwinian terms: if this is the sort of universe we inhabited, why wouldn't we already have evolved to take advantage of it?*

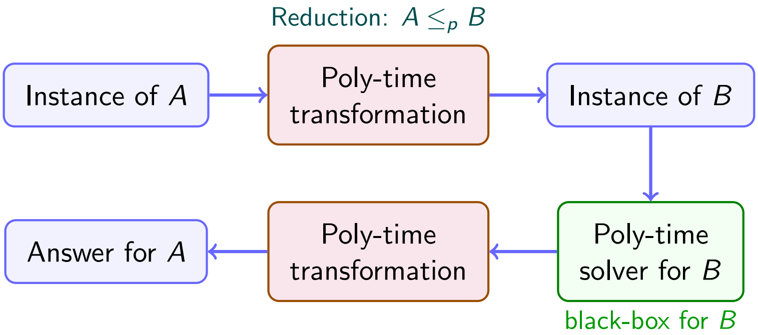
— Scott Aaronson, on Shtetl-Optimized

Reductions

Comparing Problem Difficulty

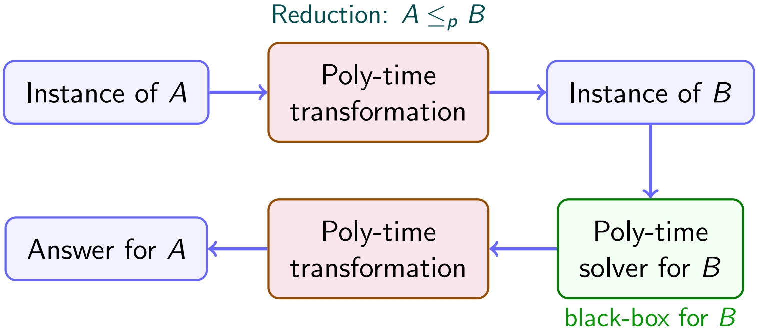
Reductions as Black-Box Transformations

- To show problem A is **no harder** than B:



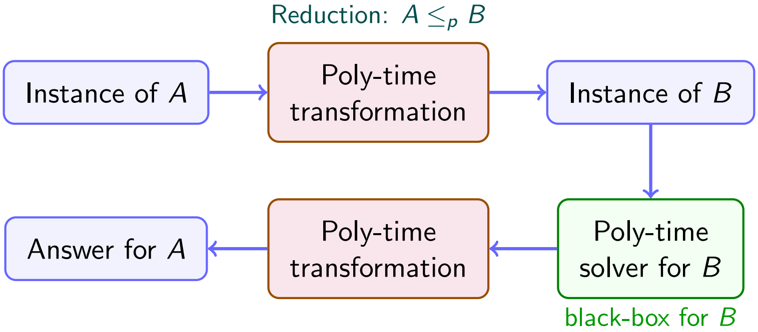
Reductions as Black-Box Transformations

- To show problem A is **no harder** than B:
  - Convert any instance of A to an instance of B in polynomial time.



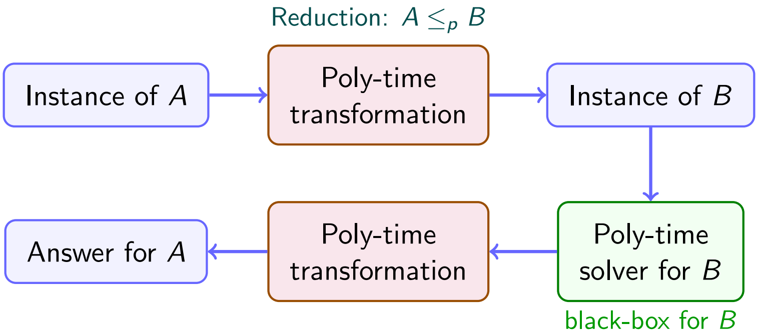
Reductions as Black-Box Transformations

- To show problem A is **no harder** than B:
  - Convert any instance of A to an instance of B in polynomial time.
  - Use a black-box solver for B.



