

# CSCI3160 Design and Analysis of Algorithms (2025 Fall)

## Approximation Algorithms 1: Introduction, Vertex Cover and MAX-3SAT

Instructor: Xiao Liang<sup>1</sup>

Department of Computer Science and Engineering  
The Chinese University of Hong Kong

---

<sup>1</sup>These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to [Prof. Tao's version from 2024 Fall](#) for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

## Motivation

We have learned several algorithms that help us solve problems efficiently:

- Sorting in time  $O(n \log n)$
- Matrix multiplication in time  $O(n^{2.81})$
- FFT-based polynomial multiplication in time  $O(n \log n)$
- Activity selection in time  $O(n \log n)$
- ...
- All-Pairs Shortest Paths in time  $O(|V|(|V| + |E|) \log(|V|))$

**However:** there are still many problems **of practical significance** for which no efficient (i.e., polynomial-time) algorithms are currently known to us humans.

## Example 1: Graph-3-Coloring

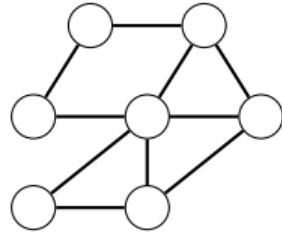
A **graph coloring** assigns colors to the vertices of a graph so that no two adjacent vertices share the same color.

In the **Graph-3-Coloring** problem, we ask:

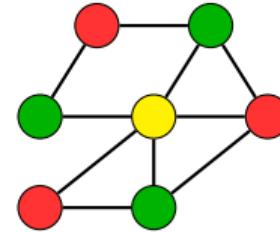
Can the vertices of a given graph be colored with at most 3 colors such that adjacent vertices have different colors?

**Extension:** 3-coloring is a special case of the general k-coloring problem.

## Exemplary Graphs



(a) A 3-colorable graph



(b) A 3-coloring scheme

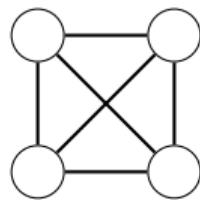


Figure: A non-3-colorable graph

## Application 1: Map Coloring

- Each region on a map can be represented as a vertex in a graph.
- An edge connects two vertices if the corresponding regions share a common border.
- The goal is to color each region so that no two adjacent regions have the same color.
- If the map can be colored with 3 colors, the corresponding graph is 3-colorable.
- This has applications in geography, political boundary planning, and resource distribution.

## Application 2: Scheduling Exams

- Vertices represent courses.
- An edge connects two courses if they have students in common.
- Goal: assign each course to one of 3 time slots, avoiding conflicts.
- 3-coloring determines if this is possible.
- If there are  $k$  courses under consideration, then it corresponds to the  $k$ -coloring problem.

## Application 3: Frequency Assignment

- Nodes represent transmitters in a communication network.
- Edges represent interference (proximity).
- Assign 3 frequencies to avoid interference between neighbors.
- 3-coloring tells us if this can be done using only 3 frequencies.
- In general  $k$ -coloring tells us if this can be done using only  $k$  frequencies.

# Subset Sum Problem and Applications

## Problem Statement:

- Given a set of integers  $S = \{x_1, x_2, \dots, x_n\}$  and a target integer  $T$ ,
- Does there exist a subset  $S' \subseteq S$  such that the sum of elements in  $S'$  is exactly  $T$ ?

## Example 1:

- $S = \{3, 34, 4, 12, 5, 2\}$ ,  $T = 9$
- Yes:  $\{4, 5\}$  or  $\{3, 4, 2\}$  sum to 9.

## Example 2:

- $S = \{3, 5, 9, 13\}$ ,  $T = 7$
- No!

# Application 1: Budget Allocation

## Scenario:

- A company has a list of proposed projects, each with a known cost.
- The total available budget is a fixed amount  $T$ .
- The goal is to determine if there is a combination of projects whose total cost exactly matches the budget.

## Example:

- Projects:  $\{P_1 : \$30k, P_2 : \$50k, P_3 : \$20k, P_4 : \$40k\}$
- Budget:  $\$90k$
- Is there a subset of projects that costs exactly  $\$90k$ ?
- Yes:  $\{P_2, P_3, P_4\} \rightarrow 50k + 20k + 20k = 90k$

## Relevance:

- Subset sum helps in decision support for finance and planning.
- Used in automated budget optimization tools and resource allocation systems.

