# Approximation Algorithms

Josh Wilcox (jw14g24)

April 29, 2025

# Table of Contents

# Overview

### The Need for Approximation

- Many important optimization problems are **NP-hard**, meaning we cannot expect to find efficient algorithms that always produce exact solutions in reasonable time.
- For such problems, we often seek **approximation algorithms** that efficiently find solutions close to the optimum.
- The quality of an approximation algorithm is measured by how close its solution is to the optimal one.

### Definition: $\alpha$-Approximation Algorithm

An algorithm is called an $\alpha$-approximation algorithm if, for every input instance, it produces a solution whose value is within a factor $\alpha$ of the optimal solution.

### Examples

- **Minimization Problem:**
  - Let $OPT(I)$ be the optimal (minimum) value for instance $I$.
  - Algorithm $A$ is an $\alpha$-approximation if, for all $I$:

$$OPT(I) \leq A(I) \leq \alpha \cdot OPT(I), \quad \alpha \geq 1$$

  - *Interpretation:* The solution found by $A$ is at most $\alpha$ times worse than the optimal.
- **Maximization Problem:**
  - Let $OPT(I)$ be the optimal (maximum) value for instance $I$.
  - Algorithm $A$ is an $\alpha$-approximation if, for all $I$:

$$\alpha \cdot OPT(I) \leq A(I) \leq OPT(I), \quad 0 < \alpha \leq 1$$

  - *Interpretation:* The solution found by $A$ is at least an $\alpha$ fraction of the optimal.

**Example:** If an algorithm for a minimization problem is a 2-approximation, then its solution will never be more than twice as large as the optimal solution.

# Load Balancing

We are given $n$ tasks with processing times $t_1, t_2, \ldots, t_n$. We want to distribute the tasks to run on $m$ machines such that the load is balanced. The load of a machine is the sum of the processing times of the tasks assigned to it. The goal is to minimize the makespan, i.e., the maximum load of any machine. Let $L(M_i)$ be the load of machine $M_i$, then the makespan is:

$$\max_{1 \leq i \leq m} L(M_i)$$

- This problem of minimizing the maximum load of any machine is NP-Hard.
- So we require an approximation to solve this problem efficiently.

### Lower Bounds for the Makespan

- *Average load:* The total work divided equally among all machines gives a lower bound:

$$\frac{1}{m} \sum_{j=1}^{n} t_j$$

- *Largest task:* The largest single task must be assigned to some machine:

$$\max_{1 \leq k \leq n} t_k$$

- *Therefore, the optimal makespan must be at least the maximum of these two:*

$$OPT(I) \geq \max \left\{ \frac{1}{m} \sum_{j=1}^{n} t_j, \ \max_{1 \leq k \leq n} t_k \right\}$$

### Greedy Approximation Algorithm

- Assign each task, in any order, to the machine with the smallest current load.
- This is simple and fast, but how good is it?

### Analysis of the Greedy Algorithm

- Let $M_x$ be the machine with the largest load when the algorithm finishes (i.e., the makespan).
- Let $J_y$ be the last task assigned to $M_x$, with processing time $t_y$.
- Let $L^*$ be the load of $M_x$ *before* $J_y$ was assigned.
- *Key observation:* When $J_y$ was assigned, $M_x$ had the smallest load among all machines.
- Therefore, before $J_y$ was assigned, every machine had load at least $L^*$.
- The total load assigned before $J_y$ is at most $\sum_{j=1}^{n} t_j$, so:

$$L^* \leq \frac{1}{m} \sum_{j=1}^{n} t_j$$

- The final load of $M_x$ is $L^* + t_y$, and $t_y \leq \max_{1 \leq i \leq n} t_i$.
- So, the makespan produced by greedy is:

$$G(I) = L^* + t_y \leq \frac{1}{m} \sum_{j=1}^{n} t_j + \max_{1 \leq i \leq n} t_i$$

- Recall our lower bound for $OPT(I)$ is the maximum of these two terms, so:

$$G(I) \leq 2 \cdot \max \left\{ \frac{1}{m} \sum_{j=1}^{n} t_j, \ \max_{1 \leq i \leq n} t_i \right\} \leq 2 \cdot OPT(I)$$

- **Conclusion:** The greedy algorithm is a 2-approximation algorithm for load balancing.

# Vertex Cover

**What is a vertex cover**

- A vertex cover is a set $C \subseteq V$ such that for every edge $(u, v)$, either $u \in C$ or $v \in C$.
- Every edge must be "covered" by at least one endpoint in $C$.
- Our goal is to find a vertex cover of minimum size.

### Approximation Algorithm

**Algorithm** Approximation Algorithm for Minimum Vertex Cover

**Require:** A graph $G = (V, E)$
**Ensure:** A vertex cover $C$ of $G$

**function** VC($G$)
    $C \leftarrow \emptyset$ ▷ Start with an empty vertex cover
    $E_c \leftarrow E$ ▷ Work with a copy of the edge set
    **while** $E_c \neq \emptyset$ **do**
        Select any edge $(u, v) \in E_c$ ▷ Pick an uncovered edge
        $C \leftarrow C \cup \{u, v\}$ ▷ Add both endpoints to the cover
        Remove all edges incident to $u$ or $v$ from $E_c$
    **end while**
    **return** $C$
**end function**

- At each step, we pick an uncovered edge and add *both* its endpoints to the cover.
- By removing all edges incident to these vertices, we ensure progress toward covering all edges.
- The algorithm is simple and fast, but may not always find the smallest possible cover.

### Why is this a 2-approximation?

- Let $C^*$ be an optimal (minimum) vertex cover.
- Each time the algorithm picks an edge $(u, v)$, *at least one* of $u$ or $v$ must be in $C^*$ (otherwise, $(u, v)$ would not be covered).
- The edges picked in each iteration are **disjoint** (no two share an endpoint), so each iteration "uses up" at least one unique vertex from $C^*$.
- If the algorithm picks $k$ edges, then $|C^*| \geq k$.
- The algorithm adds 2 vertices per edge, so $|C| = 2k$.
- Therefore, $|C| \leq 2|C^*|$.
- **Conclusion:** The algorithm always finds a vertex cover at most twice as large as the minimum possible.

*This is a classic example of a 2-approximation algorithm for an NP-hard problem.*