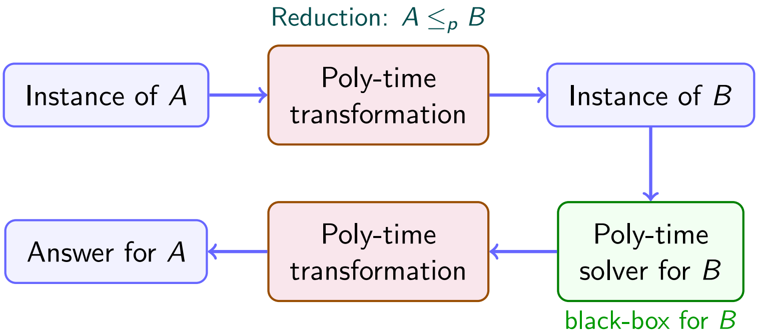
Reductions as Black-Box Transformations

- To show problem A is **no harder** than B:
  - Convert any instance of A to an instance of B in polynomial time.
  - Use a black-box solver for B.
  - Convert the answer back to an answer for A.



Reductions as Black-Box Transformations

- To show problem A is **no harder** than B:
  - Convert any instance of A to an instance of B in polynomial time.
  - Use a black-box solver for B.
  - Convert the answer back to an answer for A.
- If B is easy (in P), then A is also easy.



Reductions: Comparing Problem Difficulty

Big idea: If B were easy (poly-time), then A would also be easy.

Problem A **reduces** to problem B if, given a polynomial-time subroutine ("oracle") for B, we can solve A in polynomial time.

Reductions: Comparing Problem Difficulty

Big idea: If B were easy (poly-time), then A would also be easy.

Problem A **reduces** to problem B if, given a polynomial-time subroutine ("oracle") for B, we can solve A in polynomial time.

- We'll use reductions to show many problems are "as hard as" TSP.

Reductions: Comparing Problem Difficulty

Big idea: If B were easy (poly-time), then A would also be easy.

Problem A **reduces** to problem B if, given a polynomial-time subroutine ("oracle") for B, we can solve A in polynomial time.

- We'll use reductions to show many problems are "as hard as" TSP.
- Examples:
  - Computing the median reduces to sorting.
  - Detecting a cycle in a graph reduces to depth-first search.
  - All-pairs shortest paths reduces to repeated single-source shortest paths.

NP-hardness and NP-Completeness

NP-Hard Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \mathrm{NP}$, A reduces to B.

NP-Hard Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \mathrm{NP}$, A reduces to B.

- If you find a polynomial-time algorithm for an NP-hard problem, then *every* problem in NP becomes easy (poly-time).
- That is B is as hard as any problem in NP, or it could be even harder.

NP-Complete Problems

NP-Complete Problem

A problem B is **NP-complete** if:

1. $B \in$ NP, and
2. For every problem $A \in$ NP, A reduces to B.

NP-Complete Problems

NP-Complete Problem

A problem B is **NP-complete** if:

1. $B \in \mathrm{NP}$, and
2. For every problem $A \in \mathrm{NP}$, A reduces to B.

- If you find a polynomial-time algorithm for a NP-complete problem, then *every* problem in NP becomes easy (poly-time).
- B is one of the "hardest" problems in NP.

NP-Complete Problems

NP-Complete Problem

A problem B is **NP-complete** if:

1. $B \in \mathrm{NP}$, and
2. For every problem $A \in \mathrm{NP}$, A reduces to B.

- If you find a polynomial-time algorithm for a NP-complete problem, then *every* problem in NP becomes easy (poly-time).
- B is one of the "hardest" problems in NP.

Example: TSP is an NP-complete problem.

Is TSP as Hard as All Problems?

No! There are problems that are not even *computable*.

The Halting Problem:

Input: a program & an input.
Question: will the program eventually halt on that input?

Is TSP as Hard as All Problems?

No! There are problems that are not even *computable*.

The Halting Problem:

Input: a program & an input.
Question: will the program eventually halt on that input?

- Turing (1936): no algorithm, however slow, can solve the halting problem.
- Contrast: TSP is definitely solvable in finite time (e.g., by exhaustive search over all tours).

Is TSP as Hard as All Problems?