## Application 2: Packing and Logistics

### Scenario:

- A shipping company needs to fill containers with items of different weights or volumes.
- Goal: Select a subset of items that perfectly fills a container of limited capacity.

### Example:

- Items:  $\{w_1 = 3\text{kg}, w_2 = 7\text{kg}, w_3 = 2\text{kg}, w_4 = 6\text{kg}\}$
- Container capacity:  $9\text{kg}$
- Feasible subset:  $\{w_2, w_3\} \rightarrow 7 + 2 = 9$

### Relevance:

- Subset sum models the core problem in bin packing and cargo loading.
- Used in warehouse automation, shipping logistics, and supply chain optimization.

Unfortunately, we currently do not know any polynomial-time algorithms for Graph 3-Coloring, Subset Sum, or many other problems that have broad applications and significant real-world importance.

A central research theme for Theoretical Computer Science (TCS):

- **What can we do about these hard problems?**

Two branches of computer science have been developed centered around this theme:

- **Computation Complexity:** Seeks to understand the inherent difficulty of problems and classify them based on resource requirements. (Not our focus in this course.)
- **Approximation Algorithms:** Develops efficient algorithms that produce near-optimal solutions to hard problems. (This will be our focus for the remaining lectures.)

# Branch 1: Computational Complexity

**Goal:** Understand the fundamental limits of computation.

- Classifies problems into complexity classes such as:
  - **P:** Problems solvable in polynomial time.
  - **NP:** Problems whose solutions can be verified in polynomial time.
  - **NP-complete:** The hardest problems in NP — if any one of them can be solved in polynomial time, then all of NP can.
- Addresses profound open questions like:
  - **P vs NP:** Can every efficiently verifiable problem also be efficiently solvable?
- Also studies:
  - Reductions between problems (to compare difficulty)
  - Space and time trade-offs
  - Randomized and quantum complexity classes

**Takeaway:** Computational complexity helps us understand *why* certain problems are hard, and how that hardness is structured.

## Branch 2: Approximation Algorithms

**Goal:** Design efficient algorithms that find near-optimal solutions for hard optimization problems.

- When exact solutions are computationally infeasible (e.g., NP-hard problems), we aim for **good enough** solutions in **polynomial time**.
- An approximation algorithm returns a solution whose value is within a provable factor of the optimum.
- Central questions in this area:
  - How close can we get to the optimal solution in polynomial time?
  - Are there limits (hardness of approximation) beyond which we can't do better unless  $P = NP$ ?
- Techniques include:
  - Greedy methods
  - LP/SDP relaxations
  - Randomized rounding

**Takeaway:** Approximation algorithms offer a practical path forward for solving hard problems when exact solutions are out of reach.

## An introductory Journey to Approximation Algorithms

For a rigorous discussion about Approximation Algorithms, we first need to borrow some concepts from Computational Complexity.

What's the exact meaning of "no polynomial-time algorithms are known" for some problem (e.g., Graph 3-Coloring, Subset Sum)?

- We must be precise about the **computation model** we are using.

Long story short, **Turing Machine** has become the standard model of computation in complexity theory:

- We won't explain what a Turing Machine is. But in short, it is a slightly more powerful model than the RAM model we utilize in this course so far.
- It is mathematically simple yet powerful enough to simulate any "reasonable" algorithm.
- It is **robust**: polynomial-time computations on one reasonable model can be simulated in polynomial time on a Turing Machine.
- It is **closed under composition**: combining polynomial-time Turing Machines results in a polynomial-time Turing Machine.

# Church-Turing Thesis

## Informal Statement:

*Any function that can be computed by a "reasonable" mechanical procedure (i.e., algorithm) can be computed by a Turing Machine.*

## Key Points:

- It is a **foundational hypothesis** in computer science — not a formal theorem.
- Supported by the equivalence of many independent computational models:
  - Turing Machines
  - Lambda calculus
  - Recursive functions
  - RAM machines
- All known models of computation that align with our intuitive notion of an algorithm are **no stronger than** in power.
- Thus, this thesis conjecture that the Turing Machine is an adequate model for defining what is computable (or an Algorithm).

