Advanced Algorithms

Approximation Algorithms

Coloring and scheduling problems

Alexander Wolff · WS22
Dealing with NP-Hard Optimization Problems

What should we do?

- Sacrifice optimality for speed
  - Heuristics
  - Approximation algorithms
- Optimal solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms
Dealing with NP-Hard Optimization Problems

What should we do?

- Sacrifice optimality for speed
  - Heuristics
  - Approximation algorithms

- Optimal solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms

- This lecture
  - Heuristics
  - Approximation algorithms

In this lecture, we focus on:

- Heuristics
- Approximation algorithms

These techniques allow us to find solutions quickly, even though they may not be optimal. This is particularly useful when dealing with NP-hard optimization problems, where finding an exact solution in a reasonable time is often impractical.
Approximation Algorithms

Problem.

- For NP-hard optimization problems, we cannot compute the optimal solution of every instance efficiently (unless \( P = NP \)).
- Heuristics offer no guarantee on the quality of their solutions.
Approximation Algorithms

Problem.
- For NP-hard optimization problems, we cannot compute the optimal solution of every instance efficiently (unless P = NP).
- Heuristics offer no guarantee on the quality of their solutions.

Goal.
- Design **approximation algorithms**: run in polynomial time and compute solutions of guaranteed quality.
- Study techniques for the design and analysis of approximation algorithms.
Approximation Algorithms

Problem.
■ For NP-hard optimization problems, we cannot compute the optimal solution of every instance efficiently (unless P = NP).
■ Heuristics offer no guarantee on the quality of their solutions.

Goal.
■ Design approximation algorithms:
  ■ run in polynomial time and
  ■ compute solutions of guaranteed quality.
■ Study techniques for the design and analysis of approximation algorithms.

Overview.
■ Approximation algorithms that compute solutions with/that are
  ■ additive guarantee,  ■ relative guarantee,  ■ “arbitrarily good”.
Approximation with Additive Guarantee

**Definition.**
Let $\Pi$ be an optimization problem, let $A$ be a polynomial-time algorithm for $\Pi$, let $I$ be an instance of $\Pi$, and let $\text{ALG}(I)$ be the value of the objective function of the solution that $A$ computes given $I$.

Then $A$ is called an **approximation algorithm with additive guarantee** $\delta$ if

$$|\text{OPT}(I) - \text{ALG}(I)| \leq \delta$$

for every instance $I$ of $\Pi$. 
Approximation with Additive Guarantee

Definition.
Let \( \Pi \) be an optimization problem, let \( \mathcal{A} \) be a polynomial-time algorithm for \( \Pi \), let \( I \) be an instance of \( \Pi \), and let \( \text{ALG}(I) \) be the value of the objective function of the solution that \( \mathcal{A} \) computes given \( I \).

Then \( \mathcal{A} \) is called an approximation algorithm with additive guarantee \( \delta \) if

\[
|\text{OPT}(I) - \text{ALG}(I)| \leq \delta
\]

for every instance \( I \) of \( \Pi \).

Most problems that we know do not admit an approximation algorithm with additive guarantee.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$. 

![Graph Diagram]
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum vertex coloring**, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum vertex coloring**, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A *minimum vertex coloring*, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

**GreedyVertexColoring(connected graph $G$)**
Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A minimum vertex coloring, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph $G$)
Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

Input. A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

Output. A minimum vertex coloring, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph $G$)
Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

**Input.** A graph \( G = (V, E) \). Let \( \Delta \) be the maximum degree of \( G \).

**Output.** A **minimum vertex coloring**, that is, an assignment of the vertices of \( G \) to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

**GreedyVertexColoring**(connected graph \( G \))

Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum vertex coloring**, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph $G$)

Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A *minimum vertex coloring*, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph $G$)

Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A *minimum vertex coloring*, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph $G$)

Color vertices in some order with the lowest feasible color.
Minimum Vertex Coloring

Input. A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

Output. A minimum vertex coloring, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

GreedyVertexColoring(connected graph G)
Color vertices in some order with the lowest feasible color.

Theorem 1.
The algorithm GreedyVertexColoring computes a vertex coloring with at most $\Delta + 1$ colors in $O(V + E)$ time. Hence, it has an additive approximation guarantee of $\Delta - 1$. 

\[
\text{Theorem 1.}\quad \text{The algorithm GreedyVertexColoring computes a vertex coloring with at most } \Delta + 1 \text{ colors in } O(V + E) \text{ time. Hence, it has an additive approximation guarantee of } \Delta - 1. 
\]
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A minimum vertex coloring, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

Theorem 1. The algorithm GreedyVertexColoring computes a vertex coloring with at most $\Delta + 1$ colors in $O(V + E)$ time. Hence, it has an additive approximation guarantee of $\Delta - 1$.
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A minimum vertex coloring, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

**GreedyVertexColoring** (connected graph $G$)

Color vertices in some order with the lowest feasible color.

**Theorem 1.**
The algorithm GreedyVertexColoring computes a vertex coloring with at most $\Delta + 1$ colors in $O(V + E)$ time. Hence, it has an additive approximation guarantee of $\Delta - 1$. 
Minimum Vertex Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A *minimum vertex coloring*, that is, an assignment of the vertices of $G$ to colors such that no two adjacent vertices get the same color and the number of colors is minimum.

- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

Theorem 1.
The algorithm GreedyVertexColoring computes a vertex coloring with at most $\Delta + 1$ colors in $O(V + E)$ time. Hence, it has an additive approximation guarantee of $\Delta - 1$. 

We can get $\Delta - 2$ if we return a 2-coloring whenever $G$ is bipartite.
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$. 

\[ \begin{tikzpicture}
    \node (v1) at (0,0) [shape=circle, fill=black] {};
    \node (v2) at (1,1) [shape=circle, fill=black] {};
    \node (v3) at (1,-1) [shape=circle, fill=black] {};
    \draw (v1) -- (v2);
    \draw (v1) -- (v3);
    \draw (v2) -- (v3);
\end{tikzpicture} \]
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A minimum edge coloring, that is, an assignment of colors to the edges of $G$ such that no two adjacent edges get the same color and the number of colors is minimum.
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum edge coloring**, that is, an assignment of colors to the edges of $G$ such that no two adjacent edges get the same color and the number of colors is minimum.

- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A minimum edge coloring, that is, an assignment of colors to the edges of $G$ such that now two adjacent edges get the same color and the number of colors is minimum.

- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.
- The minimum number of colors needed for an edge coloring of $G$ is called the **chromatic index** $\chi'(G)$.
- $\chi'(G)$ is lowerbounded by
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum edge coloring**, that is, an assignment of colors to the edges of $G$ such that now two adjacent edges get the same color and the number of colors is minimum.

- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.
- The minimum number of colors needed for an edge coloring of $G$ is called the **chromatic index** $\chi'(G)$.
- $\chi'(G)$ is lowerbounded by $\Delta$. 
Minimum Edge Coloring

**Input.** A graph $G = (V, E)$. Let $\Delta$ be the maximum degree of $G$.

**Output.** A **minimum edge coloring**, that is, an assignment of colors to the edges of $G$ such that now two adjacent edges get the same color and the number of colors is minimum.

- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.
- The minimum number of colors needed for an edge coloring of $G$ is called the **chromatic index** $\chi'(G)$.
- $\chi'(G)$ is lowerbounded by $\Delta$.
- We show that $\chi'(G) \leq \Delta + 1$. 

The minimum number of colors needed for an edge coloring of $G$ is called the chromatic index $\chi'(G)$. 

$\chi'(G)$ is lowerbounded by $\Delta$.

We show that $\chi'(G) \leq \Delta + 1$. 

