CSE 317: Design and Analysis of Algorithms

Shahid Hussain

Fall 2025

Computational Complexity

Decision Problems vs Optimization Problems

Decision Problem

A computational problem is called *decision problem* if the answer is either **yes** or **no**. For example:

- **Is there a path of length k in a graph G?**
- **Is a given array A sorted in non-decreasing order?**

Optimization Problem

A computational problem is called *optimization problem* if the answer is a value that needs to be maximized or minimized. For example:

- **Find the shortest path between two nodes in a graph G.**
- **Find the minimum spanning tree of a graph G.**

Computational Complexity

- **Computational complexity** is a field that deals with measuring the resources required to solve a computational problem and classifying problems based on their resource requirements
- The most common resource is **time** required to solve a problem
- Often we are only interested in the decision problems and we are interested in the time required to solve the problem

Model of Computation

- We need a model of computation to measure the time required to solve a problem
- The most common model of computation is the **Turing machine**
- A **Turing machine** is an abstract machine that can simulate any algorithm
- The time complexity of an algorithm is measured in terms of the number of steps required to solve a problem

Languages and Decision Problems

Language

A *language* is a set of strings over a finite alphabet (Σ) For example, the set of all binary strings that represent prime numbers is a language

- We can now redefine the decision problem as follows:
- **Decision Problem:** Given a string $x \in \Sigma^*$, is x in the language L?
- That is., a decision problem is a kind of membership query for a string in a language

Algorithms and Machines

- Let Σ be a finite alphabet and $\mathcal{A}$ be an algorithm (machine) that implements a mapping
$$\varphi : \Sigma^* \to \{0, 1\}$$
- We say $\mathcal{A}$ is a polynomial time algorithm if there exists a polynomial $p(n)$ such that for all $x \in \Sigma^*$, the algorithm $\mathcal{A}$ halts in at most $p(|x|)$ steps
-

The Class **P**

The class P

The class **P** contains all languages (decision problems) which can be decided by some polynomial time algorithm i.e., the membership queries can be computed in polynomial time by some algorithm

The Class NP

- Let Σ be a finite alphabet
- We say $Q : \Sigma^* \to \{0,1\}$ is a polynomial time predicate if there exists a polynomial time algorithm that for given words $\alpha, \beta \in \Sigma^*$, compute $Q(\alpha, \beta)$

The class NP

We say that a language $L \subseteq \Sigma^*$ is in the class **NP** if and only if there exists a polynomial q and a polynomial time predicate Q such that for any word $\alpha \in \Sigma^*$, we have $\alpha \in L$ if and only if there exists a word $\beta \in \Sigma^*$ such that $Q(\alpha, \beta) = 1$ and $|\beta| \leq q(|\alpha|)$ in other words:

$$L \subseteq \mathbf{NP} \iff \exists q \; \exists Q \; \forall \alpha \in \Sigma^* \; \exists \beta \in \Sigma^* \; Q(\alpha, \beta) = 1, \quad \text{s.t.} \quad |\beta| \leq q(|\alpha|)$$

The word β is called the *certificate* and the algorithm which computes the predicate Q is called the *verifier*.

The Classes P and NP

Theorem

$\mathsf{P} \subseteq \mathsf{NP}$.

Proof.

- Let $L \in \mathsf{P}$, $L \subseteq \Sigma^*$, and $q(n) = 1$
- Let us consider a predicate $Q(\alpha, \beta)$ such that $Q(\alpha, \beta) = 1$ if and only if $\alpha \in L$
- Since $L \in \mathsf{P}$, the predicate Q is a polynomial time predicate
- It is clear that $\alpha \in L$ if and only if there exists $\beta \in \Sigma^*$ such that $|\beta| \leq 1$ and $Q(\alpha, \beta) = 1$ (we can take $\beta = \epsilon$)

$\square$

Reduction

Polynomial time reduction

We say that a language $L_1 \subseteq \Sigma^*$ is a *polynomial-time reducible* to a language $L_2 \in \Sigma^*$ (written as $L_1 \leq_{\mathsf{P}} L_2$) if there exists a polynomial time computable function $\varphi : \Sigma^* \to \Sigma^*$ such that, for all α we have $\alpha \in L_1$ if and only if $\varphi(\alpha) \in L_2$.