# A Brief History of the Church-Turing Thesis

**Early 1900s:** Mathematicians, including **David Hilbert**, were seeking to formalize all of mathematics.

- Hilbert posed the famous **Entscheidungsproblem** (decision problem):
  - *Is there a general algorithm to determine whether a given mathematical statement is provable?*
- This led to the deeper question: **What exactly is an algorithm or effective procedure?**

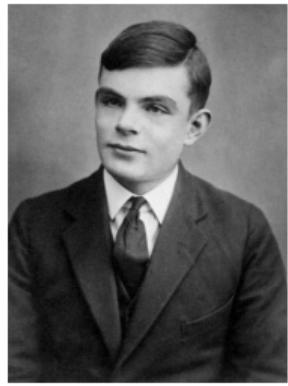
## Alonzo Church (1936):

- Proposed the **lambda calculus** as a formal model of computation.
- Introduced the notion of **computable functions**.
- Argued that this model captured all effectively calculable functions.

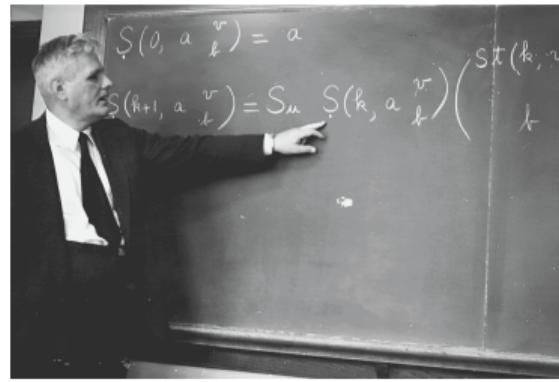
## Alan Turing (1936):

- Independently tackled the same question.
- Introduced the **Turing Machine** in his groundbreaking paper:
  - “On Computable Numbers, with an Application to the Entscheidungsproblem”
- Showed that Turing Machines could simulate any mechanical computation.

**Fun fact:** Church was Turing's academic advisor at Princeton after this work!



(a) Alan Turing



(b) Alonzo Church

# Extended Church-Turing Thesis and Quantum Computing (1/2)

## Extended Church-Turing Thesis (ECTT):

*Any "reasonable" model of computation can be **efficiently** (i.e., with at most polynomial overhead) simulated by a Turing Machine.*

### Implications:

- Suggests that all physically realizable computational models are no more powerful (in terms of efficiency) than classical computers.
- Provides a foundational assumption in classical algorithm design and complexity theory.

### Quantum Computing Challenges ECTT:

- Quantum computers operate under the laws of quantum mechanics.
- Algorithms like **Shor's algorithm** solve certain problems (e.g., integer factoring) exponentially faster than the best-known classical algorithms.
- If scalable quantum computers exist, they would violate the ECTT — showing that classical simulation may not always be efficient.

# Extended Church-Turing Thesis and Quantum Computing (2/2)

## What about Quantum Computing?

- Quantum computers operate under the laws of quantum mechanics.
- Algorithms like **Shor's algorithm** solve certain problems (e.g., integer factoring) exponentially faster than the best-known classical algorithms.
- If scalable quantum computers exist, they would violate the ECTT — showing that classical simulation may not always be efficient.

## Caveats:

- A formal and rigorous treatment would require us to:
  - build a quantum computer that could run Shor's algorithm in the real world. (Very challenging. But steady progress recently.)
  - formally prove that no polynomial-time classical algorithms can solve the Factoring problem. (Very challenging. Not much progress.)
- Even if Quantum Computing could formally refute the Extended Church-Turing Thesis, it still does not refute the original Church-Turing Thesis (what's computable).

## Summary

To reason about the hardness of computational problems, we first need a formal model of computation.

- The **Turing Machine** is the most widely accepted and robust model.

It is broadly believed that Turing Machines capture the essence of what we intuitively consider to be an **algorithm**, as formalized by the **Church-Turing Thesis** and the **Extended Church-Turing Thesis** (though the latter faces challenges from quantum computing).

With Turing Machines as our foundation, we can:

- Classify the computational difficulty of problems.
- Study the structure of complexity classes and the relationships between them.

Most relevant to our current discussion are the classes **P** and **NP** (next slide).

# Approximation Algorithms for NP-Hard Problems (1/2)

Informally,

- $\mathcal{P}$  = the set of problems that can be solved in polynomial time on a Turing machine
- $\mathcal{NP}$  = the set of problems whose solution can be efficiently verified by a Turing machine. (Think about the Graph-3-Coloring problem.)

Turing machines are formalized in CSCI3130 (Formal Languages and Automata Theory), and so is the notion of NP-hard.

Whether  $\mathcal{P} = \mathcal{NP}$  is still unsolved to this day.

## Approximation Algorithms for NP-Hard Problems (2/2)

What can we do if a problem is NP-hard?