No! There are problems that are not even *computable*.

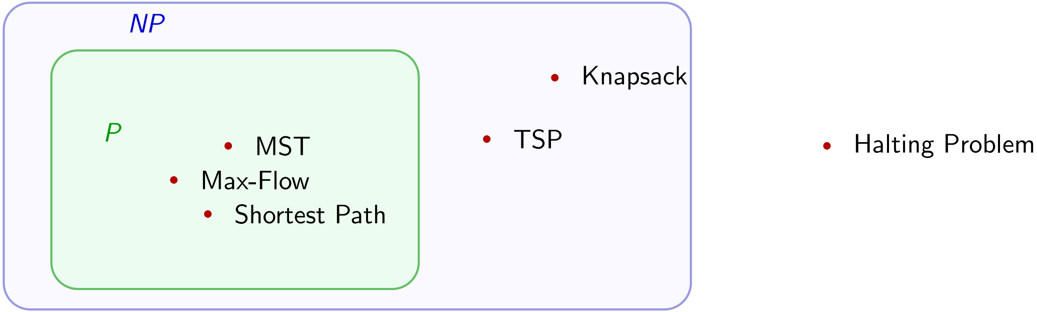
The Halting Problem:

Input: a program & an input.
Question: will the program eventually halt on that input?

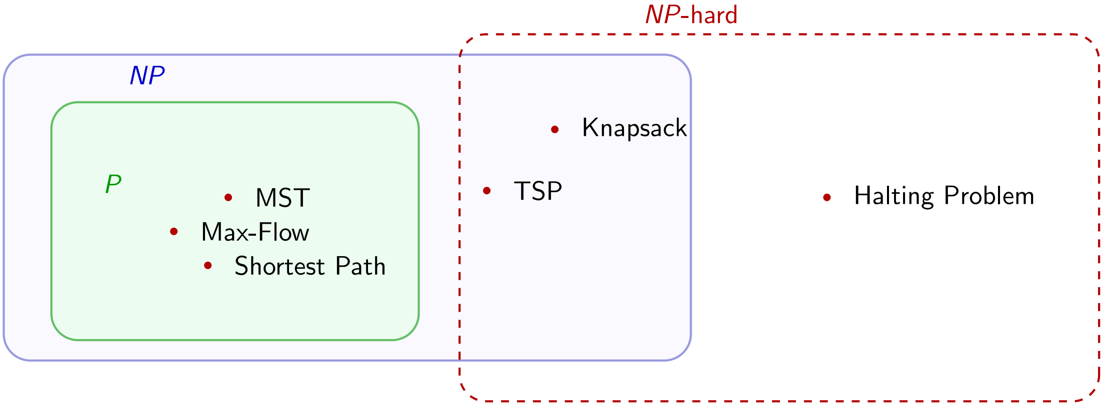
- Turing (1936): no algorithm, however slow, can solve the halting problem.
- Contrast: TSP is definitely solvable in finite time (e.g., by exhaustive search over all tours).

The halting problem is NP-hard.