Minimum Edge Coloring – Upper Bound

**Vizing’s Theorem.**
For every graph \( G = (V, E) \) with maximum degree \( \Delta \), it holds that \( \Delta \leq \chi'(G) \leq \Delta + 1 \).
Minimum Edge Coloring – Upper Bound

**Vizing’s Theorem.**
For every graph $G = (V, E)$ with maximum degree $\Delta$, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

**Proof** by induction on $m = |E|$.

- Base case $m = 1$ is trivial.
Minimum Edge Coloring – Upper Bound

**Vizing’s Theorem.**

For every graph $G = (V, E)$ with maximum degree $\Delta$, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

**Proof** by induction on $m = |E|$.

- **Base case** $m = 1$ is trivial.

Let $G$ be a graph on $m$ edges, and let $e = uv$ be an edge of $G$.

- **By induction**, $G - e$ has a $(\Delta(G - e) + 1)$-edge coloring.

Vadim G. Vizing  
(Kiew 1937 – 2017 Odessa)
Minimum Edge Coloring – Upper Bound

**Vizing’s Theorem.**
For every graph $G = (V, E)$ with maximum degree $\Delta$, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

**Proof** by induction on $m = |E|$.
- Base case $m = 1$ is trivial.

Let $G$ be a graph on $m$ edges, and let $e = uv$ be an edge of $G$.
- By induction, $G - e$ has a $(\Delta(G - e) + 1)$-edge coloring.
- If $\Delta(G) > \Delta(G - e)$, color $e$ with color $\Delta(G) + 1$. 

Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)
Minimum Edge Coloring – Upper Bound

Vizing’s Theorem.
For every graph $G = (V, E)$ with maximum degree $\Delta$, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

Proof by induction on $m = |E|$.

- Base case $m = 1$ is trivial.

Let $G$ be a graph on $m$ edges, and let $e = uv$ be an edge of $G$.
- By induction, $G - e$ has a $(\Delta(G - e) + 1)$-edge coloring.
- If $\Delta(G) > \Delta(G - e)$, color $e$ with color $\Delta(G) + 1$.
- If $\Delta(G) = \Delta(G - e)$, change the coloring such that $u$ and $v$ miss the same color $\alpha$.

Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)
Minimum Edge Coloring – Upper Bound

Vizing’s Theorem.
For every graph $G = (V, E)$ with maximum degree $\Delta$, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

Proof by induction on $m = |E|$.

- Base case $m = 1$ is trivial.

Let $G$ be a graph on $m$ edges, and let $e = uv$ be an edge of $G$.

- By induction, $G - e$ has a $(\Delta(G - e) + 1)$-edge coloring.
- If $\Delta(G) > \Delta(G - e)$, color $e$ with color $\Delta(G) + 1$.
- If $\Delta(G) = \Delta(G - e)$, change the coloring such that $u$ and $v$ miss the same color $\alpha$.
- Then color $e$ with $\alpha$.

Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color.
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u$, $v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

VizingRecoloring($G, c, u, \alpha_1$)

\[
i \leftarrow 1
\]
while $\exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\}$ do

\[
\begin{align*}
vi & \leftarrow w \\
\alpha_{i+1} & \leftarrow \text{min color missing at } w \\
i & \leftarrow i + 1
\end{align*}
\]
return $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color.

Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1)\\
i \leftarrow 1\\\text{while } \exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}\\\quad v_i \leftarrow w\\\quad \alpha_{i+1} \leftarrow \text{min color missing at } w\\\quad i \leftarrow i + 1\\\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]
**Minimum Edge Coloring – Recoloring**

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\begin{align*}
\text{VizingRecoloring}(G, c, u, \alpha_1) \quad &i \leftarrow 1 \\
\textbf{while} \; \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \; \textbf{do} \\
&v_i \leftarrow w \\
&\alpha_{i+1} \leftarrow \text{min color missing at } w \\
&i \leftarrow i + 1 \\
\textbf{return} \; v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\end{align*}
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u$, $v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is *missing* a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1) \\
i \leftarrow 1 \\
\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
\quad v_i \leftarrow w \\
\quad \alpha_{i+1} \leftarrow \min \text{ color missing at } w \\
\quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1)
\]

\[
i ← 1
\]
\[
\text{while } \exists w ∈ N(u): c(uw) = \alpha_i ∧ w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}
\]
\[
\begin{align*}
v_i & ← w \\
\alpha_{i+1} & ← \text{min color missing at } w \\
i & ← i + 1
\end{align*}
\]
\[
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color.
Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

```
\text{VizingRecoloring}(G, c, u, \alpha_1)
```

```plaintext
i ← 1
while \( \exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \) do
    \[ v_i \leftarrow w \]
    \[ \alpha_{i+1} \leftarrow \text{min color missing at } w \]
    \[ i \leftarrow i + 1 \]
return \( v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1} \)
```

Case 1: $u$ misses $\alpha_{h+1}$.
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let \( G \) be a graph with a \( (\Delta + 1) \)-edge coloring \( c \), let \( u, v \) be non-adjacent vertices with \( \text{deg}(u), \text{deg}(v) < \Delta \). Then \( c \) can be changed s.t. \( u \) and \( v \) miss the same color.

**Proof.** Note that every vertex is missing a color.
Let \( u \) miss \( \beta \) and \( v \) miss \( \alpha_1 \); apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1) \\
i \leftarrow 1 \\
\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
\quad v_i \leftarrow w \\
\quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
\quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]

**Case 1:** \( u \) misses \( \alpha_{h+1} \).

**Lemma 2.** Let \( G \) be a graph with a \( (\Delta + 1) \)-edge coloring \( c \), let \( u, v \) be non-adjacent vertices with \( \text{deg}(u), \text{deg}(v) < \Delta \). Then \( c \) can be changed s.t. \( u \) and \( v \) miss the same color.
Minimum Edge Coloring – Recoloring

Lemma 2.
Let \( G \) be a graph with a \((\Delta + 1)\)-edge coloring \( c \), let \( u, v \) be non-adjacent vertices with \( \deg(u), \deg(v) < \Delta \). Then \( c \) can be changed s.t. \( u \) and \( v \) miss the same color.

Proof. Note that every vertex is missing a color.
Let \( u \) miss \( \beta \) and \( v \) miss \( \alpha_1 \); apply the following algorithm:

\[
VizingRecoloring(G, c, u, \alpha_1) \\
i \leftarrow 1 \\
\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
\quad v_i \leftarrow w \\
\quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
\quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]

Lemma 2.
Let \( G \) be a graph with a \((\Delta + 1)\)-edge coloring \( c \), let \( u, v \) be non-adjacent vertices with \( \deg(u), \deg(v) < \Delta \). Then \( c \) can be changed s.t. \( u \) and \( v \) miss the same color.
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u$, $v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1)
\]

\[
i \leftarrow 1
\]

\[
\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}
\]

\[
\begin{align*}
& v_i \leftarrow w \\
& \alpha_{i+1} \leftarrow \text{min color missing at } w \\
& i \leftarrow i + 1
\end{align*}
\]

\[
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]

**Case 1: $u$ misses $\alpha_{h+1}$.**
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u$, $v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1) \quad \begin{align*}
i & \leftarrow 1 \\
\text{while } & \exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
& v_i \leftarrow w \\
& \alpha_{i+1} \leftarrow \text{min color missing at } w \\
& i \leftarrow i + 1 \\
\text{return } & v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\end{align*}
\]
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color.
Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

$\text{VizingRecoloring}(G, c, u, \alpha_1)$

$$i \leftarrow 1$$
$$\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}$$
$$v_i \leftarrow w$$
$$\alpha_{i+1} \leftarrow \text{min color missing at } w$$
$$i \leftarrow i + 1$$
$$\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$$

Case 1: $u$ misses $\alpha_{h+1}$. 