Proposition

1. *If* $L_1 \leq_{\mathsf{P}} L_2$ *and* $L_2 \in \mathsf{P}$ *then* $L_1 \in \mathsf{P}$.
2. *If* $L_1 \leq_{\mathsf{P}} L_2$ *and* $L_1 \notin \mathsf{P}$ *then* $L_2 \notin \mathsf{P}$.
3. *If* $L_1 \leq_{\mathsf{P}} L_2$ *and* $L_2 \leq_{\mathsf{P}} L_3$ *then* $L_1 \leq_{\mathsf{P}} L_3$. *(Transitivity property.)*

NP-hard and **NP**-complete Problems

NP-hard problems

A language L is **NP**-hard if for all $L' \in \mathbf{NP}$, we have $L' \leq_{\mathbf{P}} L$.

NP-complete problems

A language L is **NP**-complete if $L \in \mathbf{NP}$ and L is **NP**-hard.

- If L is **NP**-hard and $L \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
- If L is **NP**-hard and $\mathbf{P} \neq \mathbf{NP}$, then $L \notin \mathbf{P}$.

Boolean Functions and CNF

Boolean Function/Formula

A function $f : \{0,1\}^n \to \{0,1\}$ is called a *boolean function* of n variables.

- A boolean formula of the following kind is called a *conjunctive normal form* (CNF):

  $$f(x_1, x_2, \ldots, x_n) = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

  where each C_j is a *clause* of the form $l_{j1} \vee l_{j2} \vee \ldots \vee l_{jk}$ and each l_{ji} is a *literal* which is either a variable x_i or its negation $\neg x_i$
- We assume that each variable occurs in each clause C_j at most once

Satisfiability Problem

Satisfiability Problem

Decision Problem: SAT

Setup: A boolean formula f of n variables in CNF

Question: Is there an assignment of values to the variables such that f evaluates to TRUE?

- We show that SAT is **NP**-complete
- We need to show that SAT is in **NP** and it is **NP**-hard

Satisfiability Problem is in **NP**

- Given a boolean formula f in CNF, we can verify in polynomial time whether a given assignment of values to the variables makes f evaluate to TRUE
- We can verify this by checking each clause C_j and checking if at least one literal in the clause evaluates to TRUE
- Thus, SAT is in **NP**

Satisfiability Problem is **NP-hard**

Theorem (Cook-Levin 1971)

SAT *is* ***NP**-complete.*

- This proof was independently discovered by Stephen Cook (in United States) and Leonid Levin (in, then, Soviet Union) in 1971
- This proof laid down the theoretical/mathematical foundation of the theory of **NP**-complete problems
- In the subsequent we present some **NP-complete** problems all of which can proved to be **NP-hard** using the polynomial time reduction (due to transitivity property of reduction)

The 3 SAT problem

Decision Problem: 3SAT

Setup: A boolean formula f of n variables in CNF where each clause has at most 3 literals

Question: Is there an assignment of values to the variables such that f evaluates to TRUE?

- 3SAT $\in$ **NP** (trivial)
- SAT $\leq_{\mathbf{P}}$ 3SAT (left as an exercise)

Independent Set Problem

Independent Set Problem

Let $G = (V, E)$ be an undirected graph. We say a subset of vertices $S \subseteq V$ is *independent set* if no two vertices in S are adjacent. There is no edge that joins any two vertices in S.

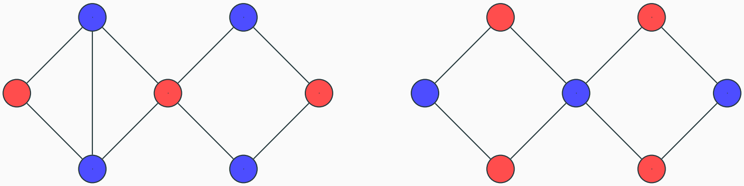
The optimization version of the *independent set problem* is to find an independent set of maximum cardinality in G.