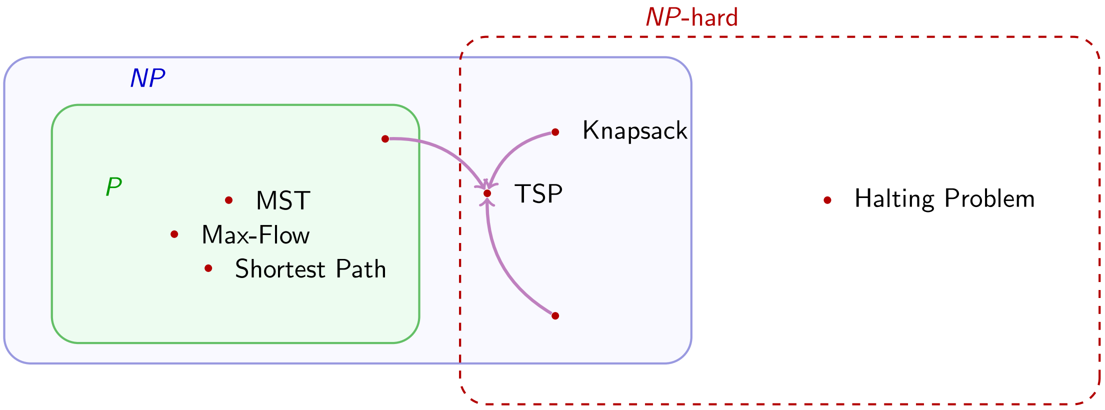
The Landscape



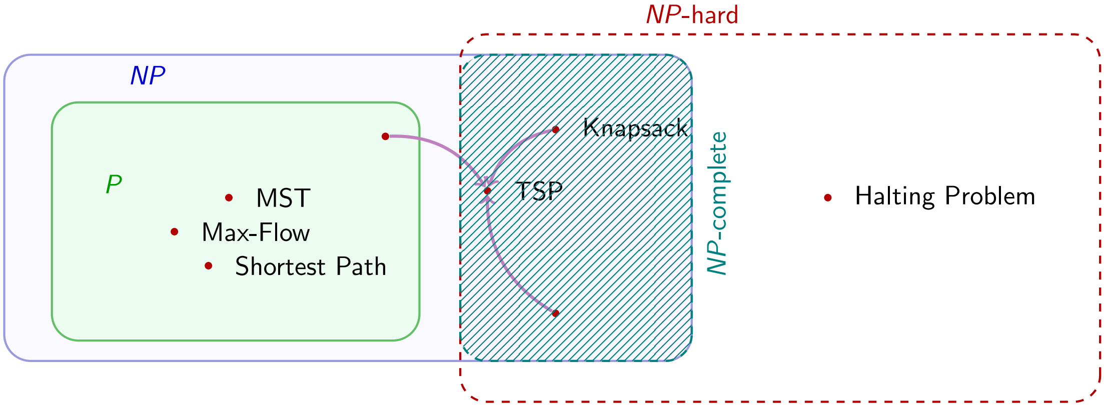
The Landscape



The Landscape



The Landscape



So your problem is NP-complete... now what?

- NP-complete does *not* mean "hopeless."
- It means: no known polytime algorithm for *all* inputs.
- We change strategy:

1. Special cases (easy structure)
2. Heuristics & approximations
3. Smarter exponential-time algorithms

Circuit Satisfiability (CIRCUIT-SAT)

where it all began

The Strange Power of NP-Completeness

It is a very strange concept.

- How can we argue that *every* problem in NP reduces to one particular problem?
- Is one problem really complex enough to capture all the nuances of every problem in NP?
- Did someone actually sit down and write a reduction from *all* NP problems to a single one?
- Do we even know all the problems that lie in NP?

The Strange Power of NP-Completeness

It is a very strange concept.

- How can we argue that *every* problem in NP reduces to one particular problem?
- Is one problem really complex enough to capture all the nuances of every problem in NP?
- Did someone actually sit down and write a reduction from *all* NP problems to a single one?
- Do we even know all the problems that lie in NP?

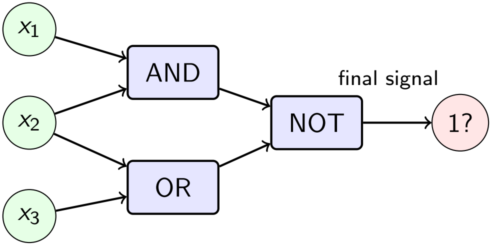
This was the breakthrough of Cook–Levin: they proved that CIRCUIT-SAT is powerful enough to express *any* NP computation.

Circuit Satisfiability (CIRCUIT-SAT)

Input: A Boolean circuit C with input bits $x_1, \ldots, x_n$ (built from AND, OR, NOT gates).

Question: Is there an assignment to $(x_1, \ldots, x_n)$ such that the output of C is $\mathbf{1}$?

- **Interpretation:** Think of the circuit as a little machine of logic gates. We ask whether there exists an input vector that makes the output wire "turn on".
- CIRCUIT-SAT is the **first NP-complete** problem (Cook and Levin 1971).



CIRCUIT-SAT Captures All of NP

Cook–Levin Theorem (1971): CIRCUIT-SAT is NP-complete.