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.
Minimum Edge Coloring – Recoloring

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color.
Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

```
VizingRecoloring(G, c, u, \alpha_1)
```

\[
i \leftarrow 1
\]
\[
\text{while } \exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_i-1\} \text{ do}
\]
\[
\begin{align*}
    v_i & \leftarrow w \\
    \alpha_{i+1} & \leftarrow \min \text{ color missing at } w \\
    i & \leftarrow i + 1
\end{align*}
\]
\[
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\begin{algorithm}
\begin{align*}
i & \leftarrow 1 \\
\text{while } & \exists w \in N(u) : c(uw) = \alpha_i \text{ and } w \not\in \{v_1, \ldots, v_{i-1}\} \text{ do} \\
& \quad v_i \leftarrow w \\
& \quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
& \quad i \leftarrow i + 1 \\
\text{return } & v_1, \ldots, v_i, \alpha_1, \ldots, \alpha_{i+1}
\end{align*}
\end{algorithm}

Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$. 

**Proof.** Note that every vertex is **missing** a color.
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1)
\]
\[
i \leftarrow 1
\]
\[
\text{while } \exists w \in N(u) : c(uw) = \alpha_i \land w \not\in \{v_1, \ldots, v_{i-1}\} \text{ do}
\]
\[
\begin{align*}
&v_i \leftarrow w \\
&\alpha_{i+1} \leftarrow \text{min color missing at } w \\
&i \leftarrow i + 1
\end{align*}
\]
\[
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
\text{VizingRecoloring}(G, c, u, \alpha_1) \rightarrow
\begin{array}{l}
i \leftarrow 1 \\
\text{while} \ \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
\quad v_i \leftarrow w \\
\quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
\quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\end{array}
\]

**Case 2:** $\alpha_{h+1} = \alpha_j, j < h$. 

**Proof.**
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

**VizingRecoloring($G, c, u, \alpha_1$)**

1. $i \leftarrow 1$
2. while $\exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\}$ do
   1. $v_i \leftarrow w$
   2. $\alpha_{i+1} \leftarrow \text{min color missing at } w$
   3. $i \leftarrow i + 1$
3. return $v_1, \ldots, v_i, \alpha_1, \ldots, \alpha_{i+1}$
Minimum Edge Coloring – Recoloring

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

$$\text{VizingRecoloring}(G, c, u, \alpha_1)$$

\[
i \leftarrow 1 \\
\text{while } \exists w \in N(u): c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\} \text{ do} \\
\quad v_i \leftarrow w \\
\quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
\quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\]

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.
Minimum Edge Coloring – Recoloring

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u$, $v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color.

Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

VizingRecoloring($G, c, u, \alpha_1$)

$$i \leftarrow 1$$

while $\exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\}$ do

$$v_i \leftarrow w$$
$$\alpha_{i+1} \leftarrow \text{min color missing at } w$$
$$i \leftarrow i + 1$$

return $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

Case 2: $\alpha_{h+1} = \alpha_j, j < h$. 

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is missing a color.
Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[
VizingRecoloring(G, c, u, \alpha_1)
\]

\[
i \leftarrow 1
\]

\[
\text{while } \exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}
\]

\[
\begin{align*}
    v_i &\leftarrow w \\
    \alpha_{i+1} &\leftarrow \text{min color missing at } w \\
    i &\leftarrow i + 1
\end{align*}
\]

\[
\text{return } v_1, \ldots, v_i, \alpha_1, \ldots, \alpha_{i+1}
\]
Minimum Edge Coloring – Recoloring

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\text{deg}(u), \text{deg}(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\begin{align*}
\text{VizingRecoloring}(G, c, u, \alpha_1) & \\
i & \leftarrow 1 \\
\text{while } \exists w \in N(u) : c(uw) = \alpha_i \land w \not\in \{v_1, \ldots, v_{i-1}\} \text{ do} & \\
& \quad v_i \leftarrow w \\
& \quad \alpha_{i+1} \leftarrow \text{min color missing at } w \\
& \quad i \leftarrow i + 1 \\
\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}
\end{align*}
Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

Proof. Note that every vertex is missing a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

$$\text{VizingRecoloring}(G, c, u, \alpha_1)$$

$$i \leftarrow 1$$

$$\text{while } \exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\} \text{ do}$$

$$v_i \leftarrow w$$

$$\alpha_{i+1} \leftarrow \text{min color missing at } w$$

$$i \leftarrow i + 1$$

$$\text{return } v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$$

Case 2: $\alpha_{h+1} = \alpha_j, j < h$. 

Lemma 2.
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.
Minimum Edge Coloring – Recoloring

**Lemma 2.**
Let $G$ be a graph with a $(\Delta + 1)$-edge coloring $c$, let $u, v$ be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$. Then $c$ can be changed s.t. $u$ and $v$ miss the same color.

**Proof.** Note that every vertex is **missing** a color. Let $u$ miss $\beta$ and $v$ miss $\alpha_1$; apply the following algorithm:

\[ \text{VizingRecoloring}(G, c, u, \alpha_1) \]

\[
i \leftarrow 1
\]

**while** $\exists w \in N(u): c(uw) = \alpha_i \land w \notin \{v_1, \ldots, v_{i-1}\}$ **do**

\[
v_i \leftarrow w
\]

\[
\alpha_{i+1} \leftarrow \text{min color missing at } w
\]

\[
i \leftarrow i + 1
\]

**return** $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

**Case 2:** $\alpha_{h+1} = \alpha_j$, $j < h$.
Minimum Edge Coloring – Recoloring

Proof continued for Case 2: \( \alpha_{h+1} = \alpha_j, \ j < h, \)
and we need to find a color for edge \( uv_j. \)
Minimum Edge Coloring – Recoloring

Proof continued for Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$, and we need to find a color for edge $uv_j$.

Consider subgraph $G'$ of $G$ induced by the edges of colors $\beta$ and $\alpha_j$. 

$\Box$
Minimum Edge Coloring – Recoloring

**Proof** continued for Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$, and we need to find a color for edge $uv_j$.

- Consider subgraph $G'$ of $G$ induced by the edges of colors $\beta$ and $\alpha_j$.
- Since $\Delta(G') \leq 2$, we can recolor components.
Minimum Edge Coloring – Recoloring

**Proof** continued for Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$, and we need to find a color for edge $uv_j$.

- Consider subgraph $G'$ of $G$ induced by the edges of colors $\beta$ and $\alpha_j$.

- Since $\Delta(G') \leq 2$, we can recolor components.
Minimum Edge Coloring – Recoloring

**Proof** continued for Case 2: \( \alpha_{h+1} = \alpha_j, j < h \), and we need to find a color for edge \( uv_j \).

- Consider subgraph \( G' \) of \( G \) induced by the edges of colors \( \beta \) and \( \alpha_j \).
- Since \( \Delta(G') \leq 2 \), we can recolor components.
- Nodes \( u, v_j, v_h \) are all leaves in \( G' \).
  \[ \implies \] They are not all in the same component of \( G' \).
Minimum Edge Coloring – Recoloring

Proof continued for Case 2: \( \alpha_{h+1} = \alpha_j, j < h \), and we need to find a color for edge \( uv_j \).

- Consider subgraph \( G' \) of \( G \) induced by the edges of colors \( \beta \) and \( \alpha_j \).

- Since \( \Delta(G') \leq 2 \), we can recolor components.

- Nodes \( u, v_j, v_h \) are all leaves in \( G' \).
  \( \Rightarrow \) They are not all in the same component of \( G' \).

- If \( u \) and \( v_j \) are not in the same component:
  - recolor component ending at \( v_j \),

\[  \]
Minimum Edge Coloring – Recoloring

Proof continued for Case 2: \( \alpha_{h+1} = \alpha_j, j < h \), and we need to find a color for edge \( uv_j \).