The rest of the course will focus on a principled approach for tackling NP-hard problems:  
**approximation**.

In many problems, even though an optimal solution may be expensive to find, we can find  
**near-optimal** solutions efficiently.

Next, we will see two examples: **vertex cover** and **MAX-3SAT**.

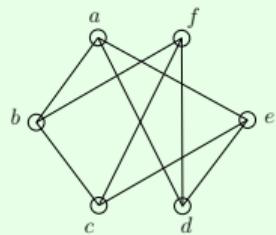
## The Vertex Cover Problem

$G = (V, E)$  is a simple undirected graph.

A subset  $S \subseteq V$  is a **vertex cover** of  $G$  if every edge  $\{u, v\} \in E$  is incident to at least one vertex in  $S$ .

**The V.C. Problem:** Find a vertex cover of the smallest size.

**Example:**



An optimal solution is  $\{a, f, c, e\}$ .

The vertex cover problem is NP-hard.

- No one has found an algorithm solving the problem in time polynomial in  $|V|$ .
- Such algorithms cannot exist if  $\mathcal{P} \neq \mathcal{NP}$ .

## Approximation Algorithms

$\mathcal{A}$  = an algorithm that, given any legal input  $G = (V, E)$ , returns a vertex cover of  $G$ .

$OPT_G$  = the smallest size of all the vertex covers of  $G$ .

$\mathcal{A}$  is a  **$\rho$ -approximate algorithm** for the vertex cover problem if, for any legal input  $G = (V, E)$ ,  $\mathcal{A}$  can return a vertex cover with size at most  $\rho \cdot OPT_G$ .

The value  $\rho$  is the **approximation ratio**.

We say that  $\mathcal{A}$  achieves an approximation ratio of  $\rho$ .

Consider the following algorithm.

**Input:**  $G = (V, E)$

$S = \emptyset$

**while**  $E$  is not empty **do**

    pick an arbitrary edge  $\{u, v\}$  in  $E$

    add  $u, v$  to  $S$

    remove from  $E$  all the edges of  $u$  and all the edges of  $v$

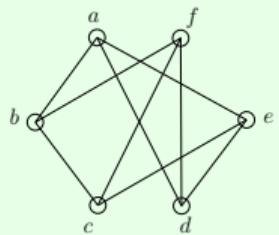
**return**  $S$

It is easy to show:

- $S$  is a vertex cover of  $G$ ;
- The algorithm runs in time polynomial to  $|V|$  and  $|E|$ .

We will prove later that the algorithm is 2-approximate.

## Example:



Suppose we start by picking edge  $\{b, c\}$ .

Then,  $S = \{b, c\}$  and  $E = \{\{a, e\}, \{a, d\}, \{d, e\}, \{d, f\}\}$ .

Any edge in  $E$  can then be chosen. Suppose we pick  $\{a, e\}$ .

Then,  $S = \{a, b, c, e\}$  and  $E = \{\{d, f\}\}$ .

Finally, pick  $\{d, f\}$ .

$S = \{a, b, c, d, e, f\}$  and  $E = \emptyset$ .

**Theorem 1:** The algorithm returns a set of at most  $2 \cdot OPT_G$  vertices.

Let  $M$  be the set of edges picked.

**Example:** In the previous example,  $M = \{\{b, c\}, \{a, e\}, \{d, f\}\}$ .

**Lemma 1:** The edges in  $M$  do not share any vertices.

The proof is left as an exercise.

**Lemma 2:**  $|M| \leq OPT_G$ .

**Proof:** Any vertex cover must include at least one vertex of each edge in  $M$ .  $|M| \leq OPT_G$  follows from Lemma 1. □

Theorem 1 holds because the algorithm returns exactly  $2|M|$  vertices.

## The MAX-3SAT Problem

A **variable**: a boolean unknown  $x$  whose value is 0 or 1.

A **literal**: a variable  $x$  or its negation  $\bar{x}$ .

A **3-literal clause**: the OR of 3 literals with different variables.

$S$  = a set of clauses

$\mathcal{X}$  = the set of variables appearing in at least one clause of  $S$

A **truth assignment** of  $S$ : a function  $f: \mathcal{X} \rightarrow \{0, 1\}$ .

A truth assignment  $f$  **satisfies** a clause in  $S$  if the clause evaluates to 1 under  $f$ .

**The MAX-3SAT Problem:** Let  $S$  be a set of  $n$  clauses. Find a truth assignment of  $S$  to maximize the number of clauses satisfied.