CIRCUIT-SAT Captures All of NP

Cook–Levin Theorem (1971): CIRCUIT-SAT is NP-complete.

The proof consists of two main steps:

1. **CIRCUIT-SAT $\in$ NP.**

2. For every problem $A \in$ NP, we show $A \leq_p$ CIRCUIT-SAT.

CIRCUIT-SAT Captures All of NP

Cook–Levin Theorem (1971): CIRCUIT-SAT is NP-complete.

The proof consists of two main steps:

1. **CIRCUIT-SAT $\in$ NP.**

   Proof: an assignment to $x_1, \ldots, x_n$ can serve as a witness, and its correctness can be verified in time polynomial in the size of the input.

2. For every problem $A \in$ NP, we show $A \leq_p$ CIRCUIT-SAT.

CIRCUIT-SAT Captures All of NP

Cook–Levin Theorem (1971): CIRCUIT-SAT is NP-complete.

The proof consists of two main steps:

1. **CIRCUIT-SAT $\in$ NP.**

   Proof: an assignment to $x_1, \ldots, x_n$ can serve as a witness, and its correctness can be verified in time polynomial in the size of the input.

2. For every problem $A \in$ NP, we show $A \leq_p$ CIRCUIT-SAT.

   Proof: Next!

What NP Really Means

For every decision problem $A \in \mathrm{NP}$, each input instance x has a definite answer:

- **YES** (the property holds) or
- **NO** (it does not).

What NP Really Means

For every decision problem $A \in \mathrm{NP}$, each input instance x has a definite answer:

- **YES** (the property holds) or
- **NO** (it does not).

For example, in TSP an instance x consists of a weighted graph together with a number k, and the question is: "Is there a Hamiltonian cycle of total length at most k?"

What NP Really Means

For every decision problem $A \in \mathrm{NP}$, each input instance x has a definite answer:

- **YES** (the property holds) or
- **NO** (it does not).

For example, in TSP an instance x consists of a weighted graph together with a number k, and the question is: "Is there a Hamiltonian cycle of total length at most k?"

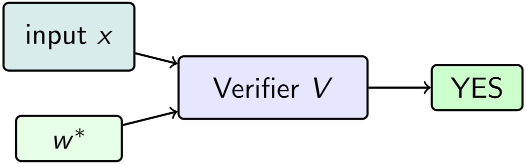
What characterizes problems in NP is how their **YES-instances** behave:

- If x is a **YES-instance** of A, then there exists a polynomial-size **witness** w that certifies this.
- There is a polynomial-time **verifier** $V(x, w)$ that checks whether w is a valid witness for x.

What NP Really Means

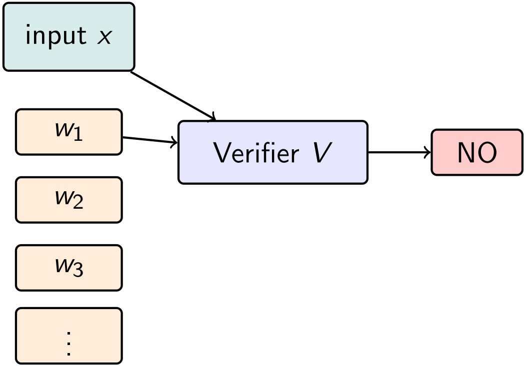
For YES-instances:

$\exists w$ such that $V(x, w) = \text{YES}$.



For NO-instances:

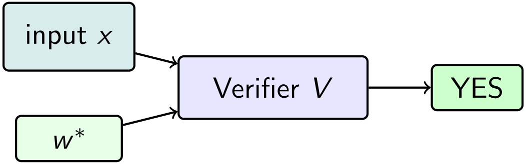
$$\forall w,\ V(x, w) = \text{NO}.$$



What NP Really Means

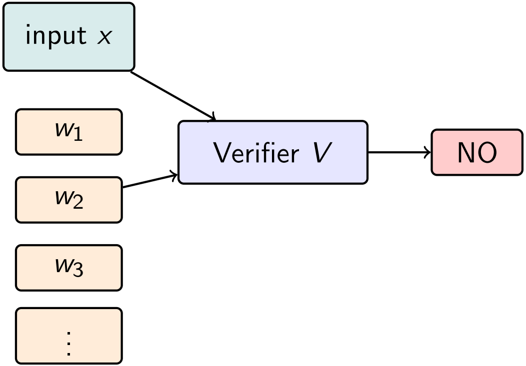
For YES-instances:

$\exists w$ such that $V(x, w) = \text{YES}$.