- Consider subgraph \( G' \) of \( G \) induced by the edges of colors \( \beta \) and \( \alpha_j \).
- Since \( \Delta(G') \leq 2 \), we can recolor components.
- Nodes \( u, v_j, v_h \) are all leaves in \( G' \).
  \( \Rightarrow \) They are not all in the same component of \( G' \).
- If \( u \) and \( v_j \) are not in the same component:
  - recolor component ending at \( v_j \),
  - \( v_j \) now misses \( \beta \);
Minimum Edge Coloring – Recoloring

**Proof** continued for Case 2: \( \alpha_{h+1} = \alpha_j, j < h, \)
and we need to find a color for edge \( uv_j. \)

- Consider subgraph \( G' \) of \( G \) induced by
  the edges of colors \( \beta \) and \( \alpha_j. \)

- Since \( \Delta(G') \leq 2, \) we can recolor components.

- Nodes \( u, v_j, v_h \) are all leaves in \( G'. \)
  \( \Rightarrow \) They are not all in the same component of \( G'. \)

- If \( u \) and \( v_j \) are not in the same component:
  - recolor component ending at \( v_j, \)
  - \( v_j \) now misses \( \beta; \)
  - color \( uv_j \) with \( \beta. \)
Minimum Edge Coloring – Recoloring

**Proof** continued for Case 2: \( \alpha_{h+1} = \alpha_j, \ j < h \), and we need to find a color for edge \( uv_j \).

- Consider subgraph \( G' \) of \( G \) induced by the edges of colors \( \beta \) and \( \alpha_j \).
- Since \( \Delta(G') \leq 2 \), we can recolor components.
- Nodes \( u, v_j, v_h \) are all leaves in \( G' \).
  \( \Rightarrow \) They are not all in the same component of \( G' \).
- If \( u \) and \( v_j \) are not in the same component:
  - recolor component ending at \( v_j \),
  - \( v_j \) now misses \( \beta \);
  - color \( uv_j \) with \( \beta \).
- What if \( u \) and \( v_j \) are in the same component?
Minimum Edge Coloring – Algorithm

VizingEdgeColoring(graph G, coloring c ≡ 0)

if $E(G) \neq \emptyset$ then
  Let $e = uv$ be an arbitrary edge of $G$.
  $G_e \leftarrow G - e$
  VizingEdgeColoring($G_e$, $c$)
  if $\Delta(G_e) < \Delta(G)$ then
    Color $e$ with lowest free color.
  else
    Recolor $G_e$ as in Lemma 2.
    Color $e$ with color now missing at $u$ and $v$. 

Minimum Edge Coloring – Algorithm

VizingEdgeColoring(graph \( G \), coloring \( c \equiv 0 \))

\[
\text{if } E(G) \neq \emptyset \text{ then} \\
\quad \text{Let } e = uv \text{ be an arbitrary edge of } G. \\
\quad G_e \leftarrow G - e \\
\quad \text{VizingEdgeColoring}(G_e, c) \\
\text{if } \Delta(G_e) < \Delta(G) \text{ then} \\
\quad \quad \text{Color } e \text{ with lowest free color.} \\
\text{else} \\
\quad \quad \text{Recolor } G_e \text{ as in Lemma 2.} \\
\quad \quad \text{Color } e \text{ with color now missing at } u \text{ and } v.
\]

**Theorem 4.**

VizingEdgeColoring is an approximation algorithm with additive approximation guarantee

\[ \text{ALG}(G) - \text{OPT}(G) \leq 1. \]
Approximation with Relative Factor

- An additive approximation guarantee can rarely be achieved; but sometimes, there is a multiplicative approximation!
Approximation with Relative Factor

- An additive approximation guarantee can rarely be achieved; but sometimes, there is a multiplicative approximation!

**Definition.**
Let \( \Pi \) be a minimization problem, and let \( \alpha \in \mathbb{Q}^+ \).
A **factor-\( \alpha \)** approximation algorithm for \( \Pi \) is a polynomial-time algorithm \( \mathcal{A} \) that computes, for every instance \( I \) of \( \Pi \), a solution of value \( \text{ALG}(I) \) such that

\[
\frac{\text{ALG}(I)}{\text{OPT}(I)} \leq \alpha.
\]

We call \( \alpha \) the **approximation factor** of \( \mathcal{A} \).
Approximation with Relative Factor

An additive approximation guarantee can rarely be achieved; but sometimes, there is a multiplicative approximation!

**Definition. maximization**

Let $\Pi$ be a minimization problem, and let $\alpha \in \mathbb{Q}^+$. A **factor-$\alpha$ approximation algorithm** for $\Pi$ is a polynomial-time algorithm $A$ that computes, for every instance $I$ of $\Pi$, a solution of value $\text{ALG}(I)$ such that

$$\frac{\text{ALG}(I)}{\text{OPT}(I)} \leq \alpha.$$

We call $\alpha$ the **approximation factor** of $A$. 

**Definition.**
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$. 

![Graph](image)
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$. 

![Diagram of a complete graph with vertices u, v, and w connected by edges]
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**

- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**

- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V: d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

Input. Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

Output. A shortest Hamiltonian cycle in $G$.

Algorithm.
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
2-Approximation for Metric TSP (from AGT)

Input. Complete graph $G = (V, E)$ and a distance function
$d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality,
i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

Output. A shortest Hamiltonian cycle in $G$.

Algorithm.
- Compute MST.
- Double edges.
- Walk along tree,
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices
- skpping visited vertices
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V: d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices
- skipping visited vertices
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
- skipping visited vertices
- and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V: d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
  - skipping visited vertices
- and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d : E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree,
  - skipping visited vertices
- and adding shortcuts.
2-Approximation for Metric TSP (from AGT)

**Input.** Complete graph $G = (V, E)$ and a distance function $d: E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V: d(u, w) \leq d(u, v) + d(v, w)$.

**Output.** A shortest Hamiltonian cycle in $G$.

**Algorithm.**
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.

**Theorem 5.** The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.
2-Approximation for Metric TSP (from AGT)

Input. Complete graph $G = (V, E)$ and a distance function $d: E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V: d(u, w) \leq d(u, v) + d(v, w)$.

Output. A shortest Hamiltonian cycle in $G$.

Algorithm.
- Compute MST.
- Double edges.
- Walk along tree, skipping visited vertices and adding shortcuts.

Proof. The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.

$\text{ALG} \leq d(\text{cycle}) = 2d(\text{MST}) \leq 2\text{OPT}$.

Theorem 5.

The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say $i$ and $k$.
Set tour $T$ to go from $i$ to $k$ to $i$ (clockwise).

while $T \subsetneq V$ do
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let $k$ be vertex after $i$ in $T$.
  Add $j$ between $i$ and $k$. 
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say \( i \) and \( k \).
Set tour \( T \) to go from \( i \) to \( k \) to \( i \) (clockwise).

\[
\text{while } T \subseteq V \text{ do}
\]

Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \( d(i, j) \).
Let \( k \) be vertex after \( i \) in \( T \).
Add \( j \) between \( i \) and \( k \).
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm\((G = (V, E), d)\)

Find closest pair, say \(i\) and \(k\).
Set tour \(T\) to go from \(i\) to \(k\) to \(i\) (clockwise).

\[
\text{while } T \subsetneq V \text{ do}
\]

Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \(d(i, j)\).
Let \(k\) be vertex after \(i\) in \(T\).
Add \(j\) between \(i\) and \(k\).
Nearest Addition Algorithm for Metric TSP

```
NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while T ⊊ V do
  Find pair (i, j) ∈ T × (V \ T) minimizing d(i, j).
  Let k be vertex after i in T.
  Add j between i and k.
```
Nearest Addition Algorithm for Metric TSP

\[
\text{NearestAdditionAlgorithm}(G = (V, E), d)
\]

Find closest pair, say \( i \) and \( k \).
Set tour \( T \) to go from \( i \) to \( k \) to \( i \) (clockwise).
while \( T \subset V \) do

- Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \( d(i, j) \).
- Let \( k \) be vertex after \( i \) in \( T \).
- Add \( j \) between \( i \) and \( k \).
Nearest Addition Algorithm for Metric TSP

\[ \text{NearestAdditionAlgorithm}(G = (V, E), d) \]

