Halting Problem & Problem Classification

Halting Problem:

- No general algorithm can solve it for all cases (undecidable)
- Not solvable in any amount of time — not even non-polynomial
- No formal proof method exists to solve it algorithmically

Classifying Problems:

- Examples of problems not known to be solvable in polynomial time:
  - **Longest Path Problem**
  - **Travelling Salesman Problem (TSP)**
  - **Subset Sum Problem**
  - **Clique Problem**
  - **Independent Set Problem**

Defining a Problem in Algorithms

What is a computational problem?

- Given an input, compute an output that satisfies some condition.
- Often represented as a function: $x \rightarrow y$

General Case

- **Input:** A list of numbers x
- **Output:** A corresponding list y, where each y_i depends on x_i
- Mapping a complex output list can be messy and hard to analyze or optimize.

Simplifying with Decision Problems

- To reduce complexity, we often ask a simpler question for each input:
  - Is the answer **yes (1)** or **no (0)**?
- These are called **decision problems**.
- Decision problems are easier to classify and analyze, especially in complexity theory.

MST – Decision Version

Input:

- A weighted graph G
- A number p

Output:

- 1 if there exists an MST in G with total weight $\leq p$
- 0 otherwise

Simplified Form:

- $G, p \rightarrow 1$ if $\exists$ an MST in G with weight $\leq p$
- Output is binary: 1 or 0

Optimisation Problem – MST

Type:

- Optimisation Problem (minimization)

Original Formulation:

- **Input:** A weighted graph G
- **Output:** The minimum total weight of its MST

Goal:

- Find the value (not just existence) of the minimum spanning tree
- Maps input to a numeric output, not just 0/1

Decision vs. Optimisation

Question 1:

- If the decision version is solved in $O(n^2)$, can we solve the optimisation version in $O(n \log n)$?
- Approach: binary search over possible values of p

Question 2:

- Can we solve the decision problem using the optimisation problem?
- Yes:
  - Compute MST weight using optimisation
  - Compare result to p: Output 1 if $\leq p$, else 0
  - Simple post-check after solving optimisation

Understanding NP

NP (Nondeterministic Polynomial time):

- Class of problems for which yes instance can be verified

Key Concepts:

- **Instance:** fixed problem input
- **Prover–Verifier Model:**
  - **Prover:** Provides a proposed solution (0 or 1)
  - If the answer is "No" $\rightarrow$ no requirement to prove
  - If the answer is "Yes" $\rightarrow$ must provide polynomial-time verifiable certificate

Examples of NP Problems

Subset Sum:

- Given a set of integers, is there a subset whose sum is zero?
- If yes, the subset itself serves as a certificate verifiable in polynomial time

Longest Path:

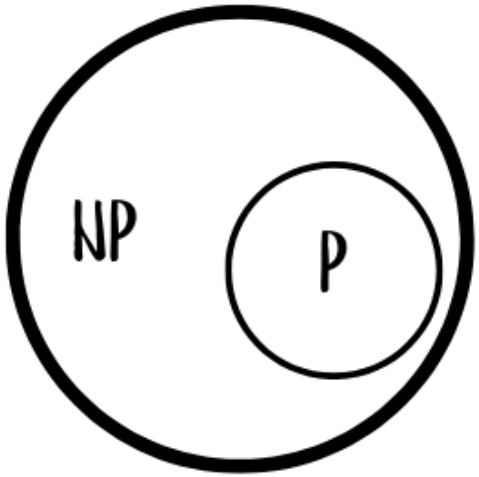
- Given a graph and integer k, is there a simple path of length $\geq k$?
- If yes, the specific path can be checked efficiently

NP Characteristics

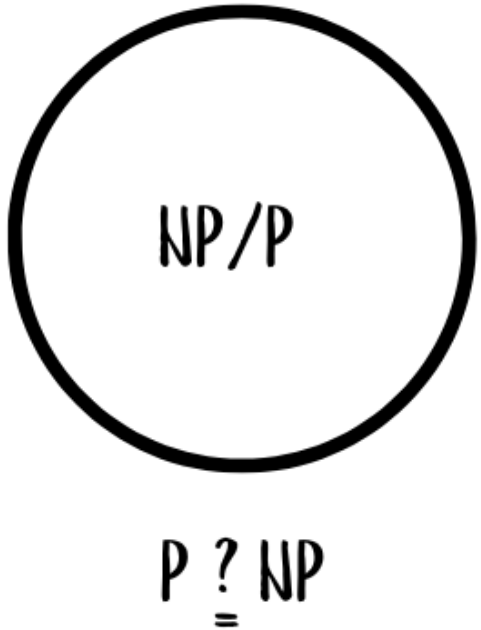
- Focuses on **verification** rather than solution
- The certificate must be **polynomial** in size
- Verification must run in **polynomial time**

An open question

Is the picture like this?