Decision Problem: IS

Setup: An undirected graph $G = (V, E)$ and an integer k

Question: Is there an independent set of size at least k in G?

Independent Set Problem



Proving Independent Set to **NP-complete**

1. We need to show that $\textsc{is} \in \mathbf{NP}$
2. We show that for some **NP**-complete problem Π, $\Pi \leq_{\mathbf{P}} \textsc{is}$

Showing Independent Set is in **NP**

- Given a graph $G = (V, E)$ and an integer k, we can verify in polynomial time whether a given subset of vertices $S \subseteq V$ is an independent set of size at least k
- We can verify this by checking if no two vertices in S are adjacent and if $|S| \geq k$
- Thus, IS is in **NP**

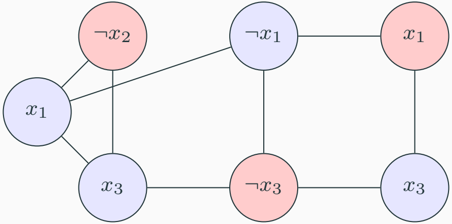
Showing Independent Set is **NP-hard**

- We show that IS is **NP**-hard by showing that 3SAT $\leq_{\mathbf{P}}$ IS
- Given a boolean formula $f = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ in 3CNF, where each $C_j = l_{j1} \vee l_{j2} \vee l_{j3}$, we construct a graph $G = (V, E)$ such that f is satisfiable if and only if G has an independent set of size at least k
- We construct the graph G as follows:
  - Each literal in each clause has a corresponding vertex in G
  - Each clause C_j has a corresponding triangle in G
  - Each vertex v from each clause is connected to $\neg v$ from other clauses and vice versa
  - We set $k = m$
  - We show that f is satisfiable if and only if G has an independent set of size at least k

Example Reduction

- Polynomial-time reduction of 3SAT to IS

$$f = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_3)$$



- Above formula is satisfied: $x_1 = 1$, $x_2 = 0$, $x_3 = 0$

Vertex Cover Problem

Let $G = (V, E)$ be an undirected graph. We say a subset of vertices $S \subseteq V$ is a *vertex cover* if every edge in E has one endpoint in S.

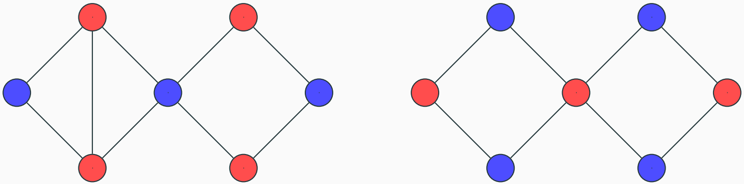
The optimization version of the *vertex cover problem* is to find a vertex cover of minimum cardinality in G.

Decision Problem: VC

Setup: An undirected graph $G = (V, E)$ and an integer k

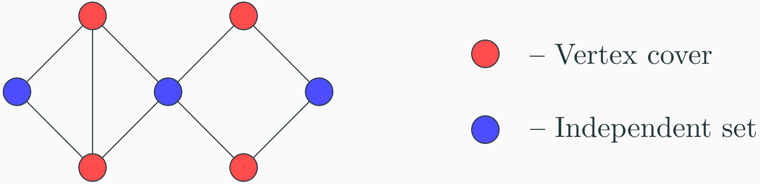
Question: Is there a vertex cover of size at most k in G?