Find closest pair, say \( i \) and \( k \).
Set tour \( T \) to go from \( i \) to \( k \) to \( i \) (clockwise).

\[ \text{while } T \subseteq V \text{ do} \]

Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \( d(i, j) \).
Let \( k \) be vertex after \( i \) in \( T \).
Add \( j \) between \( i \) and \( k \).
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while T ⊊ V do
    Find pair (i, j) ∈ T × (V \ T) minimizing d(i, j).
    Let k be vertex after i in T.
    Add j between i and k.
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).

while T ⊊ V do
  Find pair (i, j) ∈ T × (V \ T) minimizing d(i, j).
  Let k be vertex after i in T.
  Add j between i and k.
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm(G = (V, E), d)

Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).

while T ⊊ V do
  Find pair (i, j) ∈ T × (V \ T) minimizing d(i, j).
  Let k be vertex after i in T.
  Add j between i and k.
Nearest Addition Algorithm for Metric TSP

\[
\text{NearestAdditionAlgorithm}(G = (V, E), d)
\]

Find closest pair, say \(i\) and \(k\).
Set tour \(T\) to go from \(i\) to \(k\) to \(i\) (clockwise).

\[
\text{while } T \subsetneq V \text{ do}
\]

- Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \(d(i, j)\).
- Let \(k\) be vertex after \(i\) in \(T\).
- Add \(j\) between \(i\) and \(k\).
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm\((G = (V, E), d)\)

Find closest pair, say \(i\) and \(k\).
Set tour \(T\) to go from \(i\) to \(k\) to \(i\) (clockwise).

\[
\text{while } T \subsetneq V \text{ do}
\]

Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \(d(i, j)\).
Let \(k\) be vertex after \(i\) in \(T\).
Add \(j\) between \(i\) and \(k\).

Theorem 6.
NearestAdditionAlgorithm is a 2-approximation algorithm for metric TSP.
Nearest Addition Algorithm for Metric TSP

NearestAdditionAlgorithm\((G = (V, E), d)\)

Find closest pair, say \(i\) and \(k\).
Set tour \(T\) to go from \(i\) to \(k\) to \(i\) (clockwise).

while \(T \subseteq V\) do
  Find pair \((i, j) \in T \times (V \setminus T)\) minimizing \(d(i, j)\).
  Let \(k\) be vertex after \(i\) in \(T\).
  Add \(j\) between \(i\) and \(k\).

Theorem 6.
NearestAdditionAlgorithm is a 2-approximation algorithm for metric TSP.

Proof.
- Exercise.
- Hints: MST and Prim’s algorithm.
Approximation Schemes

- In some cases, we can get arbitrarily good approximations.
Approximation Schemes

- In some cases, we can get arbitrarily good approximations.

**Definition.**
Let \( \Pi \) be a minimization problem. An algorithm \( \mathcal{A} \) is called a **polynomial-time approximation scheme (PTAS)** if \( \mathcal{A} \) computes, for every input \((I, \varepsilon)\) (consisting of an instance \(I\) of \( \Pi \) and a real \( \varepsilon > 0 \)), a value \( \text{ALG}(I) \) such that:

- \( \text{ALG}(I) \leq (1 + \varepsilon) \cdot \text{OPT}(I) \), and
- the runtime of \( \mathcal{A} \) is polynomial in \(|I|\) for every \( \varepsilon > 0 \).
Approximation Schemes

- In some cases, we can get arbitrarily good approximations.

**Definition.** Let $\Pi$ be a minimization problem. An algorithm $\mathcal{A}$ is called a **polynomial-time approximation scheme (PTAS)** if $\mathcal{A}$ computes, for every input $(I, \varepsilon)$ (consisting of an instance $I$ of $\Pi$ and a real $\varepsilon > 0$), a value $\text{ALG}(I)$ such that:

- $\text{ALG}(I) \leq (1 + \varepsilon) \cdot \text{OPT}(I)$, and
- the runtime of $\mathcal{A}$ is polynomial in $|I|$ for every $\varepsilon > 0$. 

**maximization**

Let $\Pi$ be a minimization problem. An algorithm $\mathcal{A}$ is called a polynomial-time approximation scheme (PTAS) if $\mathcal{A}$ computes, for every input $(I, \varepsilon)$ (consisting of an instance $I$ of $\Pi$ and a real $\varepsilon > 0$), a value $\text{ALG}(I)$ such that:

$\text{ALG}(I) \geq (1 - \varepsilon)$

- $\text{ALG}(I) \leq (1 + \varepsilon) \cdot \text{OPT}(I)$, and
- the runtime of $\mathcal{A}$ is polynomial in $|I|$ for every $\varepsilon > 0$. 

Approximation Schemes

- In some cases, we can get arbitrarily good approximations.

**Definition.** Let Π be a minimization problem. An algorithm A is called a polynomial-time approximation scheme (PTAS) if A computes, for every input (I, ε) (consisting of an instance I of Π and a real ε > 0), a value ALG(I) such that:

\[ ALG(I) \leq (1 + \varepsilon) \cdot OPT(I), \]

- the runtime of A is polynomial in |I| for every ε > 0.

A is called a fully polynomial-time approximation scheme (FPTAS) if it runs in time polynomial in |I| and 1/ε.
Approximation Schemes

- In some cases, we can get arbitrarily good approximations.

**Definition.** A minimization problem $\Pi$ is called a polynomial-time approximation scheme (PTAS) if $A$ computes, for every input $(I, \varepsilon)$ (consisting of an instance $I$ of $\Pi$ and a real $\varepsilon > 0$), a value $\text{ALG}(I)$ such that:

$$\text{ALG}(I) \leq (1 + \varepsilon) \cdot \text{OPT}(I),$$

- the runtime of $A$ is polynomial in $|I|$ for every $\varepsilon > 0$.

$A$ is called a fully polynomial-time approximation scheme (FPTAS) if it runs in time polynomial in $|I|$ and $1/\varepsilon$.

**Examples.**

- $O\left(n^2 + n^{\frac{1}{\varepsilon}}\right)$
- $O\left(n^2 \cdot 3^{\frac{1}{\varepsilon}}\right)$
- $O\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right)$
Approximation Schemes

- In some cases, we can get arbitrarily good approximations.

**Definition. maximization**

Let \( \Pi \) be a minimization problem. An algorithm \( \mathcal{A} \) is called a **polynomial-time approximation scheme (PTAS)** if \( \mathcal{A} \) computes, for every input \((I, \varepsilon)\) (consisting of an instance \(I\) of \( \Pi \) and a real \( \varepsilon > 0 \)), a value \( \text{ALG}(I) \) such that:

\[
\geq (1 - \varepsilon)
\]

- \( \text{ALG}(I) \leq (1 + \varepsilon) \cdot \text{OPT}(I) \), and
- the runtime of \( \mathcal{A} \) is polynomial in \(|I|\) for every \( \varepsilon > 0 \).

\( \mathcal{A} \) is called a **fully polynomial-time approximation scheme (FPTAS)** if it runs in time polynomial in \(|I|\) and \(1/\varepsilon\).

**Examples.**

- \( \mathcal{O}(n^2 + \frac{n}{\varepsilon}) \) ⇒ PTAS but not FPTAS
- \( \mathcal{O}\left(n^2 \cdot \frac{3}{\varepsilon}\right) \) ⇒ PTAS but not FPTAS
- \( \mathcal{O}\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right) \) ⇒ FPTAS
Multiprocessor Scheduling

**Input.**

- $n$ jobs $J_1, \ldots, J_n$ with durations $p_1, \ldots, p_n$.

- $m$ identical machines ($m < n$)
Multiprocessor Scheduling

**Input.**
- $n$ jobs $J_1, \ldots, J_n$ with durations $p_1, \ldots, p_n$.
- $m$ identical machines ($m < n$)

**Output.**
Assignment of jobs to machines such that the time when all jobs have been processed is minimum. This is called the **makespan** of the assignment.
Multiprocessor Scheduling

**Input.**  
- $n$ jobs $J_1, \ldots, J_n$ with durations $p_1, \ldots, p_n$.

**Output.**  
Assignment of jobs to machines such that the time when all jobs have been processed is minimum. This is called the **makespan** of the assignment.