**Example:**

$$S = \{x_1 \vee x_2 \vee x_3, \\ x_1 \vee x_2 \vee \bar{x}_3, \\ x_1 \vee \bar{x}_2 \vee x_3, \\ x_1 \vee \bar{x}_2 \vee \bar{x}_3, \\ \bar{x}_1 \vee x_3 \vee x_4, \\ \bar{x}_1 \vee x_3 \vee \bar{x}_4, \\ \bar{x}_1 \vee \bar{x}_3 \vee x_4, \\ \bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4\}.$$

$n = 8$  and  $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$ .

The truth assignment  $x_1 = x_2 = x_3 = x_4 = 1$  satisfies 7 clauses. It is impossible to satisfy 8.

The MAX-3SAT problem is NP-hard.

- No one has found an algorithm solving the problem in time polynomial in  $n$ .
- Such algorithms cannot exist if  $\mathcal{P} \neq \mathcal{NP}$ .

## Approximation Algorithms

$\mathcal{A}$  = an algorithm that, given any legal input  $S$ , returns a truth assignment of  $S$ .

$OPT_S$  = the largest number of clauses that a truth assignment of  $S$  can satisfy.

$Z_S$  = the number of clauses satisfied by the truth assignment  $\mathcal{A}$  returns.

- $Z_S$  is a random variable if  $\mathcal{A}$  is randomized. So, instead of talking about the exact value  $Z_S$ , it makes better sense to talk about statistical properties of  $Z_S$ . As usual, we focus on its expectation.

$\mathcal{A}$  is a **randomized  $\rho$ -approximate algorithm** for MAX-3SAT if  $E[Z_S] \geq \rho \cdot OPT_S$  holds for any legal input  $S$ .

The value  $\rho$  is the **approximation ratio**.

We also say that  $\mathcal{A}$  achieves an approximation ratio of  $\rho$  **in expectation**.

Consider the following algorithm.

**Input:** a set  $S$  of clauses with variable set  $\mathcal{X}$

**for** each variable  $x \in \mathcal{X}$  **do**

toss a fair coin

**if** the coin comes up heads **then**  $x \leftarrow 1$

**else**  $x \leftarrow 0$

It is clear that the algorithm runs in  $O(n)$  time.

Next, we show that the algorithm achieves an approximation ratio  $7/8$  in expectation.

**Theorem 2:** The algorithm produces a truth assignment that satisfies  $\frac{7}{8}n$  clauses in expectation.

**Proof:** First, we claim that it suffices to show that each clause is satisfied with probability  $7/8$ .

Think: Why is this true?

W.l.o.g., suppose that the clause is  $x_1 \vee x_2 \vee x_3$ . The clause is 0 if and only if  $x_1$ ,  $x_2$ , and  $x_3$  are all 0. The probability for  $x_1 = x_2 = x_3 = 0$  is  $1/8$ . □

Advertisement!

## Background:

- CUHK once had a strong presence in Theoretical Computer Science (TCS), supported by several faculty members and a well-rounded curriculum.
- However, after the departure of several TCS professors, the number of courses in the “Theory Stream” (a.k.a. Algorithm and Complexity Stream) has declined.
- Compared to top universities worldwide, our current curriculum is missing several key courses in theoretical computer science.

## Our Aspiration:

- The department is making efforts to diversify the faculty team by recruiting more professors, to provide better education for our undergraduates.
- We are introducing the following courses to strengthen the TCS curriculum:
  - **CSCI3350 Introduction to Quantum Computing** (offered next semester)
  - **CSCI3360 Introduction to Computational Complexity** (planned for the 2026–27 academic year)
  - **CSCI4240 Advanced Algorithms: Randomization and Approximation** (proposal submitted; expected to be available in 2026–27)
- We also plan to rename the stream from  
*“Algorithm and Complexity Stream”*  
to  
*“Theoretical Computer Science (TCS) Stream”*  
to better reflect its broader academic scope, and to make it more appealing to students :)

## Why These Courses Matter:

- They provide a rigorous foundation in algorithm design, complexity theory, and emerging models like quantum computing.
- These topics are essential for research, innovation, and understanding the limits of computation.
- A strong TCS background is highly valued in graduate programs and competitive tech careers.

**Instructor:** These courses will likely be taught (or co-taught) by Prof. Xiao Liang. If you've enjoyed **CSCI3160** this semester, you are strongly encouraged to go for these new TCS courses!