For NO-instances:

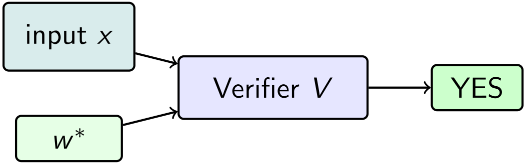
$$\forall w,\ V(x, w) = \text{NO}.$$



What NP Really Means

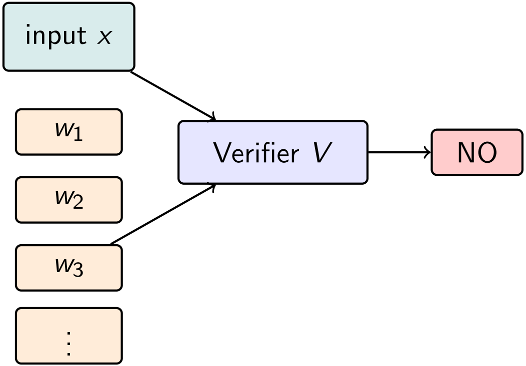
For YES-instances:

$\exists w$ such that $V(x, w) = \text{YES}$.



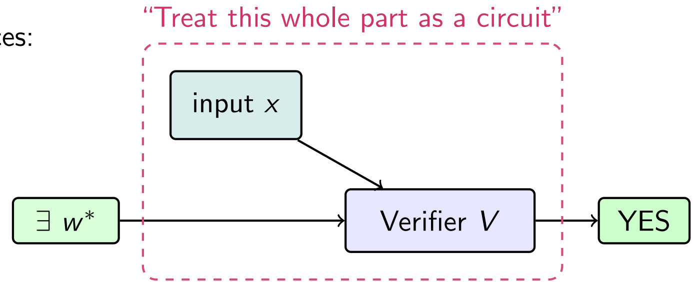
For NO-instances:

$$\forall w,\ V(x, w) = \text{NO}.$$

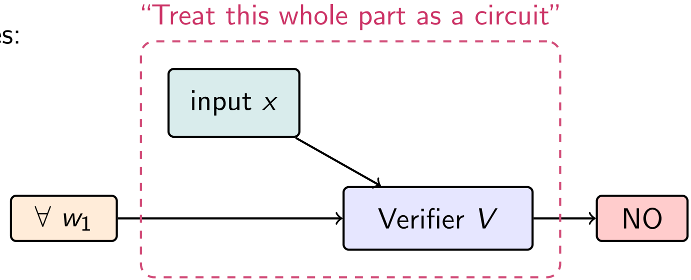


NP literally means CIRCUIT-SAT

For YES-instances:



For NO-instances:



Cook–Levin Theorem

Key idea: SAT (via CIRCUIT-SAT) can act as a *universal witness finder* for every problem in NP.

For any decision problem $A \in \mathrm{NP}$ and any instance x of A:

- If x is a **YES**-instance, then there exists a witness w that convinces the verifier $V(x, w)$ to accept.
- If x is a **NO**-instance, then *no* witness can make the verifier accept.

The Cook–Levin theorem encodes this verifier behavior into a CIRCUIT-SAT formula. Given an instance x of A, we construct a Boolean formula Φ_x such that:

$$\Phi_x \text{ is satisfiable} \quad \Longleftrightarrow \quad \exists w : V(x, w) = \text{accept}.$$

Thus SAT simulates the entire accepting computation of the verifier— it captures the witness *and* every step showing that the witness is correct.

References

Roughgarden, T. (2022).
Algorithms Illuminated: Omnibus Edition.
Soundlikeyourself Publishing, LLC.