**Input.**  
- $m$ identical machines ($m < n$)

**Output.**  
- makespan
Multiprocessor Scheduling

**Input.**
- $n$ jobs $J_1, \ldots, J_n$ with durations $p_1, \ldots, p_n$.

**Output.**
- Assignment of jobs to machines such that the time when all jobs have been processed is minimum. This is called the **makespan** of the assignment.

**Example Diagram:**
- $n = 7$ jobs $J_1, J_2, J_3, J_4, J_5, J_6, J_7$ with durations $p_1, p_2, p_3, p_4, p_5, p_6, p_7$.
- $m = 3$ identical machines.

**Makespan Diagram:**
- The makespan is the maximum completion time of any job.

**Equation:**
- Makespan = Maximum completion time of any job.
Multiprocessor Scheduling

**Input.**  
- *n* jobs $J_1, \ldots, J_n$ with durations $p_1, \ldots, p_n$.

**Output.**  
Assignment of jobs to machines such that the time when all jobs have been processed is minimum. This is called the makespan of the assignment.

- *m* identical machines ($m < n$)

- Multiprocessor scheduling is NP-hard.
Multiprocessor Scheduling – List Scheduling

**LISTSCHEDULING**($J_1, \ldots, J_n, m$)

- Put the first $m$ jobs on the $m$ machines.
- Put the next job on the first free machine.

**Example.**

```
\begin{align*}
&\text{p}_1 [ J_1 ] \quad \text{p}_2 [ J_2 ] \quad \text{p}_3 [ J_3 ] \\
&\text{p}_4 [ J_4 ] \quad \text{p}_5 [ J_5 ] \quad \text{p}_6 [ J_6 ] \\
&\text{p}_7 [ J_7 ]
\end{align*}
```
Multiprocessor Scheduling – List Scheduling

**LISTSCHEDULING**$(J_1, \ldots, J_n, m)$

- Put the first $m$ jobs on the $m$ machines.
- Put the next job on the first free machine.

Example.

```
J_1 \rightarrow p_1
J_2 \rightarrow p_2
J_3 \rightarrow p_3
J_4 \rightarrow p_4
J_5 \rightarrow p_5
J_6 \rightarrow p_6
J_7 \rightarrow p_7
```
Multiprocessor Scheduling – List Scheduling

**LISTSCHEDULING**(*J_1*, ..., *J_n*, *m*)

Put the first *m* jobs on the *m* machines.

Put the next job on the first free machine.

**Example.**
Multiprocessor Scheduling – List Scheduling

\textbf{LISTSCHEDULING}(J_1, \ldots, J_n, m)

- Put the first $m$ jobs on the $m$ machines.
- Put the next job on the first free machine.

Example.
Multiprocessor Scheduling – List Scheduling

\textsc{ListScheduling}(J_1, \ldots, J_n, m)

Put the first \(m\) jobs on the \(m\) machines.

\textbf{Put the next job on the first free machine.}

Example.

\begin{itemize}
  \item \(J_1\) on \(p_1\)
  \item \(J_2\) on \(p_2\)
  \item \(J_3\) on \(p_3\)
  \item \(J_4\) on \(p_4\)
  \item \(J_5\) on \(p_5\)
  \item \(J_6\) on \(p_6\)
  \item \(J_7\) on \(p_7\)
\end{itemize}
**Multiprocessor Scheduling – List Scheduling**

**LISTSCHEDULING** ($J_1, \ldots, J_n, m$)

Put the first $m$ jobs on the $m$ machines.
Put the next job on the first free machine.

**Example.**

- **LISTSCHEDULING runs in** $O(n \log m)$ time.
Multiprocessor Scheduling – List Scheduling

**LISTSCHEDULING**($J_1, \ldots, J_n, m$)

Put the first $m$ jobs on the $m$ machines.
Put the next job on the first free machine.

Example.

- **LISTSCHEDULING** runs in $O(n \log m)$ time.
Multiprocessor Scheduling – List Scheduling

**LISTSCHEDULING**($J_1, \ldots, J_n, m$)

Put the first $m$ jobs on the $m$ machines. Put the next job on the first free machine.

**Example.**

![Diagram of jobs and machines](image)

- **LISTSCHEDULING** runs in $O(n \log m)$ time.

**Theorem 7.**

LISTSCHEDULING is a factor-($2 - \frac{1}{m}$) approximation algorithm.
**Multiprocessor Scheduling – List Scheduling**

ListScheduling($J_1, \ldots, J_n, m$)

Put the first $m$ jobs on the $m$ machines.
Put the next job on the first free machine.

**Example.**

```
\[ \begin{array}{c}
J_1 & J_2 & J_3 & J_4 \\
p_1 & p_2 & p_3 & p_4 \\
J_5 & J_6 & J_7 \\
p_5 & p_6 & p_7 \\
\end{array} \]
```

- **ListScheduling** runs in $O(n \log m)$ time.
- **Theorem 7.**
  ListScheduling is a factor-$(2 - \frac{1}{m})$ approximation algorithm.
Multiprocessor Scheduling – List scheduling (proof)

**Proof.** Let \( J_k = (S_k, T_k) \) be the last job, that is, \( T_k \) determines the makespan.

**Theorem 7.** ListScheduling is a \((2 - \frac{1}{m})\)-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.

$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$$

weight of all jobs but $J_k$

evenly distributed on $m$ machines

\[ T_k = \text{Makespan} \]
Multiprocessor Scheduling – List scheduling (proof)

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.

\[ S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \]

- For the optimal makespan $T_{\text{OPT}}$, we have:

\[ T_{\text{OPT}} \geq p_k \]

---

**Theorem 7.** ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.
  
  \[
  S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \quad \text{weight of all jobs but } J_k \quad \text{evenly distributed on } m \text{ machines}
  \]

- For the optimal makespan $T_{OPT}$, we have:
  
  \[
  T_{OPT} \geq p_k
  \]
  \[
  T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \quad \text{weight of all jobs} \quad \text{evenly distributed}
  \]

**Theorem 7.** ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

**ListScheduling**($J_1, \ldots, J_n, m$)

- Put the first $m$ jobs on the $m$ machines.
- Put the next job on the first free machine.

**Theorem 7.** ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.
  
  $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \text{ weight of all jobs but } J_k \text{ evenly distributed on } m \text{ machines}$

- For the optimal makespan $T_{OPT}$, we have:
  
  - $T_{OPT} \geq p_k$
  
  - $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \text{ weight of all jobs evenly distributed}$

- Hence:
  
  $T_k = S_k + p_k$

- For the optimal makespan $T_{OPT}$, we have:

  - $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \text{ weight of all jobs evenly distributed}$
**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.

$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \quad \text{weight of all jobs but } J_k \quad \text{evenly distributed on } m \text{ machines}$$

- For the optimal makespan $T_{OPT}$, we have:

$$T_{OPT} \geq p_k$$

$$T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \quad \text{weight of all jobs} \quad \text{evenly distributed}$$

- Hence:

$$T_k = S_k + p_k \\ \leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k$$

**Theorem 7.** ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.

  $$ S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \text{ weight of all jobs but } J_k \text{ evenly distributed on } m \text{ machines} $$

- For the optimal makespan $T_{OPT}$, we have:

  $$ T_{OPT} \geq p_k $$

  $$ T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \text{ weight of all jobs evenly distributed} $$

- Hence:

  $$ T_k = S_k + p_k $$

  $$ \leq \frac{1}{m} \sum_{i \neq k} p_i + p_k $$

  $$ = \frac{1}{m} \sum_{i=1}^{n} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k $$

**Theorem 7.**

ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

- No machine idles at time $S_k$.

$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \text{ weight of all jobs but } J_k$$

- For the optimal makespan $T_{OPT}$, we have:

  - $T_{OPT} \geq p_k$
  - $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \text{ weight of all jobs evenly distributed}$

- Hence:

  $$T_k = S_k + p_k \leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k$$

  $$= \frac{1}{m} \cdot \sum_{i=1}^{n} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k$$

  $$\leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}$$

**Theorem 7.** ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.
Multiprocessor Scheduling – List scheduling (proof)

**Theorem 7.** ListScheduling is a \((2 - \frac{1}{m})\)-approximation alg.

**Proof.** Let \(J_k = (S_k, T_k)\) be the last job, that is, \(T_k\) determines the makespan.

- No machine idles at time \(S_k\).
  \[
  S_k \leq \frac{1}{m} \sum_{i \neq k} p_i \quad \text{weight of all jobs but } J_k \text{ evenly distributed on } m \text{ machines}
  \]

- For the optimal makespan \(T_{OPT}\), we have:
  
  \[
  T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i \quad \text{weight of all jobs evenly distributed}
  \]

- Hence:
  
  \[
  T_k = S_k + p_k \\
  \leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
  = \frac{1}{m} \cdot \sum_{i=1}^{n} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
  \leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT} \\
  = \left(2 - \frac{1}{m}\right) \cdot T_{OPT}
  \]

**ListScheduling**

- Put the first \(m\) jobs on the \(m\) machines.
- Put the next job on the first free machine.
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use $\text{LISTSCHEDULING}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$. 
Multiprocessor Scheduling – PTAS

For a constant $\ell$ (1 $\leq$ $\ell$ $\leq$ $n$) define the algorithm $A_{\ell}$ as follows.

$A_{\ell}(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use LISTSCHEDULING for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

jobs

Example.

$\ell = 6$
For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use $\text{LISTSCHEDULING}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.
$\ell = 6$

sorted jobs
Multiprocessor Scheduling – PTAS

For a constant \(1 \leq \ell \leq n\) define the algorithm \(A_\ell\) as follows.

\[
A_\ell(J_1, \ldots, J_n, m)
\]

Sort jobs in descending order of runtime.

Schedule the \(\ell\) longest jobs \(J_1, \ldots, J_\ell\) optimally.

Use ListScheduling for the remaining jobs \(J_{\ell+1}, \ldots, J_n\).

Example.

\(\ell = 6\)

sorted jobs
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$
- Sort jobs in descending order of runtime.
- Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
- Use LISTSCHEDULING for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs

Example.
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs

Example.

$\ell = 6$

sorted jobs
Multiprocessor Scheduling – PTAS

For a constant $\ell$ \((1 \leq \ell \leq n)\) define the algorithm $A_\ell$ as follows.

$$A_\ell(J_1, \ldots, J_n, m)$$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

**Example.**

$\ell = 6$

sorted jobs

![Job Scheduling Diagram](image)
Multiprocessor Scheduling – PTAS

For a constant $\ell \ (1 \leq \ell \leq n)$ define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$
- Sort jobs in descending order of runtime.
- Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
- Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

**Example.**

$\ell = 6$

Sorted jobs

$M_4$ $M_3$ $M_2$ $M_1$

$J_1$ $J_2$ $J_3$ $J_4$ $J_5$ $J_6$
For a constant \( \ell \) (1 \( \leq \) \( \ell \) \( \leq \) \( n \)) define the algorithm \( A_\ell \) as follows.

\[
A_\ell(J_1, \ldots, J_n, m)
\]

Sort jobs in descending order of runtime.
Schedule the \( \ell \) longest jobs \( J_1, \ldots, J_\ell \) optimally.
Use \textsc{ListScheduling} for the remaining jobs \( J_{\ell+1}, \ldots, J_n \).

Example.
\( \ell = 6 \)

sorted jobs

\[
\begin{array}{cccccccc}
M_4 & j_1 & & & & & & \\
M_3 & j_2 & & j_5 & & & & \\
M_2 & j_3 & & & & & & \\
M_1 & j_4 & j_6 & & & & & \\
\end{array}
\]
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example. $\ell = 6$

sorted jobs
For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.

Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.

Use LISTSCHEDULING for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.

Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.

Use $\text{LISTSCHEDULING}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs

Polynomial time for constant $\ell$:

$O(n \log n)$  
$O(m^\ell)$  
$O(n \log m)$
Multiprocessor Scheduling – PTAS

For a constant $\ell$ ($1 \leq \ell \leq n$) define the algorithm $A_\ell$ as follows.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.

Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.

Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Polynomial time for constant $\ell$:
$O(m^\ell + n \log n)$
Multiprocessor Scheduling – PTAS

For a constant \( \ell \) \((1 \leq \ell \leq n)\) define the algorithm \( A_\ell \) as follows.

\[
A_\ell(J_1, \ldots, J_n, m)
\]

- Sort jobs in descending order of runtime.
- Schedule the \( \ell \) longest jobs \( J_1, \ldots, J_\ell \) optimally.
- Use \textsc{ListScheduling} for the remaining jobs \( J_{\ell+1}, \ldots, J_n \).

\[
\mathcal{O}(n \log n)
\]
\[
\mathcal{O}(m^\ell)
\]
\[
\mathcal{O}(n \log m)
\]

\[\Box\]

\begin{itemize}
  \item Polynomial time for constant \( \ell \):
    \[\mathcal{O}(m^\ell + n \log n)\]
\end{itemize}

\[\Box\]

\textbf{Theorem 8.}

For constant \( 1 \leq \ell \leq n \), the algorithm \( A_\ell \)
is a \( 1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor} \)-approximation algorithm.

\[\Box\]

For \( \epsilon > 0 \), choose \( \ell \) such that \( A_\epsilon = A_\ell(\epsilon) \)
is a \((1 + \epsilon)\)-approximation algorithm.

\[\Box\]

\textbf{Corollary 9.}

For a constant number of machines, \( \{A_\epsilon \mid \epsilon > 0\} \) is a PTAS.
Multiprocessor Scheduling – PTAS

For a constant \( \ell \) (\( 1 \leq \ell \leq n \)) define the algorithm \( A_\ell \) as follows.

\[
A_\ell(J_1, \ldots, J_n, m) =
\begin{align*}
&\text{Sort jobs in descending order of runtime.} \\
&\text{Schedule the } \ell \text{ longest jobs } J_1, \ldots, J_\ell \text{ optimally.} \\
&\text{Use ListScheduling for the remaining jobs } J_{\ell+1}, \ldots, J_n.
\end{align*}
\]

\( \mathcal{O}(m^{\ell}) \)

\( \mathcal{O}(n \log m) \)

\( \mathcal{O}(n \log n) \)

\( \mathcal{O}(m^{\ell} + n \log n) \)

For \( \varepsilon > 0 \), choose \( \ell \) such that \( A_{\varepsilon} = A_\ell(\varepsilon) \) is a \( (1 + \varepsilon) \)-approximation algorithm.

\( \{A_{\varepsilon} \mid \varepsilon > 0\} \) is not an FPTAS since the running time is not polynomial in \( \frac{1}{\varepsilon} \).
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \frac{\ell}{m}}$-approximation algorithm.

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.
Theorem 8. For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1}{1+\left\lfloor \frac{n}{m} \right\rfloor}$-approximation algorithm.

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

Case 1. $J_k$ is one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use LISTSCHEDULING for the remaining jobs $J_{\ell+1}, \ldots, J_n$. 

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.
Multiprocessor Scheduling – PTAS (proof)

**Theorem 8.**
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

**Proof.** Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

**Case 1.** $J_k$ is one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.

- Solution is optimal for $J_1, \ldots, J_k$
- Hence, solution is optimal for $J_1, \ldots, J_n$
Theorem 8. For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a
$1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

Case 1. $J_k$ is one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.
- Solution is optimal for $J_1, \ldots, J_k$
- Hence, solution is optimal for $J_1, \ldots, J_n$

Case 2. $J_k$ is not one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use LISTSCHEDULING for the remaining jobs $J_{\ell+1}, \ldots, J_n$. 

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof.

Case 1. $J_k$ is one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.
- Solution is optimal for $J_1, \ldots, J_k$
- Hence, solution is optimal for $J_1, \ldots, J_n$

Case 2. $J_k$ is not one of the longest $\ell$ jobs $J_1, \ldots, J_\ell$.
- Similar analysis to ListScheduling
- Use that there are $\ell + 1$ jobs that are at least as long as $J_k$ (including $J_k$).

Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, $T_k$ determines the makespan.

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$. 

\[ M_1 \]
\[ \begin{array}{c}
M_2 \\
M_3 \\
M_4
\end{array}
\]

\[ S_k \]
\[ T_k = \text{Makespan } A_\ell \]
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_{\ell}$ is a
$1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- $T_{OPT} \geq p_k$

$A_{\ell}(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a
$1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- $T_{OPT} \geq p_k$

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_{\ell}$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$
$\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k$
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a
$1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{\text{OPT}} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- $T_{\text{OPT}} \geq p_k$

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$

\[
\begin{align*}
T_k &\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
&= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k
\end{align*}
\]
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lceil \frac{\ell}{m} \right\rceil}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- $T_{OPT} \geq p_k$

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

\[
T_k = S_k + p_k \\
\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
\leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}
\]
Theorem 8. For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a
$1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{\text{OPT}} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- $T_{\text{OPT}} \geq p_k$

$A_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$
\[
\leq \frac{1}{m} \sum_{i \neq k} p_i + p_k
\]
\[
= \frac{1}{m} \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k
\]
\[
\leq T_{\text{OPT}} + \left(1 - \frac{1}{m}\right) \cdot T_{\text{OPT}}
\]

can we do better?
Multiprocessor Scheduling – PTAS (proof)

**Theorem 8.**
For constant \(1 \leq \ell \leq n\), the algorithm \(A_\ell\) is a \(1 + \frac{1 - \frac{1}{m}}{1 + \frac{\ell}{m}}\)-approximation algorithm.

**Proof of Case 2.**

- \(S_k \leq \frac{1}{m} \sum_{i \neq k} p_i\)
- \(T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i\)
- Consider only \(J_1, \ldots, J_\ell, J_k\):

\[
T_{OPT} \geq p_k .
\]

\(A_\ell(J_1, \ldots, J_n, m)\)

Sort jobs in descending order of runtime.
Schedule the \(\ell\) longest jobs \(J_1, \ldots, J_\ell\) optimally.
Use ListScheduling for the remaining jobs \(J_{\ell+1}, \ldots, J_n\).

\[
T_k = S_k + p_k \leq \frac{1}{m} \sum_{i \neq k} p_i + p_k = \frac{1}{m} \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}
\]

**can we do better?**
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_{\ell}$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

Consider only $J_1, \ldots, J_{\ell}, J_k$:

$$T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)$$

$A_{\ell}(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_{\ell}$ optimally.
Use $\text{ListScheduling}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$$T_k = S_k + p_k \leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k$$
$$= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k$$
$$\leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}$$

Can we do better?
Multiprocessor Scheduling – PTAS (proof)

**Theorem 8.**
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

**Proof of Case 2.**

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- Consider only $J_1, \ldots, J_\ell, J_k$:

  $T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)$ one machine has this many jobs*

where $T_k = S_k + p_k$

\[
T_k = \frac{1}{m} \cdot \left(1 - \frac{1}{m}\right) \cdot p_k 
\leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}
\]

- $A_\ell(J_1, \ldots, J_n, m)$
  - Sort jobs in descending order of runtime.
  - Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
  - Use $\text{ListScheduling}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

- Consider only $J_1, \ldots, J_\ell, J_k$:

  one machine has
  this many jobs

- Can we do better?
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.
- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

Consider only $J_1, \ldots, J_\ell, J_k$:

$T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor \right)$ one machine has this many jobs*

* on average, each machine has more than $\frac{\ell}{m}$ of the $\ell + 1$ jobs
at least one machine achieves the average

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$
\[
\leq \frac{1}{m} \sum_{i \neq k} p_i + p_k
\]
\[
= \frac{1}{m} \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) p_k
\]
\[
\leq T_{OPT} + \left(1 - \frac{1}{m}\right) T_{OPT}
\]
can we do better?
Theorem 8. For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

Consider only $J_1, \ldots, J_\ell, J_k$:

- $T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)$ one machine has this many jobs\(^*\)
  each has length $\geq p_k$

- \(^*\) on average, each machine has more than $\frac{\ell}{m}$ of the $\ell + 1$ jobs
- at least one machine achieves the average

$T_k = S_k + p_k$

\[ \leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \]

\[ = \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \]

\[ \leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT} \]

Can we do better?

$A_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use $\text{ListScheduling}$ for the remaining jobs $J_{\ell+1}, \ldots, J_n$. 

$M_4$

$M_3$

$M_2$

$M_1$

\[ S_k \quad T_k = \text{Makespan}_{A_\ell} \]
Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $A_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$-approximation algorithm.

Proof of Case 2.

- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- Consider only $J_1, \ldots, J_\ell, J_k$:
  - $T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)$ one machine has this many jobs* each has length $\geq p_k$
- * on average, each machine has more than $\frac{\ell}{m}$ of the $\ell + 1$ jobs
- at least one machine achieves the average

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the $\ell$ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

$T_k = S_k + p_k$
$\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k$
$= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k$
$\leq T_{OPT} + \left(1 - \frac{1}{m}\right) \cdot T_{OPT}$

\[ Tk = \text{Makespan} \mathcal{A}_\ell \]
Theorem 8.
For constant \(1 \leq \ell \leq n\), the algorithm \(A_\ell\) is a \(1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}\)-approximation algorithm.

Proof of Case 2.

- \(S_k \leq \frac{1}{m} \sum_{i \neq k} p_i\)
- \(T_{OPT} \geq \frac{1}{m} \sum_{i=1}^{n} p_i\)
- Consider only \(J_1, \ldots, J_\ell, J_k\):
  \(T_{OPT} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)\) one machine has this many jobs*: each has length \(\geq p_k\)
- * on average, each machine has more than \(\frac{\ell}{m}\) of the \(\ell + 1\) jobs
- at least one machine achieves the average

\[
T_k = S_k + p_k \\
\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
\leq T_{OPT} + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor} \cdot T_{OPT}
\]
Discussion

- Only “easy” NP-hard problems admit FPTAS (PTAS).
- Some problems cannot be approximated very well (e.g., Maximum Clique).
- Study of approximability of NP-hard problems yields a more fine-grained classification of the difficulty.
Discussion

- Only “easy” NP-hard problems admit FPTAS (PTAS).
- Some problems cannot be approximated very well (e.g., Maximum Clique).
- Study of approximability of NP-hard problems yields a more fine-grained classification of the difficulty.
- Approximation algorithms exist also for non-NP-hard problems.
- Approximation algorithms can be of various types: greedy, local search, geometric, DP, . . .
- One important technique is LP-relaxation (next lecture).
Discussion

- Only “easy” NP-hard problems admit FPTAS (PTAS).
- Some problems cannot be approximated very well (e.g., Maximum Clique).
- Study of approximability of NP-hard problems yields a more fine-grained classification of the difficulty.

- Approximation algorithms exist also for non-NP-hard problems
- Approximation algorithms can be of various types: greedy, local search, geometric, DP, ...

- One important technique is LP-relaxation (next lecture).

- Minimum Vertex Coloring on planar graphs can be approximated with an additive approximation guarantee of 2.

- Christofides’ approximation algorithm for Metric TSP has approximation factor 1.5.
Literature

Main references


Another book recommendation:

- [Vazirani, 2013] “Approximation Algorithms”