Or like this?



Complexity Classes

Reduction: P_1, P_2 (2 problems)

- Is a way to transform P_1 instance (X) to P_2 instance (Y) such that: If $P_1(X)$ is a yes-instance, then $P_2(Y)$ must also be a yes-instance and the other way around
- Basically solve P_1 using P_2

NP-hard:

- Is a class of problems y such that every x in NP can be poly-reduced to [some] y

NP-complete:

- Problems which are in NP $\cap$ NP-hard

Longest Path Problem

- The **Longest Path** problem is an **NP-Hard** problem.
- It involves finding the longest simple path between two nodes in a graph.
- Finding the longest path cannot be done efficiently in polynomial time.
- **NP-Hard** means the problem is at least as hard as any problem in NP.

Relationship: NP $\Rightarrow$ Longest Path (via Reduction)

What Does This Mean?

Every problem in **NP** can be **reduced** in polynomial time to the **Longest Path** problem.

- This implies that **Longest Path** is **NP-Hard**.
- If we can solve the Longest Path problem, we can solve any problem in NP by reducing it to Longest Path.
- A **black-box solver** for Longest Path would allow us to solve all NP problems.

Why Are These Problems Important?

- We cannot solve the Longest Path problem efficiently, but we can use it as a benchmark.
- The Longest Path problem is NP-Hard, meaning it is at least as hard as any other NP problem.
- This provides **evidence** that certain problems are fundamentally hard to solve.

Significance

The difficulty of the Longest Path problem helps us understand the boundaries of **NP-completeness** and **NP-hardness**. This insight allows us to better grasp the complexity of other problems in **NP**.

Relationship: NP $\Rightarrow$ Longest Path (via Reduction)

Key Insight

Longest Path is at the heart of **NP-Hard** problems. If we could solve it efficiently in polynomial time, we would have a solution for all NP problems.

- Solving Longest Path would imply **P = NP**, a famous unsolved question in computational complexity theory.
- The power of **polynomial-time reduction** means we can reduce any NP problem to Longest Path.
- This reduction shows that finding an efficient algorithm for Longest Path would unlock the ability to solve all NP problems.

The Challenge of Classifying Problems

Why Classification Was Difficult

- Early on, there was no clear way to compare the difficulty of problems.
- Researchers needed a way to formalize when one problem is "at least as hard" as another.
- This led to the idea of using **reductions** — transforming one problem into another.

The Breakthrough: Cook and Levin

- In 1971, **Stephen Cook** introduced the concept of NP-Completeness with his proof that **SAT** is NP-Complete.
- Around the same time, **Leonid Levin** independently discovered similar ideas in the Soviet Union.
- Their work created the foundation for classifying many hard problems using reductions.

The Challenge of Classifying Problems (Continued)

Impact

Cook and Levin's work made it possible to group problems by difficulty using reductions. This helped define what we now call **NP-Complete** problems — the hardest problems in NP.

Levin-Cook Theorem: Breakthrough and Impact

The Breakthrough

- **Stephen Cook (1971)** and **Leonid Levin (1973)** independently proved:
- The **Boolean Satisfiability Problem (SAT)** is the **first NP-Complete** problem.

Why It Mattered

- SAT was the **first** known NP-Complete problem.
- Their result created a **starting point** to classify other problems.
- Any new problem could now be shown NP-Complete by **reducing SAT** to it.
- This was a **turning point** in computational complexity theory.

SAT – The First NP-Complete Problem

What is SAT?

- **Input:** A Boolean formula in **Conjunctive Normal Form (CNF)**:
  - CNF = AND of **clauses**, each clause is an OR of literals (variables or their negations).
- **Goal:** Does there exist a true/false assignment that makes the formula true?

Example

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

- **Clauses:**
  - Clause 1: $(x_1 \vee x_2 \vee \neg x_3)$
  - Clause 2: $(x_2 \vee x_3)$
  - Clause 3: $(\neg x_1)$
  - Clause 4: $(x_1 \vee x_4)$

Evaluating SAT Examples

- **Simple examples:**
  - $(x_1) \wedge (x_2) \rightarrow$ **Satisfiable**, e.g., $x_1 = \mathrm{T},\ x_2 = \mathrm{T}$
  - $(\neg x_1) \wedge (x_2) \rightarrow$ **Satisfiable**, e.g., $x_1 = \mathrm{F},\ x_2 = \mathrm{T}$
  - $(\neg x_1) \wedge (x_1) \rightarrow$ **Unsatisfiable** (contradiction)
- **Larger example:**
  - Formula: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_1) \wedge (x_1 \vee x_4)$
  - Try assignment: $x_1 = \mathrm{F},\ x_2 = \mathrm{T},\ x_3 = \mathrm{F},\ x_4 = \mathrm{T}$
  - **Check:**
    - Clause 1: $(F \vee T \vee T) = T$
    - Clause 2: $(T \vee F) = T$
    - Clause 3: $(T) = T$
    - Clause 4: $(F \vee T) = T$
  - **Result:** Satisfied (All Clauses are True)

Is SAT in NP?