Vertex Cover Problem



Proving Vertex Cover is NP-complete

- We show that VC is **NP**-hard by showing that IS $\leq_{\mathbf{P}}$ VC
- Clearly, if S is an independent set in G, then $V \setminus S$ is a vertex cover in G



- If S is an independent set, each $e \in E$ has at least one endpoint is in $V \setminus S$
- Therefore, $V \setminus S$ is a vertex cover in G
- It implies that G has an independent set with at least k vertices if and only if G has a vertex cover with at most $n - k$ vertices
- Therefore, VC is **NP**-complete

Set Cover Problem

Set Cover Problem

Let $\mathcal{U}$ be a set and $\mathcal{F}$ a family of subsets of $\mathcal{U}$ s.t., $\mathcal{U} = \bigcup_{S \in \mathcal{F}} S$.

The optimization version of the *set cover problem* is to find a set cover of minimum cardinality in $\mathcal{F}$.

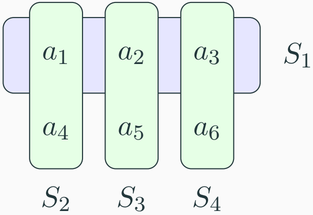
Decision Problem: SC

Setup: A finite set $\mathcal{U}$, a collection of subsets $\mathcal{F}$ of $\mathcal{U}$, and an integer k

Question: Is there a set cover of size at most k in $\mathcal{F}$?

Set Cover Problem

- Let $\mathcal{U} = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ and $\mathcal{F} = \{S_1, S_2, S_3, S_4\}$
- $S_1 = \{a_1, a_2, a_3\}$, $S_2 = \{a_1, a_4\}$, $S_3 = \{a_2, a_5\}$, $S_4 = \{a_3, a_6\}$



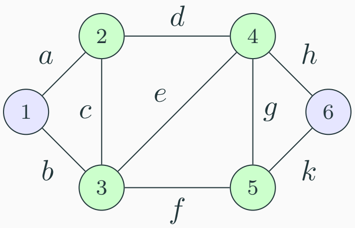
Set Cover Problem

- It is easy to show that SC is in **NP**
- We show that VC $\leq_{\mathbf{P}}$ SC
- Let $G = (V, E)$ be an undirected graph and k be an integer
- We construct a set cover problem $(\mathcal{U}_G, \mathcal{F}_G)$ as follows:
    - Let $\mathcal{U}_G = E$ and $\mathcal{F}_G = \{S_v \mid v \in V\}$
    - Where S_v is the set of edges from E that have v as an endpoint
    - We set $k = |V|$
    - We show that $\{v_{i_1}, \ldots, v_{i_k}\}$ is a vertex cover for G if and only if:

$$\bigcup_{S_v \in \{S_{v_{i_1}}, \ldots, S_{v_{i_k}}\}} S_v = \mathcal{U}_G$$

- Thus, SC is **NP**-complete

Set Cover Problem



- The set cover problem $\mathcal{U}_G$, $\mathcal{F}_G$ for the above graph $G = (V, E)$ is:
- $\mathcal{U}_G = \{a, b, c, d, e, f, g, h, k\}$
- $\mathcal{F}_G = \{S_1, S_2, S_3, S_4, S_5, S_6\}$
- $S_1 = \{a, d\}$, $S_2 = \{a, c\}$, $S_3 = \{d, e\}$, $S_4 = \{d, h\}$, $S_5 = \{e, g\}$, $S_6 = \{h, k\}$

Working with **NP-complete** Problems

Working with **NP-complete** Problems

- Most practical problems are **NP**-hard
- This does not stop us from solving these problems
- We can often find an approximate solution to these problems
- In the following we will discuss approximation algorithm for some **NP**-hard problems
- Most approximate algorithms are greedy in nature

Approximate Algorithm for Set Cover Problem

- The set cover problem is **NP**-hard
- Consider the following algorithm

Algorithm: GREEDY-SET-COVER

Input: A finite set $\mathcal{U}$, a collection of subsets $\mathcal{F}$ of $\mathcal{U}$

Output: A set cover of $\mathcal{U}$

- during each step this algorithm chooses a subset $S_i \in \mathcal{F}$ with minimal index i and maximum number of elements uncovered during previous steps (S_i covers all elements from $\mathcal{U}$ that belong to S_i)
- This algorithm will halt when all elements from $\mathcal{U}$ are covered