- **Yes**, SAT $\in$ NP.
- A satisfying assignment (e.g., $x_1 = T, x_2 = F, \dots$) acts as a **certificate**.
- Given such an assignment, we can **verify** whether the formula is true in **polynomial time**.
- **To prove SAT is NP-Complete**, we must also show it is **NP-Hard**.
- Cook and Levin showed all problems in NP can be **reduced to SAT**.
- **Implication:** If one problem in a set of problems is NP-Complete, all other problems in that set are also NP-Complete, as they can be reduced to each other.

Independent Set $\leq_p$ SAT

- **Independent Set Problem**:
  - Given a graph $G = (V, E)$ and an integer k
  - Does there exist a subset $W \subseteq V$, with $|W| \geq k$, such that:
  - No two vertices in W are connected by an edge, i.e., $\forall u, v \in W : (u, v) \notin E$
- **Reduction:** Independent Set $\leq_p$ SAT
  - This means we can **transform any instance** of Independent Set into a SAT instance in polynomial time.
  - So, if we could solve SAT efficiently, we could solve Independent Set efficiently.

Independent Set Permutations (Examples)

Figure: $\{1\}$

Figure: $\{1\}, \{2\}$

Independent Set Permutations (Examples)

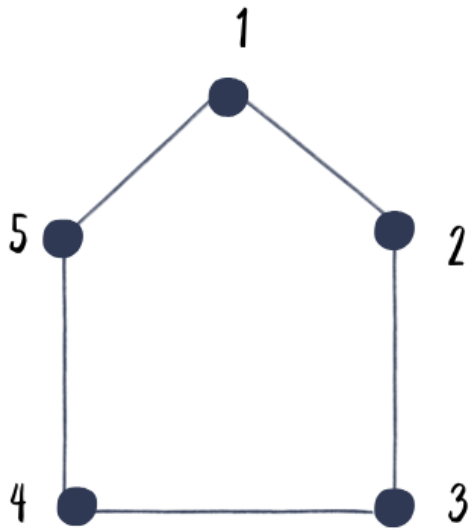
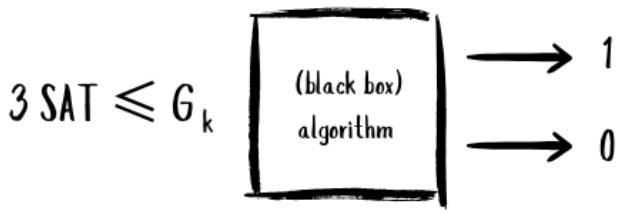


Figure: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{5,3\}, \{2,4\}, \{1,3\}, \{1,4\}, \{5,2\}$

Maximum Independent Set (MIS)

- **MIS Problem:** Given a graph $G = (V, E)$ and a number k
- Question: Is there an independent set of size $\geq k$ in G?
- MIS is **NP-Complete**.
- **Implication:** If you could solve MIS efficiently, you could solve SAT as well.



Reduction: 3-SAT to Maximum Independent Set

Example 3-SAT Formula

- Variables: x_1, x_2, x_3, x_4
- Literals: $x_1, x_2, \neg x_3, \neg x_4$, etc.

Reduction Process:

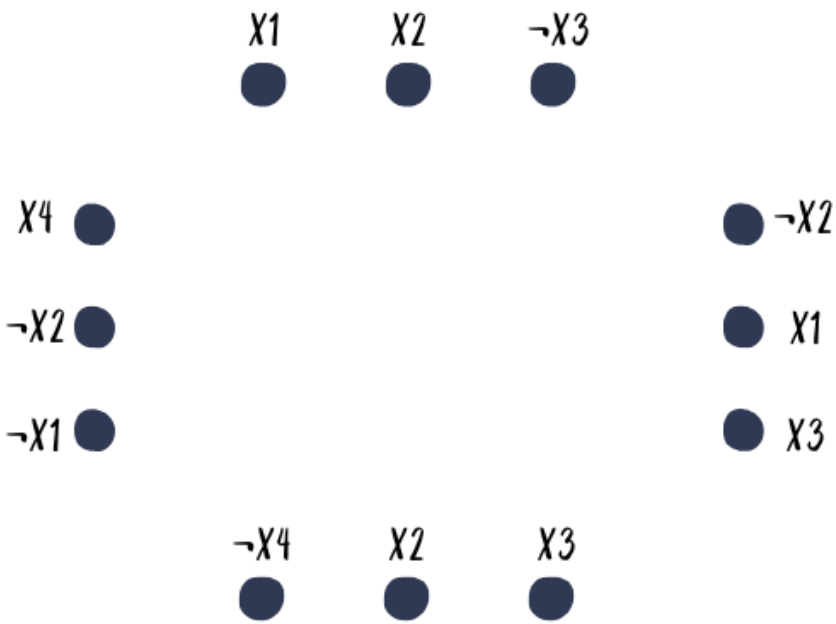
1. Create a vertex for each literal in each clause
2. Connect contradicting literals across clauses (x_i and $\neg x_i$)
3. Connect literals from the same clause (making clause triangles)
4. Set $k =$ number of clauses (here, $k = 4$)

Solve:

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

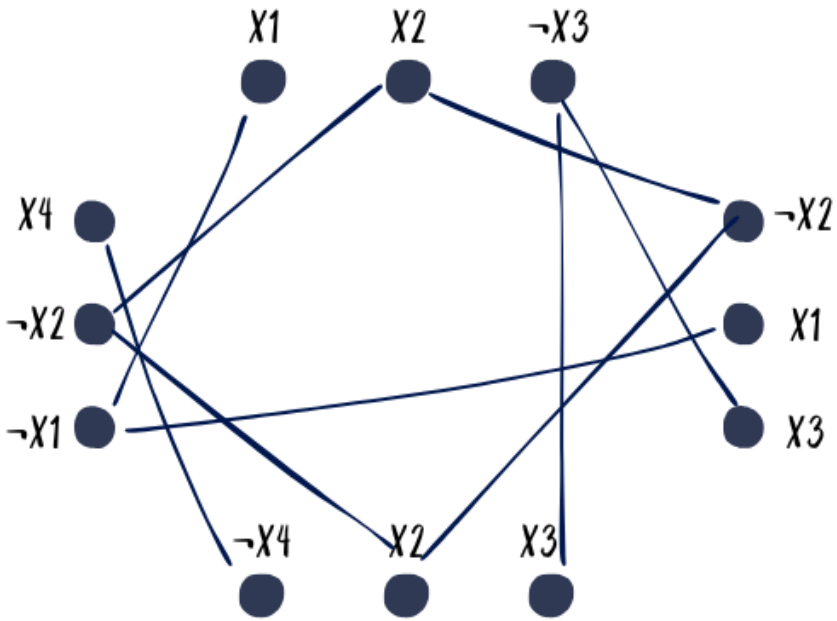
Step 1

For each literal in your formula, create a vertex



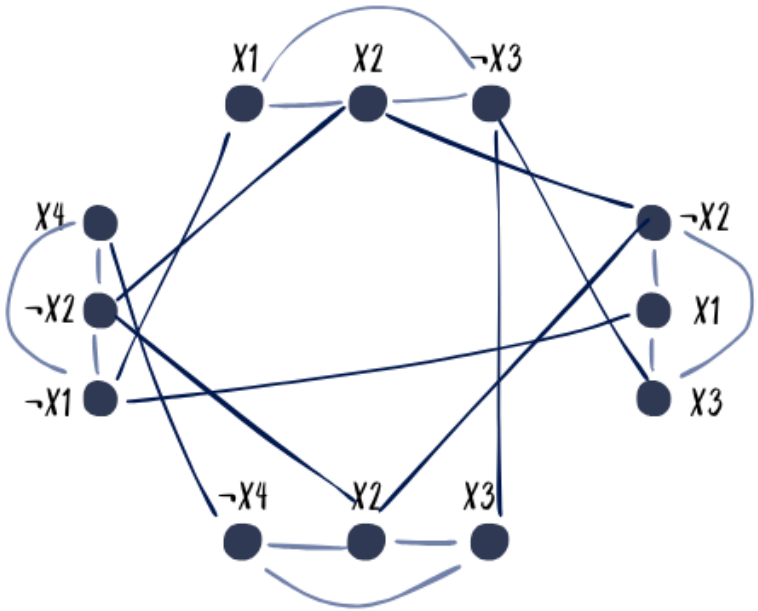
Step 2

Across clauses, add an edge between each literal and its negation



Step 3

Add all edges among vertices of the same clause
(k = number of clauses)



Step 4

Independent set: