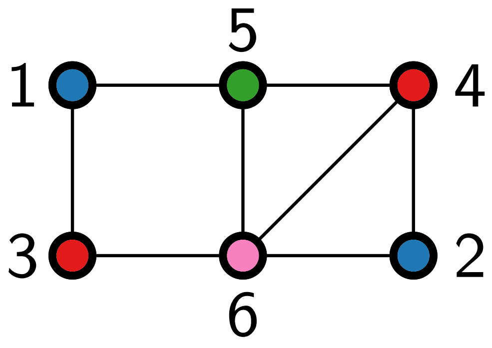
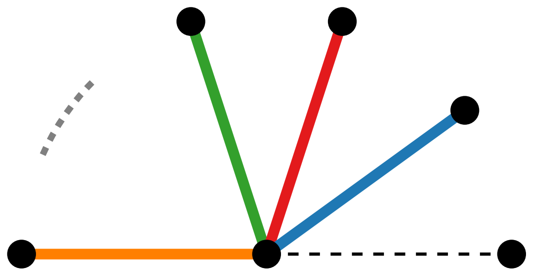
Advanced Algorithms

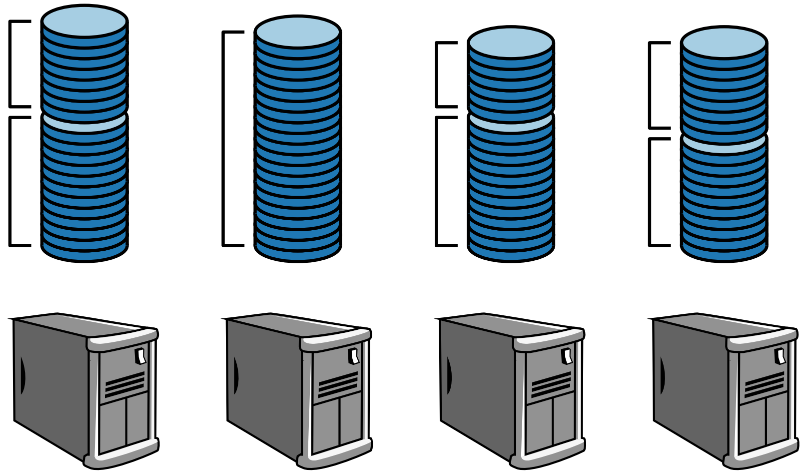
Approximation Algorithms

Coloring and Scheduling Problems

Johannes Zink · WS23/24



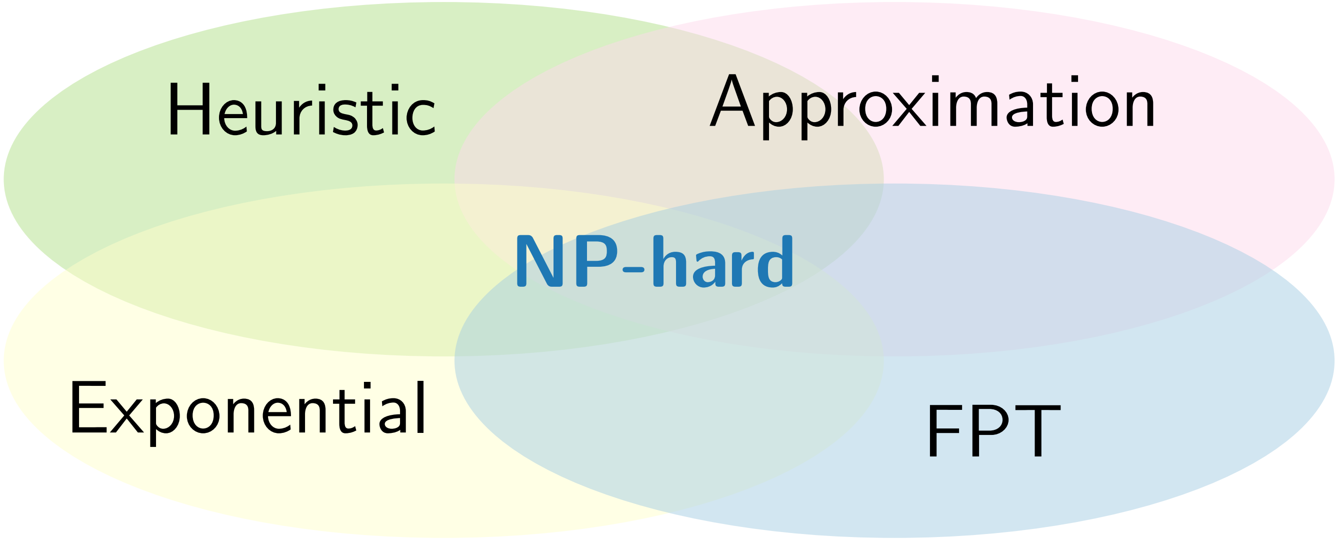




Dealing with NP-Hard Optimization Problems

What should we do?

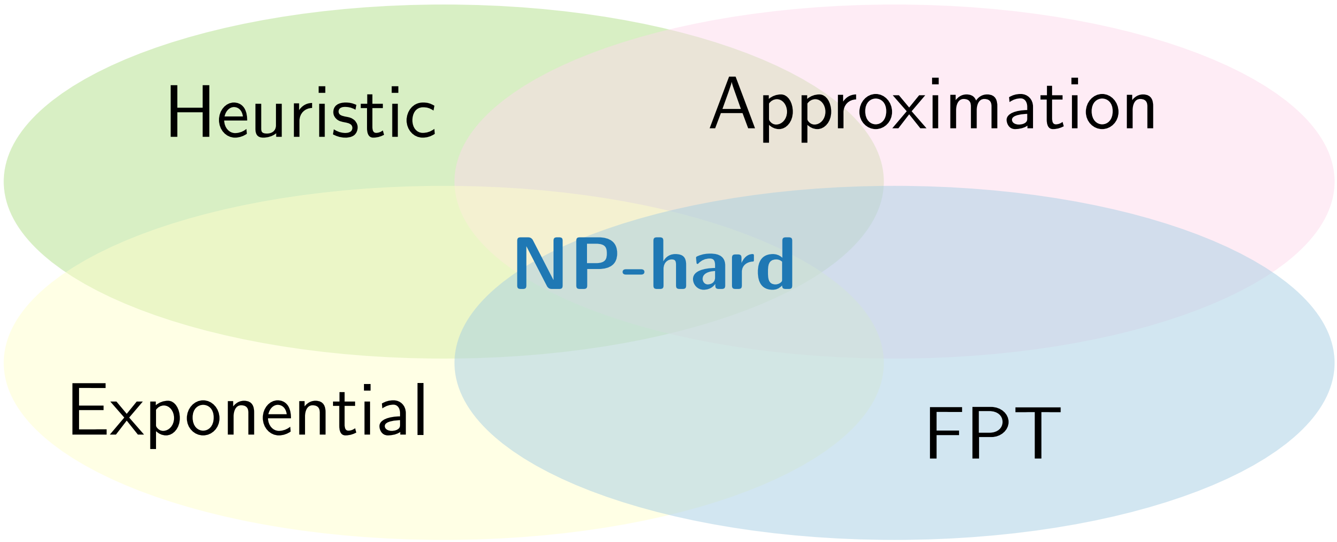
- Sacrifice optimality for speed
  - Heuristics
  - Approximation algorithms
- Optimal solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms



Dealing with NP-Hard Optimization Problems

What should we do?

- Sacrifice optimality for speed
  - Heuristics
  - Approximation algorithms ← this lecture
- Optimal solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized 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.

Approximation Algorithms

Problem.

- For NP-hard optimization problems, we cannot compute the optimal solution of every instance efficiently (unless $\mathrm{P} = \mathrm{NP}$).
- 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 additive guarantee,
  - with relative guarantee,
  - that are "arbitrarily good".

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 additive guarantee,
  - with relative guarantee,
  - that are "arbitrarily good".

PTAS (*polynomial-time approximation scheme*)

Approximation with Additive Guarantee

Definition.
Let Π be an optimization problem,
let $\mathcal{A}$ be a polynomial-time algorithm for Π,
let I be an instance of Π, and
let 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** δ (which can depend on I) if

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

for every instance I of Π.

Approximation with Additive Guarantee

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

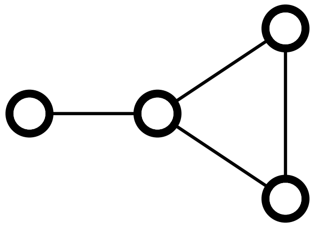
Then $\mathcal{A}$ is called an **approximation algorithm with additive guarantee** δ (which can depend on I) if

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

for every instance I of Π.

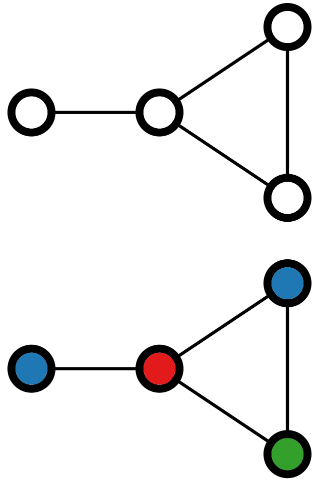
- Most problems that we know do not admit an approximation algorithm with additive guarantee.

Minimum Vertex Coloring



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

Minimum Vertex Coloring



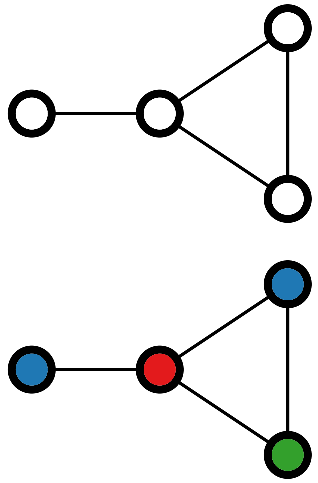
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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 Δ 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.

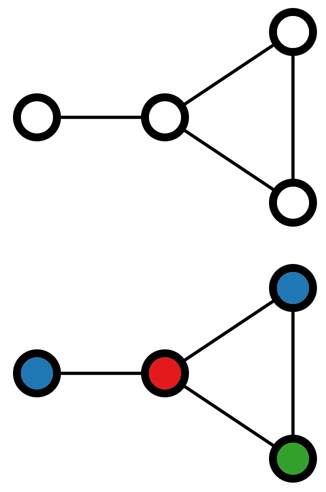


- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

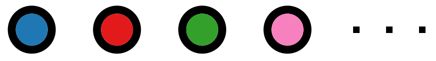
Minimum Vertex Coloring

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

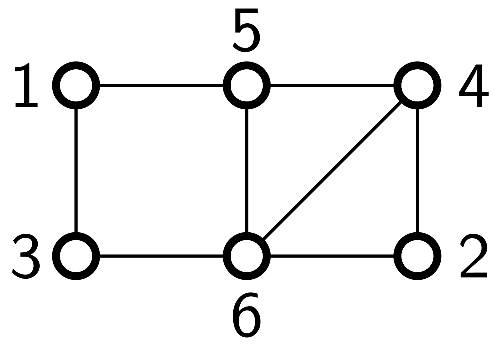
Output. A **minimum vertex coloring**, that is, an assignment of the vertices of G to colors such that no two adjacent vertices get the same color and the number of colors is minimum.



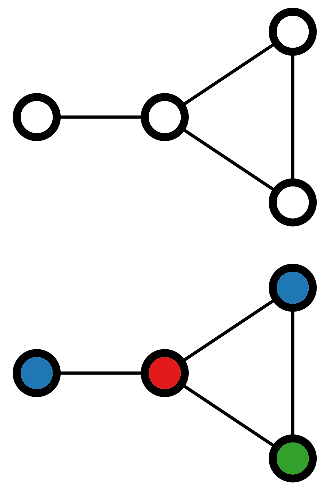
- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.



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



Minimum Vertex Coloring

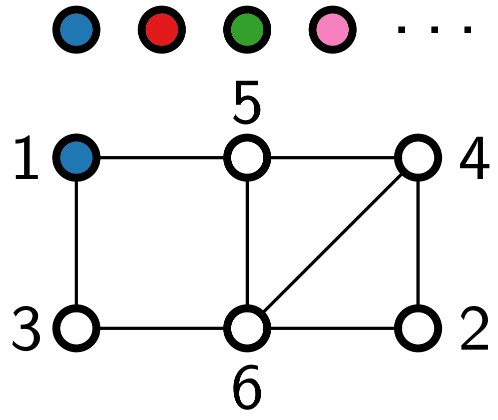


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

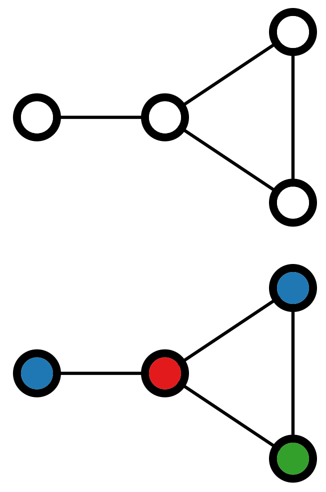
`GreedyVertexColoring`(connected graph G)
Color vertices in some order with the lowest feasible color.



Minimum Vertex Coloring

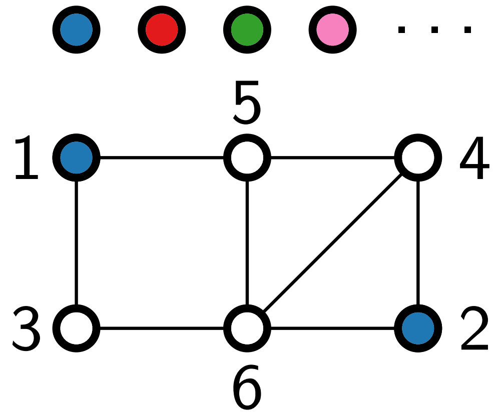
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

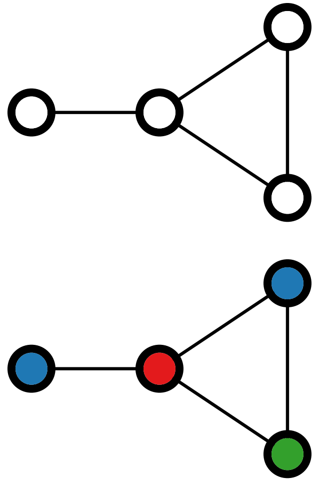
`GreedyVertexColoring`(connected graph G)
  Color vertices in some order with the lowest feasible color.



Minimum Vertex Coloring

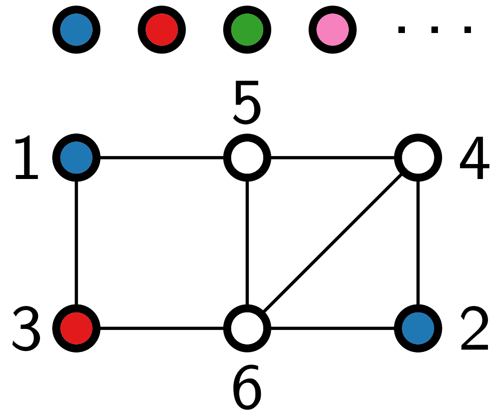
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

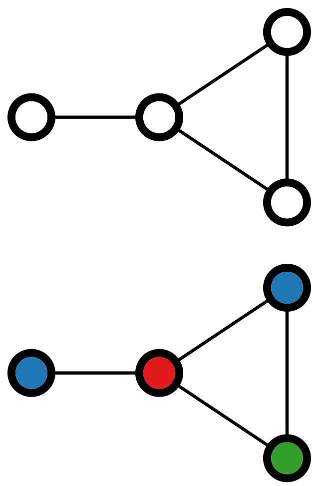
`GreedyVertexColoring`(connected graph G)
  Color vertices in some order with the lowest feasible color.



Minimum Vertex Coloring

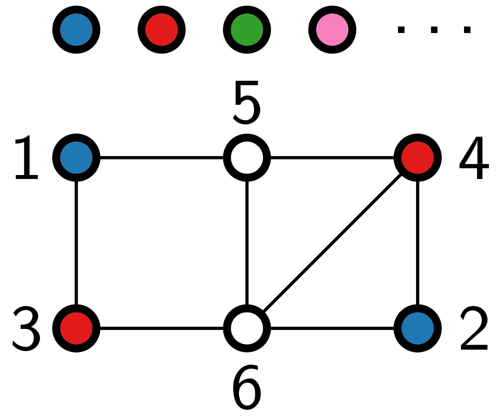
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

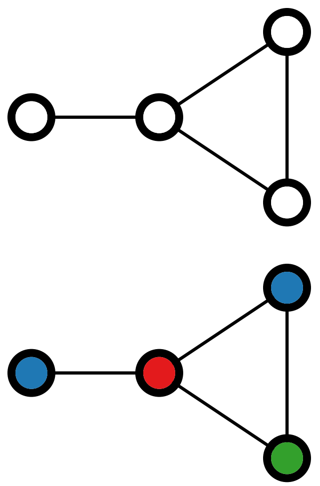
`GreedyVertexColoring`(connected graph G)
  Color vertices in some order with the lowest feasible color.



Minimum Vertex Coloring

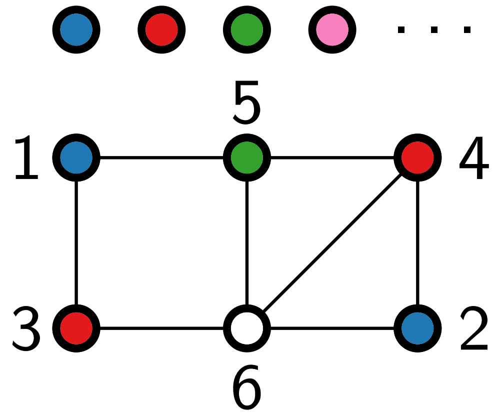
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

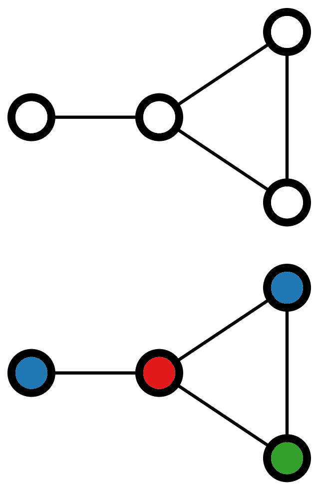
GreedyVertexColoring(connected graph G)
  Color vertices in some order with the lowest feasible color.



Minimum Vertex Coloring

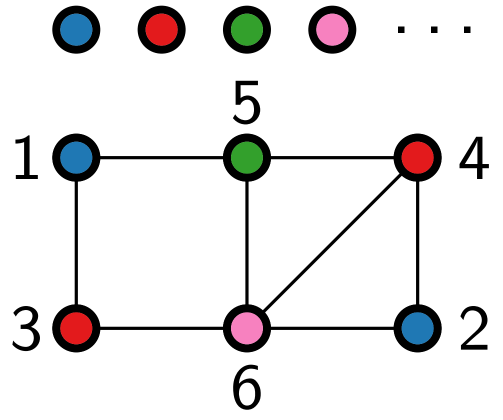
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

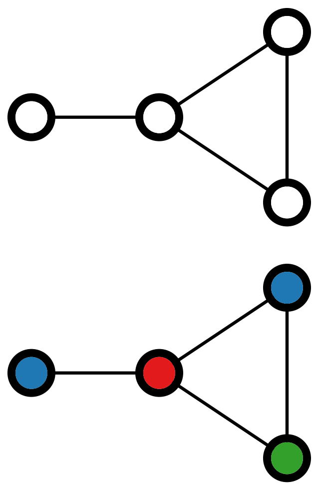
`GreedyVertexColoring`(connected graph G)
  Color vertices in some order with the lowest feasible color.



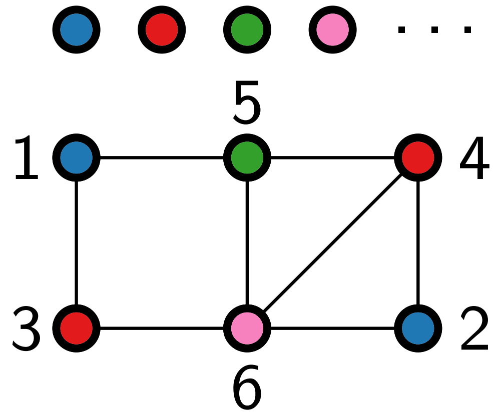
Minimum Vertex Coloring

Input. A graph $G = (V, E)$. Let Δ 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.



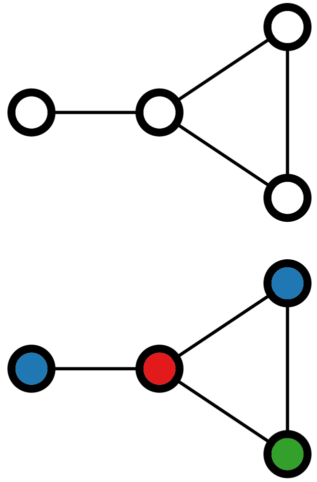
- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.




```
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 colors in $\mathcal{O}(V + E)$ time.
Hence, it has an additive approximation gurantee of .

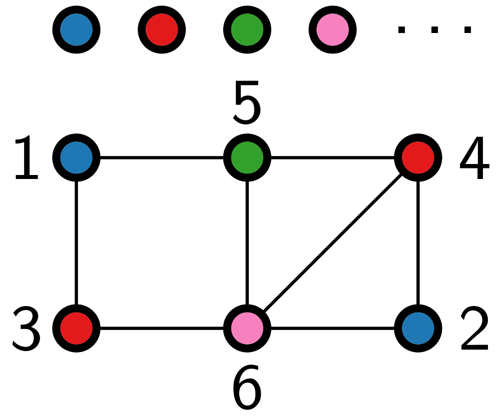
Minimum Vertex Coloring



Input. A graph $G = (V, E)$. Let Δ 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.

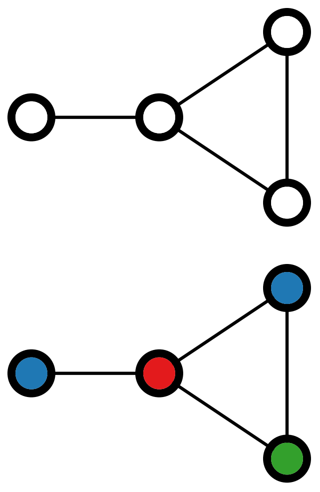
- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.



`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 $\mathcal{O}(V + E)$ time.
Hence, it has an additive approximation gurantee of   .

Minimum Vertex Coloring

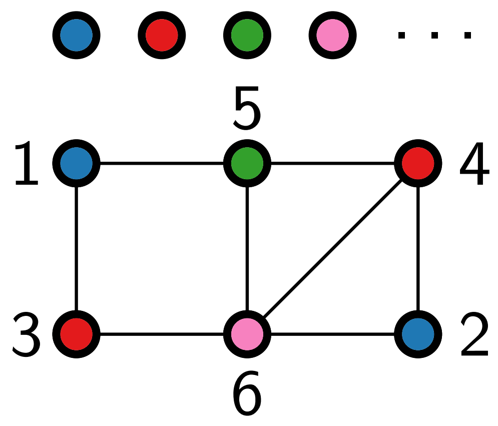


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

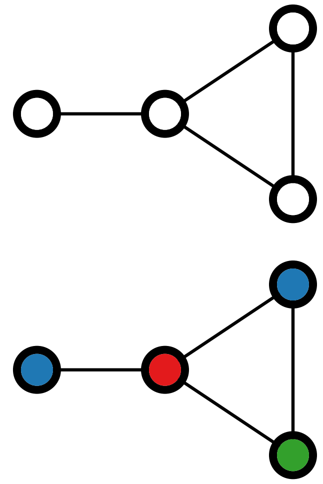
- Minimum Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

`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 $\mathcal{O}(V + E)$ time.
Hence, it has an additive approximation gurantee of $\Delta - 1$.

Minimum Vertex Coloring

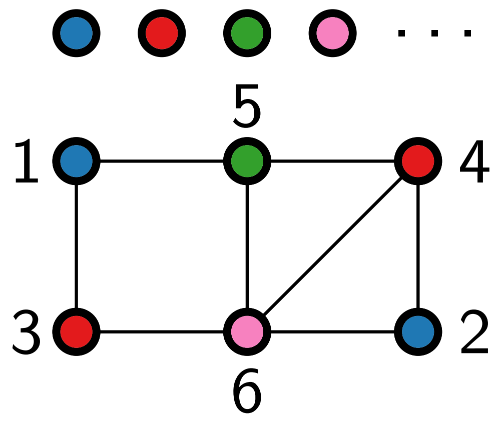


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

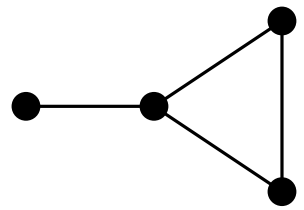
`GreedyVertexColoring`(connected graph G)
  Color vertices in some order with the lowest feasible color.



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

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

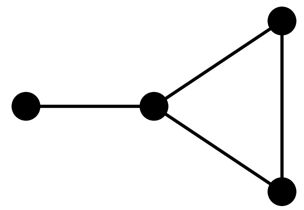
Minimum Edge Coloring



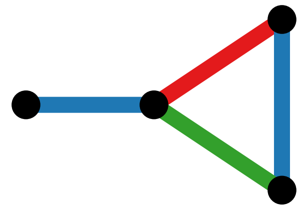
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Minimum Edge Coloring

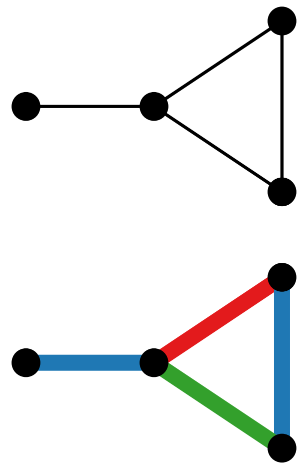
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.



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

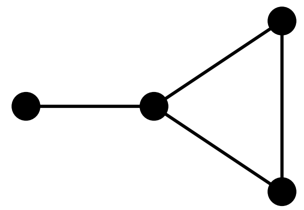


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

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.

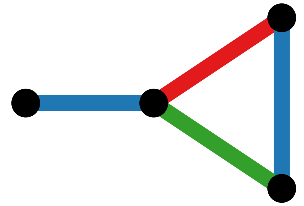
- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.

Minimum Edge Coloring



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

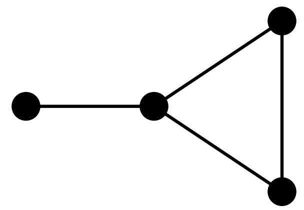
Output. A **minimum edge coloring**, that is, an assignment of colors to the edges of G such that no two adjacent edges get the same color and the number of colors is minimum.



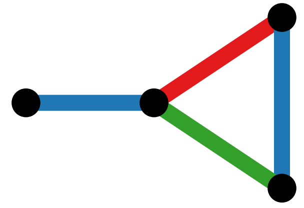
- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.

- 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

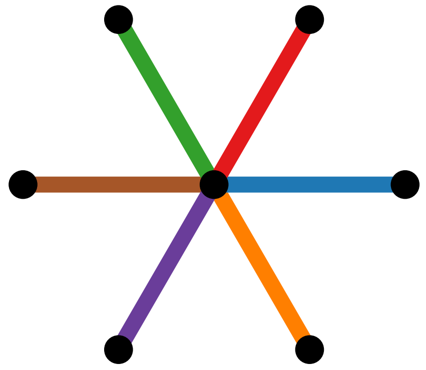


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.



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.

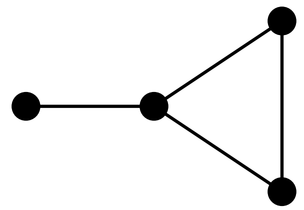
- Minimum Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.



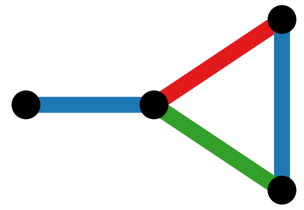
- The minimum number of colors needed for an edge coloring of G is called the **chromatic index** $\chi'(G)$.
- $\chi'(G)$ is lowerbounded by Δ.

Minimum Edge Coloring

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

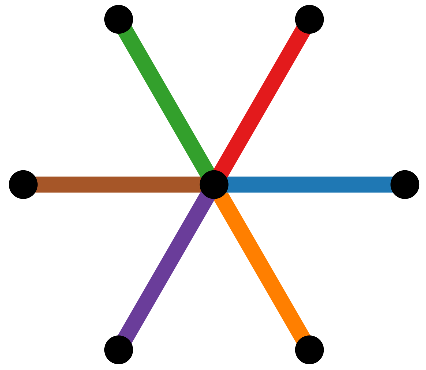


Output. A **minimum edge coloring**, that is, an assignment of colors to the edges of G such that no two adjacent edges get the same color and the number of colors is minimum.



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

- The minimum number of colors needed for an edge coloring of G is called the **chromatic index** $\chi'(G)$.
- $\chi'(G)$ is lowerbounded by Δ.
- 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 Δ, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)

Minimum Edge Coloring – Upper Bound

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

Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)

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 Δ, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

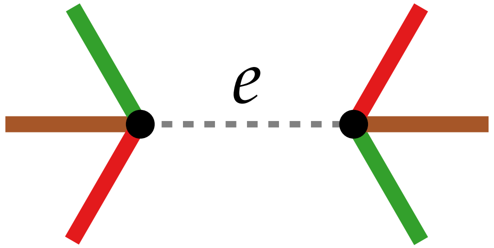
Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)

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.



Minimum Edge Coloring – Upper Bound

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

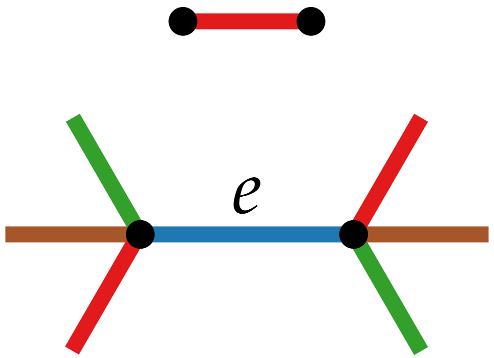
Proof by induction on $m = |E|$.

- 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 Δ, it holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

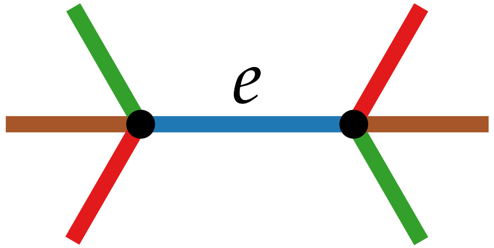
Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)

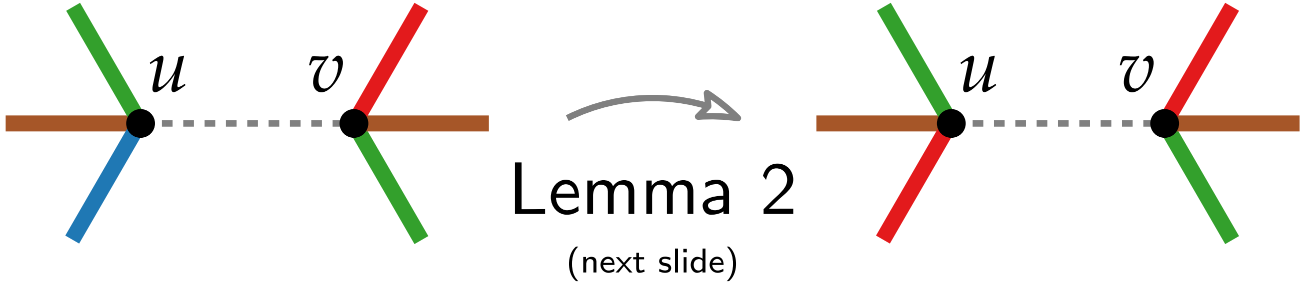
Proof by induction on $m = |E|$.

- 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 α.





Minimum Edge Coloring – Upper Bound

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

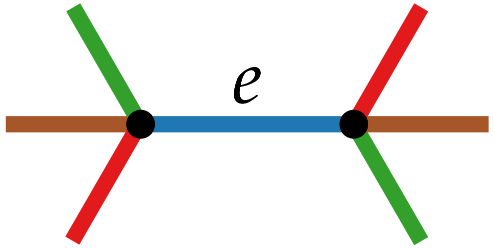
Vadim G. Vizing
(Kiew 1937 – 2017 Odessa)

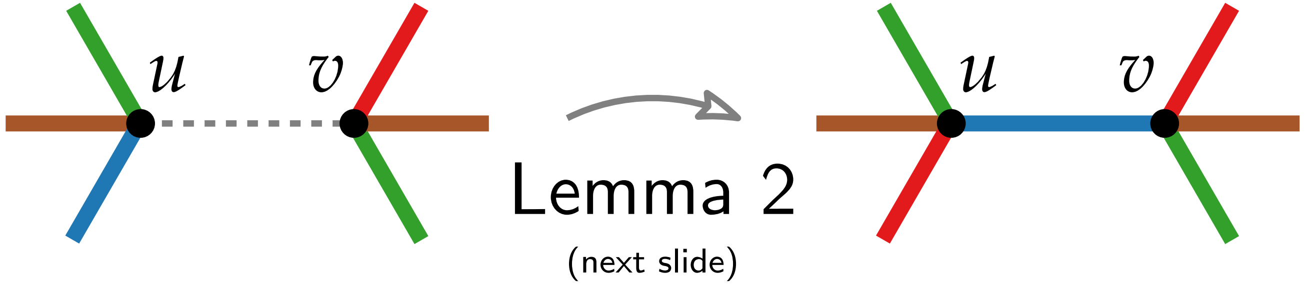
Proof by induction on $m = |E|$.

- 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 α.
- Then color e with α.





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.

Minimum Edge Coloring – Recoloring

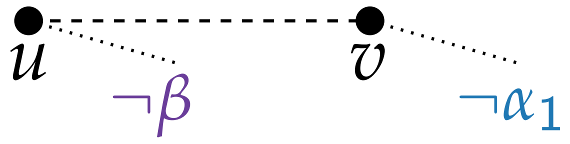
Lemma 2.
Let G be a graph with a $(\Delta + 1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.

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

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.

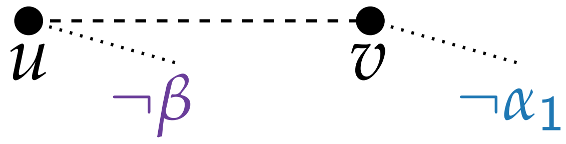
Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:



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 β and v miss α_1; apply the following algorithm:



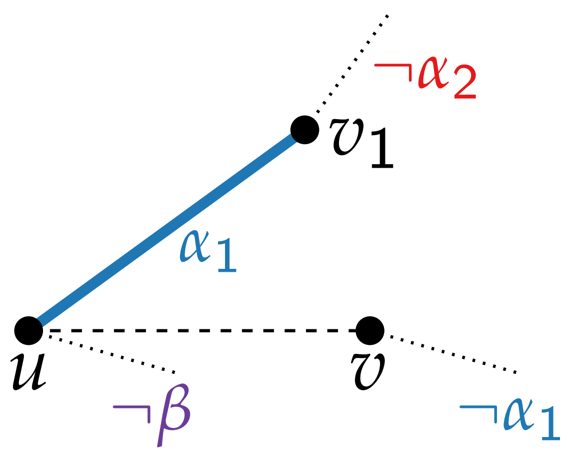
VizingRecoloring(G, c, u, α_1)

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

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.

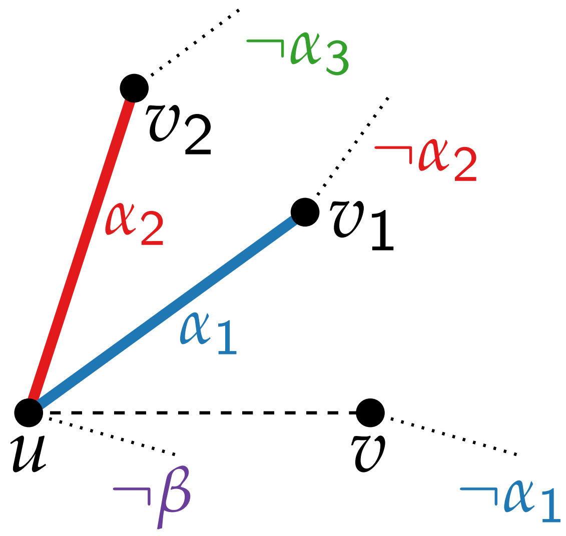
Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:



VizingRecoloring(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i + 1$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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.

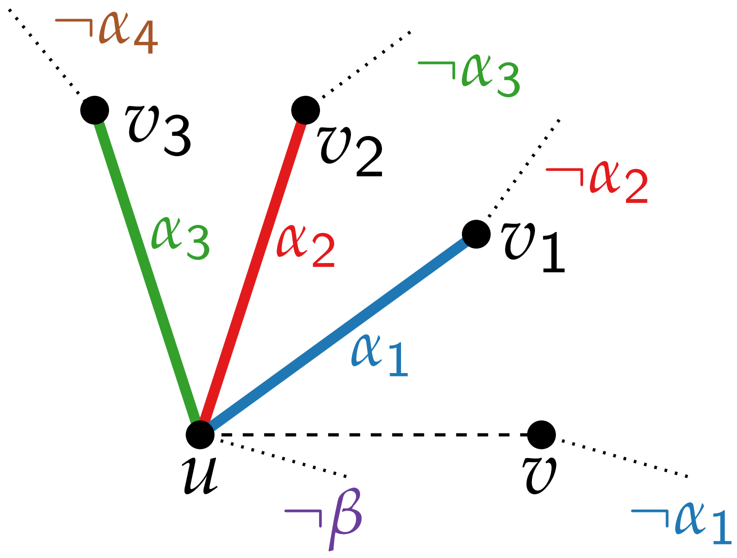


Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i+1$
  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.

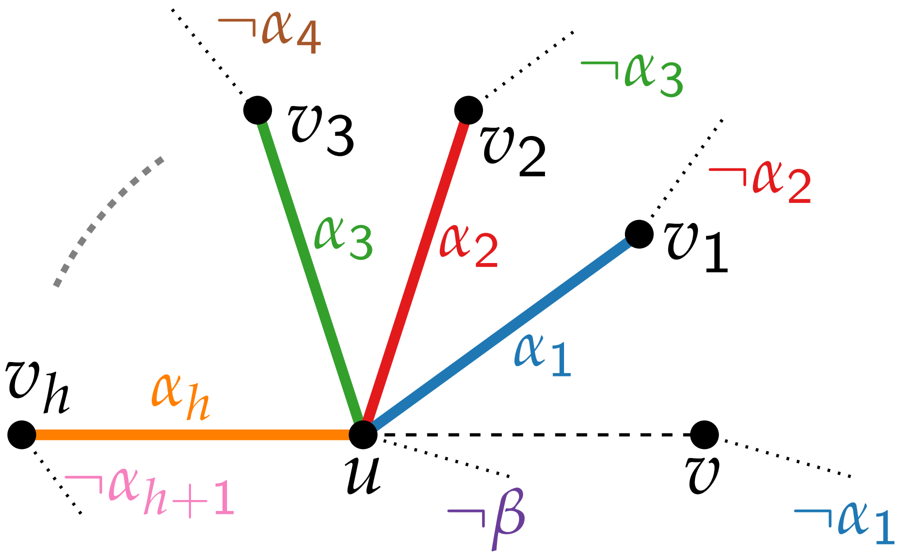


Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i+1$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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.



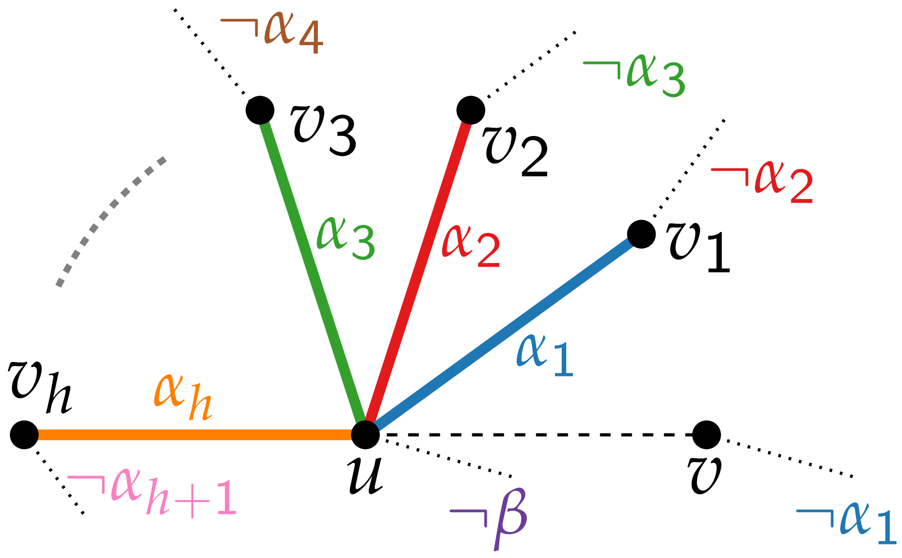
Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)

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

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.



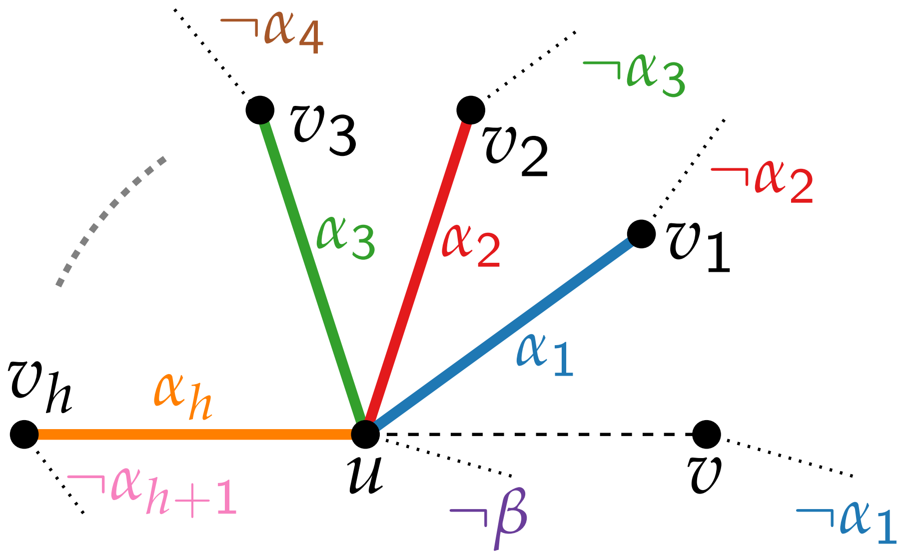
Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u) \colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.

Minimum Edge Coloring – Recoloring

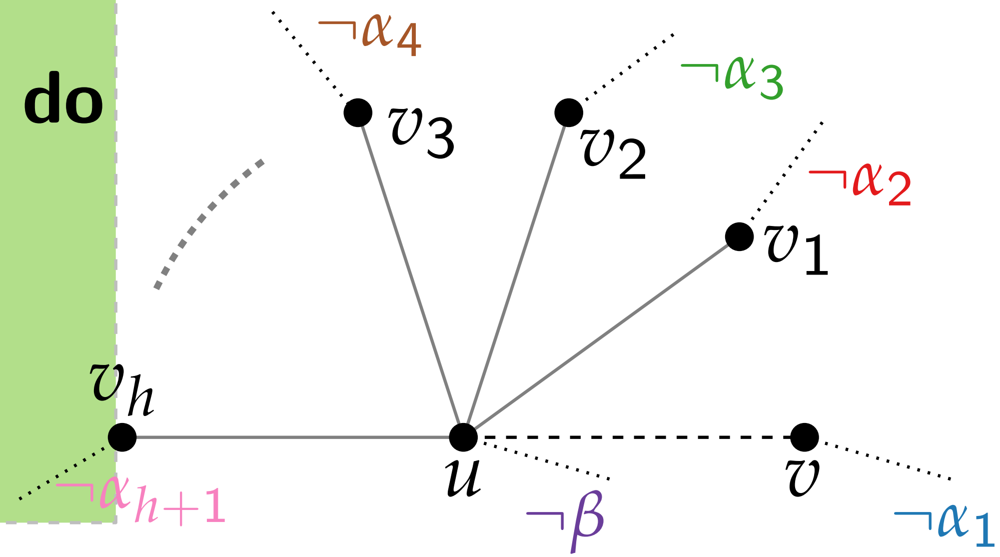
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

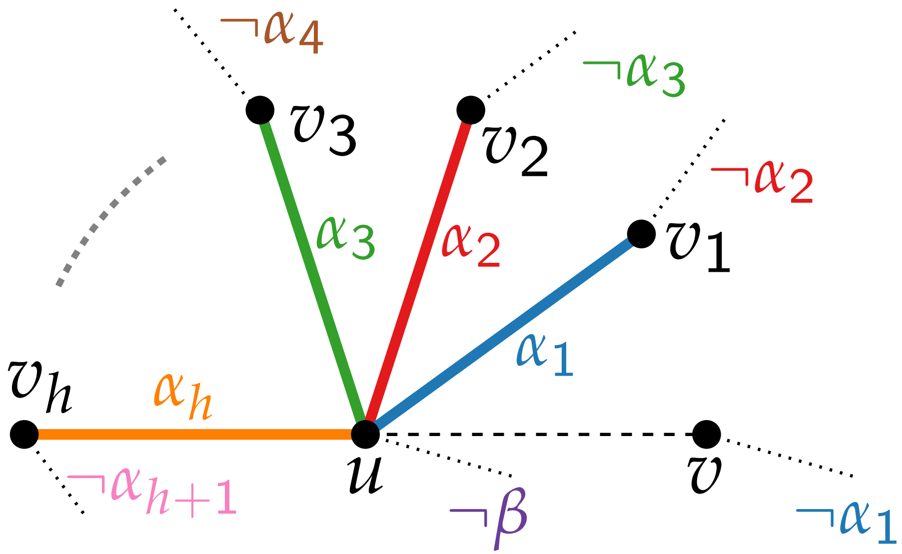
VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u) : c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum Edge Coloring – Recoloring

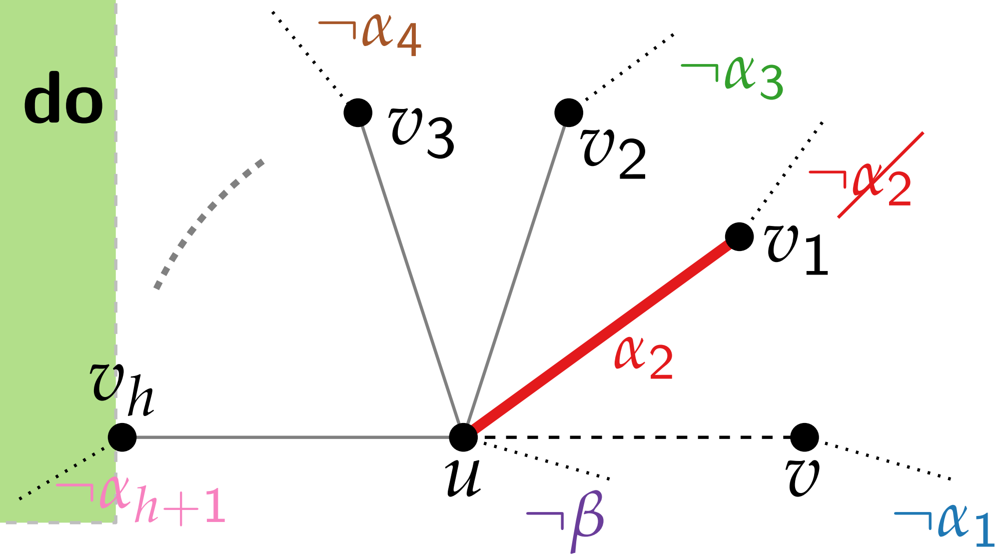
Lemma 2.
Let G be a graph with a $(\Delta + 1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

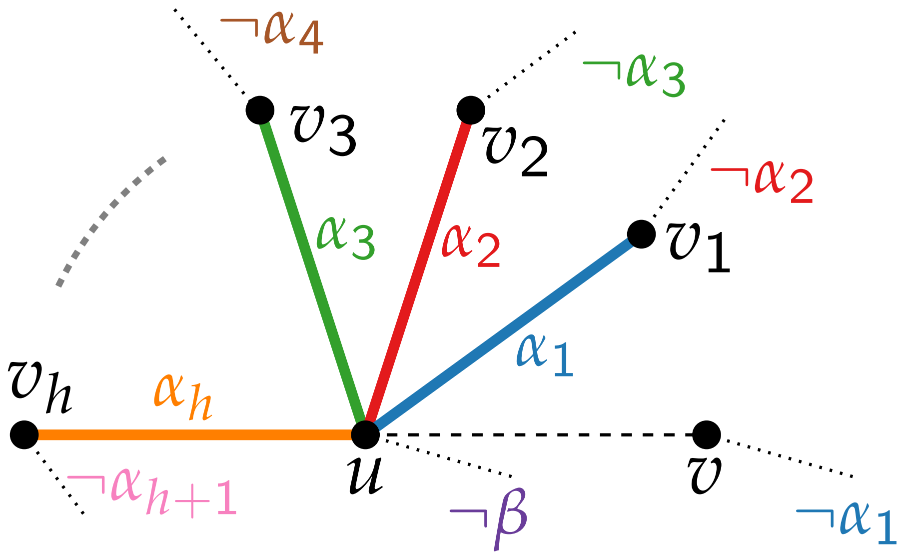
VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u) \colon c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum Edge Coloring – Recoloring

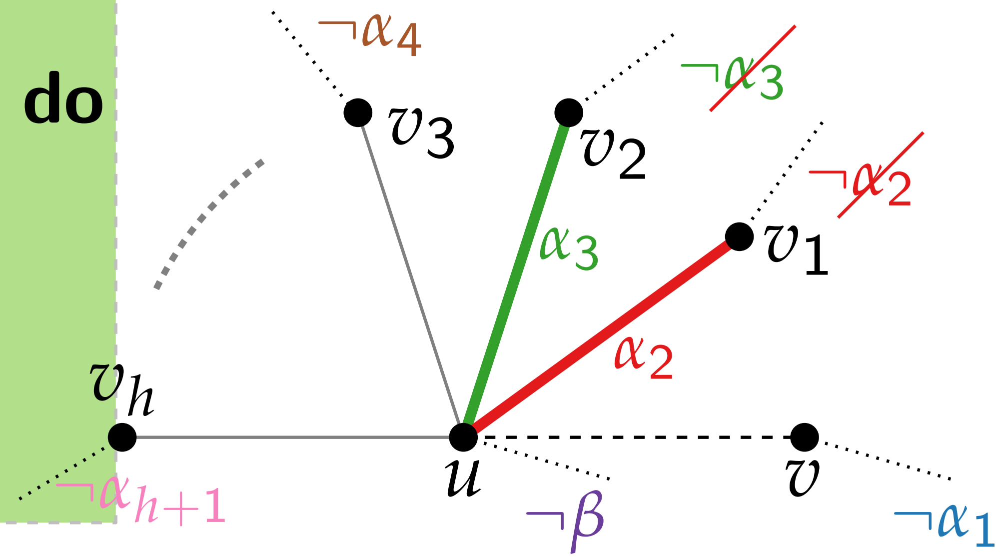
Lemma 2.
Let G be a graph with a $(\Delta + 1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u) : c(uw) = \alpha_i \land w \notin \{v_1, \dots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

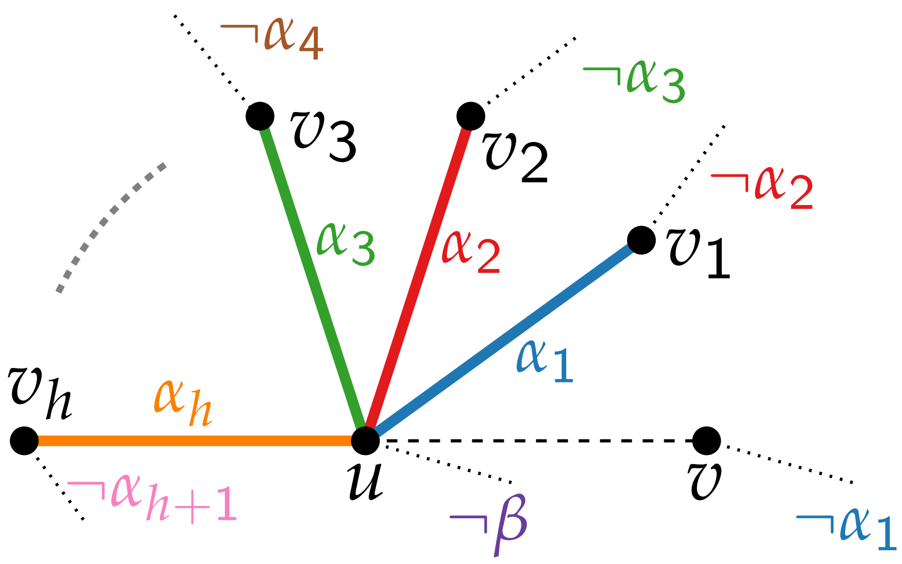
Case 1: u misses α_{h+1}.



Minimum Edge Coloring – Recoloring

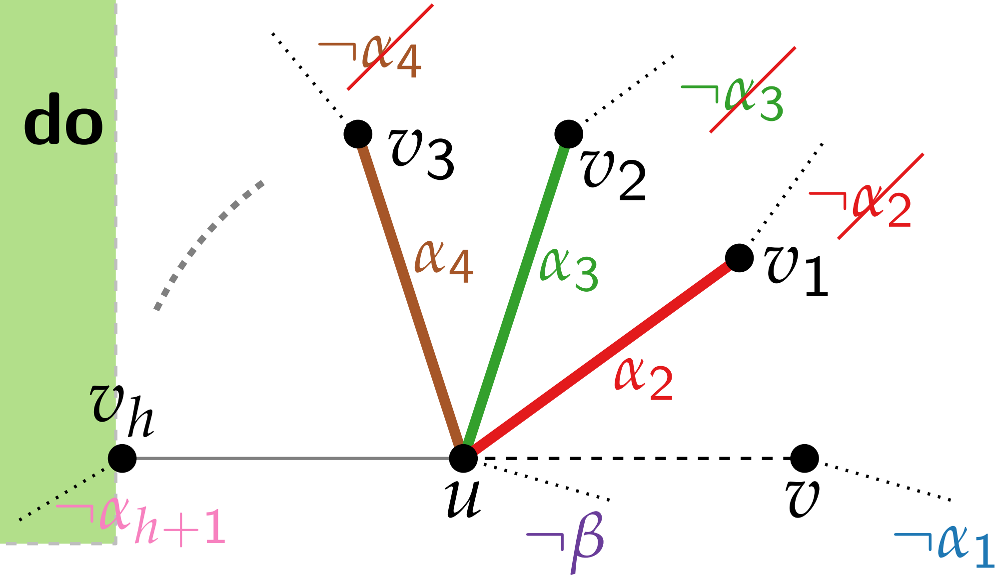
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.

Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:



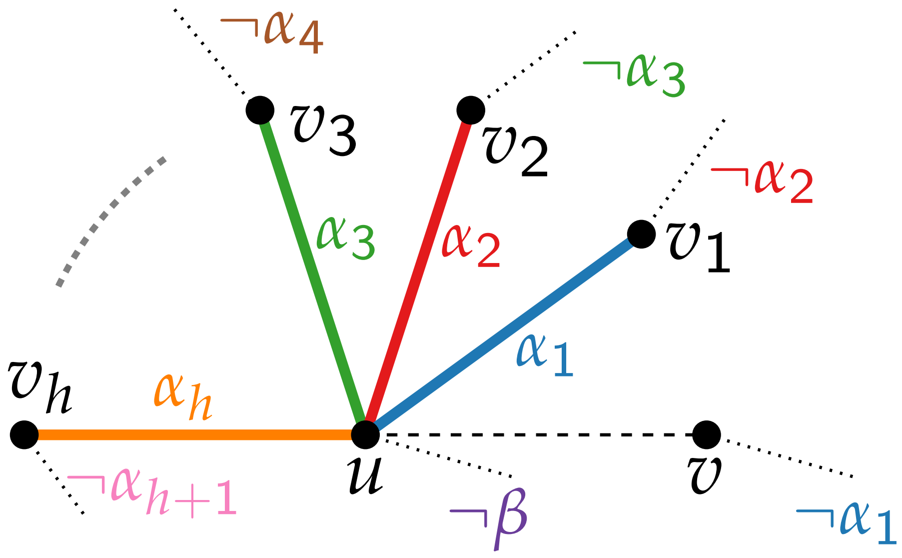

```
VizingRecoloring(G, c, u, α_1)
  i ← 1
  while ∃w ∈ N(u): c(uw) = α_i ∧ w ∉ {v_1, ..., v_{i-1}} do
      v_i ← w
      α_{i+1} ← min color missing at w
      i ← i + 1
  return v_1, ..., v_i; α_1, ..., α_{i+1}
```

Case 1: u misses α_{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.

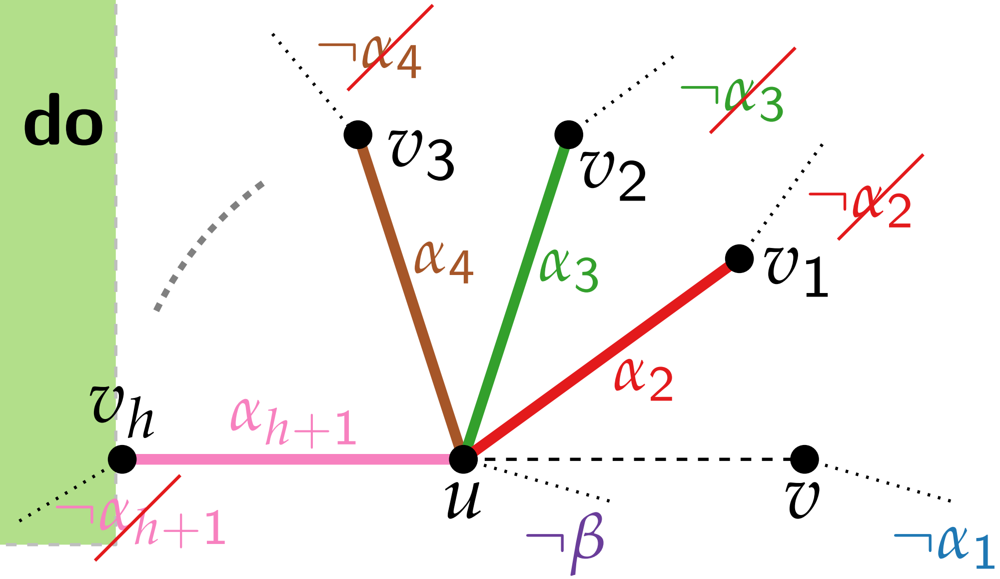


Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)

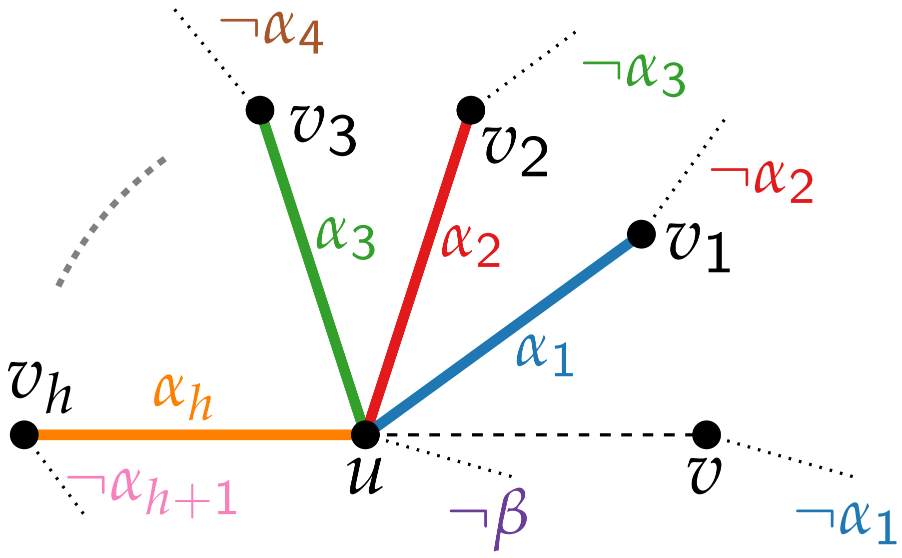
$i \leftarrow 1$
while $\exists w \in N(u) : c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i \leftarrow i + 1$
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum Edge Coloring – Recoloring

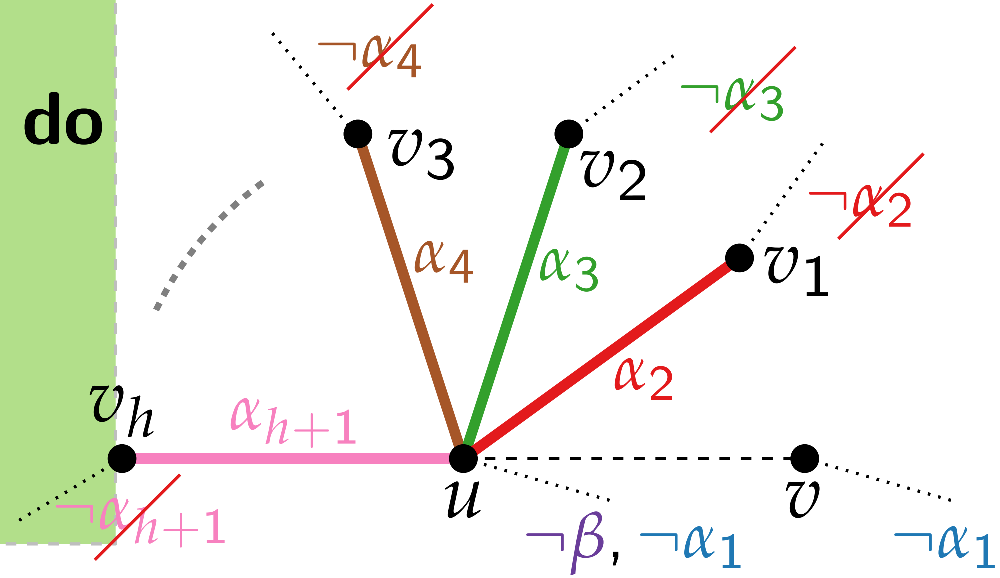
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

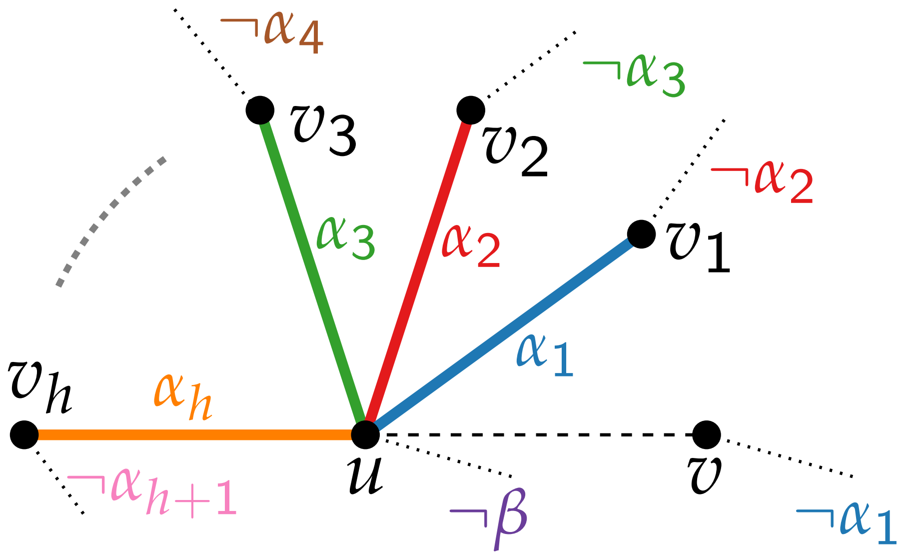
VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u)$: $c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{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.



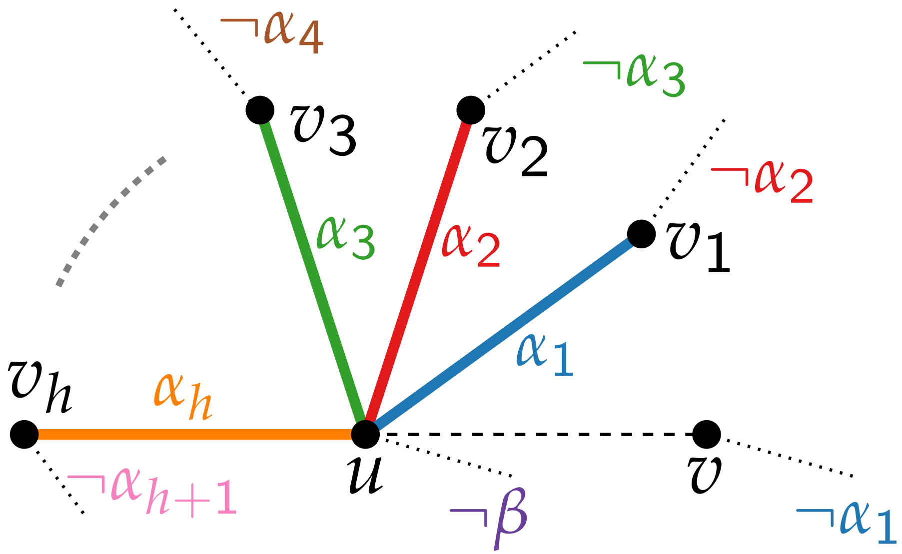
Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

VizingRecoloring(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i + 1$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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

Minimum Edge Coloring – Recoloring

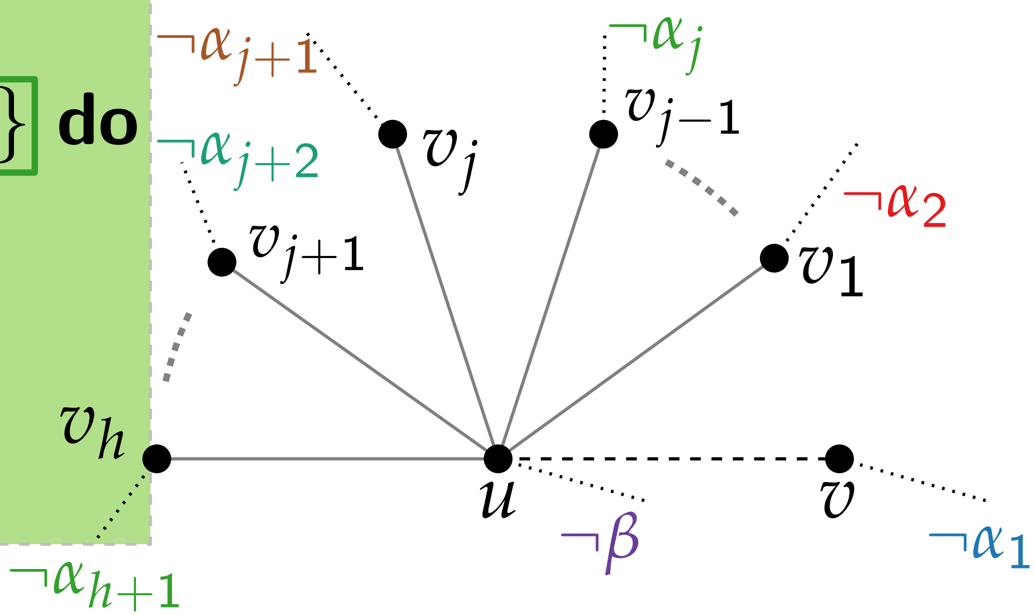
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

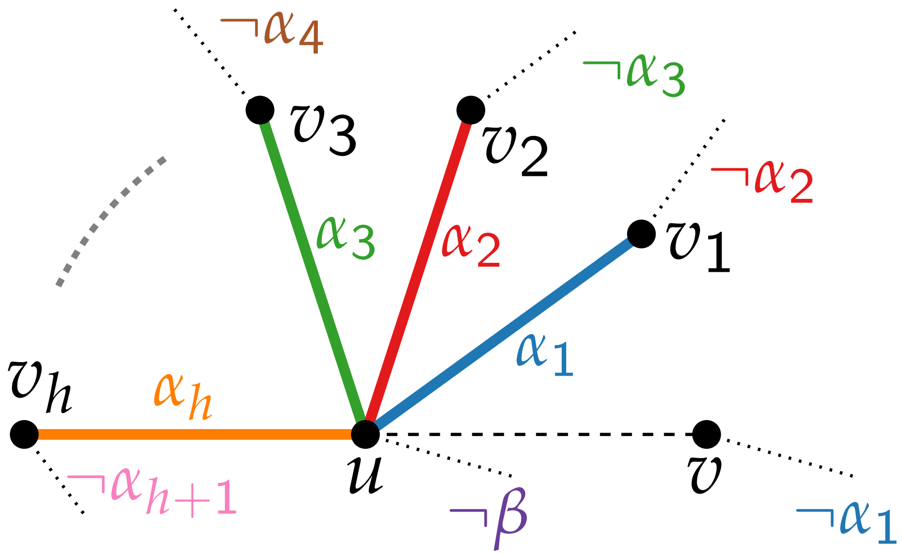
```
VizingRecoloring(G, c, u, α_1)
  i ← 1
  while ∃w ∈ N(u): c(uw) = α_i ∧ w ∉ {v_1, ..., v_{i-1}} do
      v_i ← w
      α_{i+1} ← min color missing at w
      i ← i + 1
  return v_1, ..., v_i; α_1, ..., α_{i+1}
```

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



Minimum Edge Coloring – Recoloring

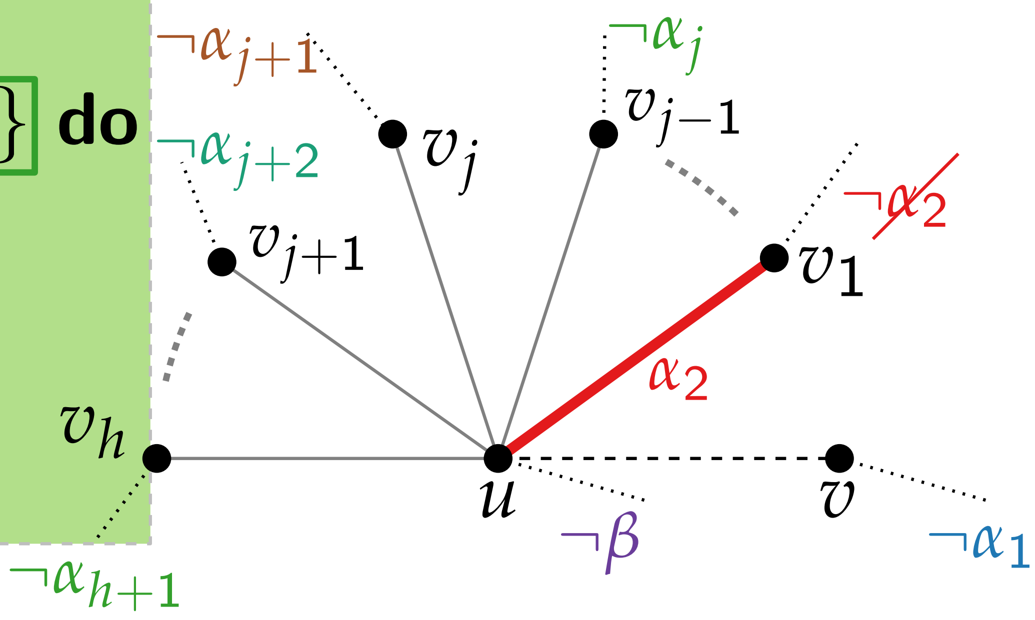
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

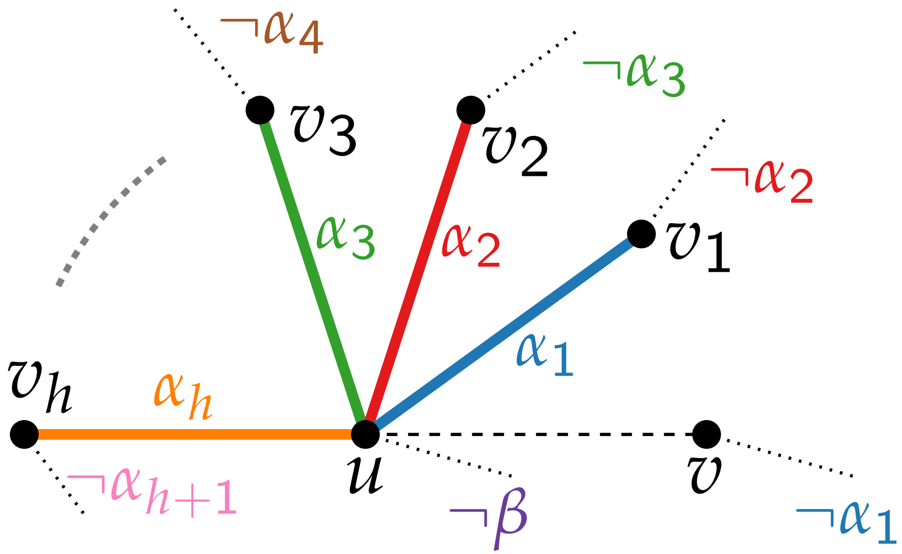
`VizingRecoloring`(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i+1$
- **return** $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

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



Minimum Edge Coloring – Recoloring

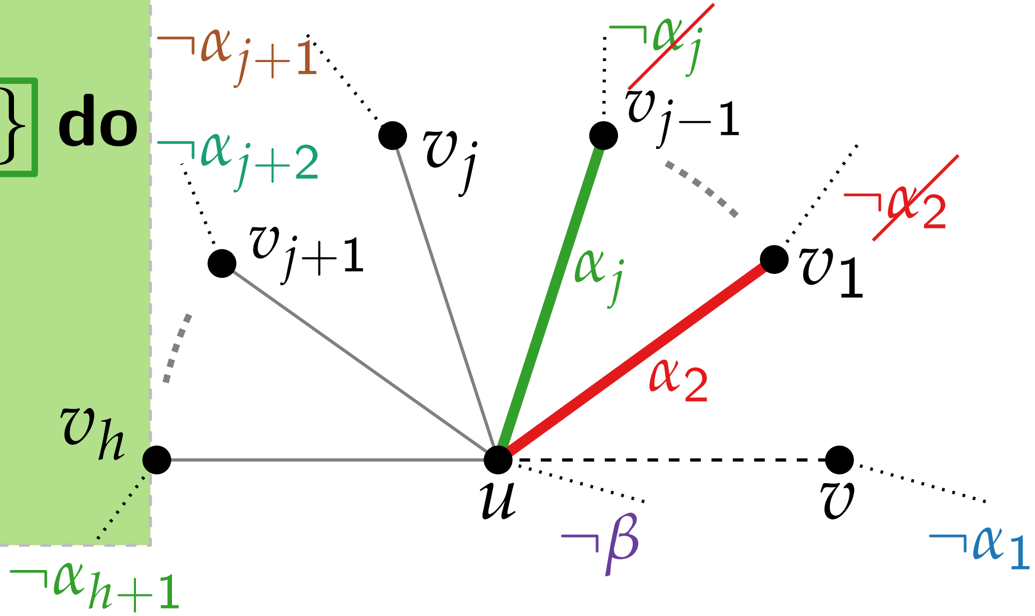
Lemma 2.
Let G be a graph with a $(\Delta + 1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

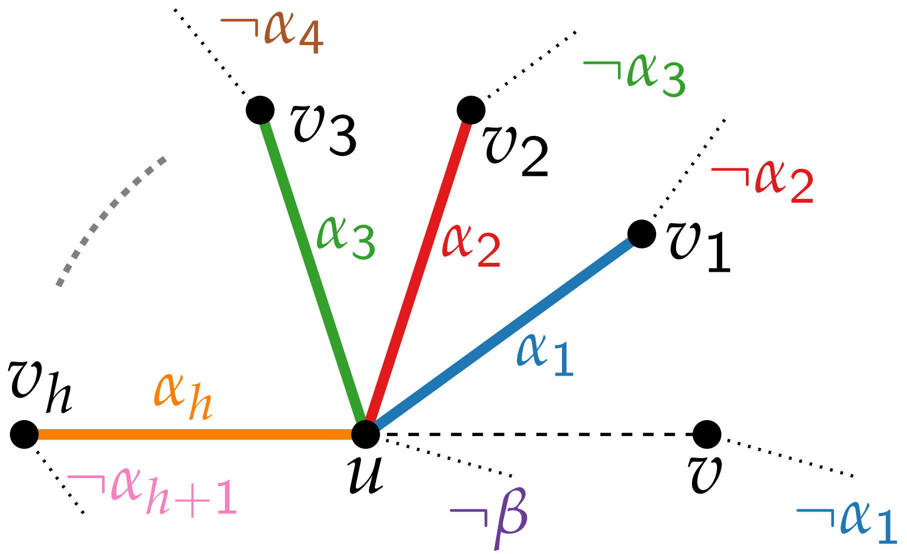
VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i + 1$
  return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum Edge Coloring – Recoloring

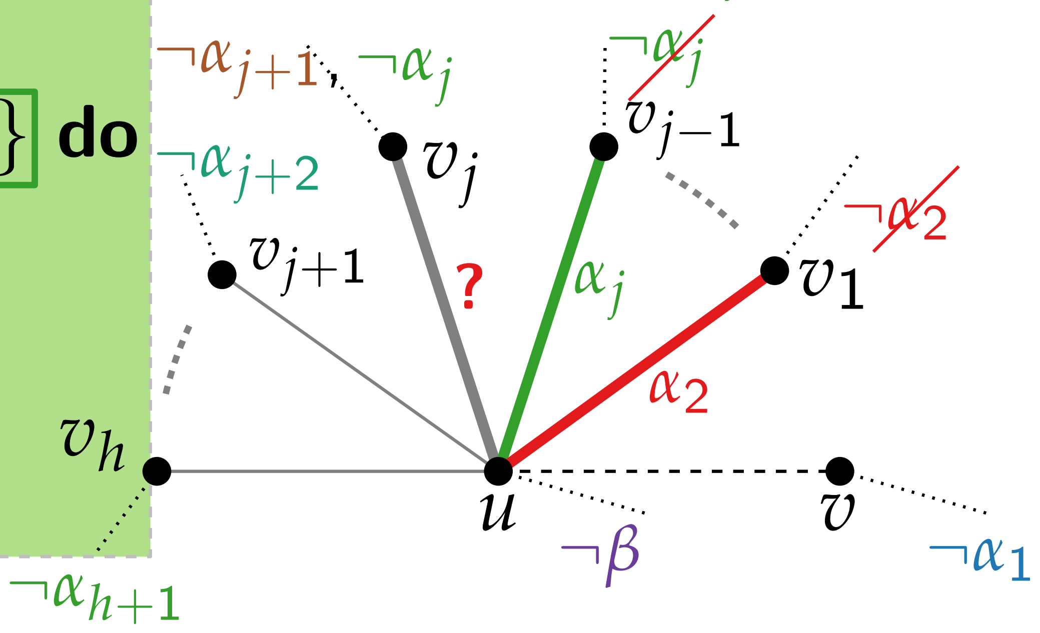
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

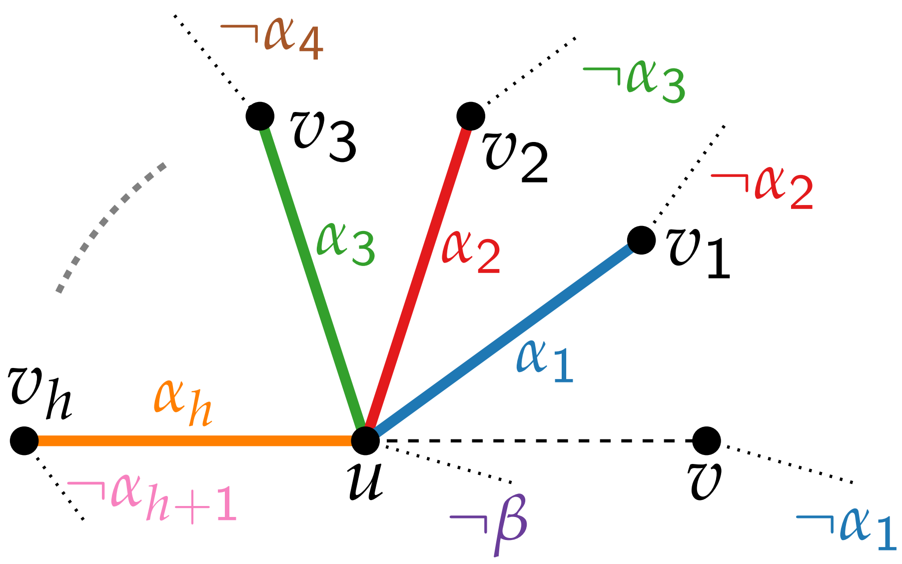
VizingRecoloring(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u) \colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i + 1$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum Edge Coloring – Recoloring

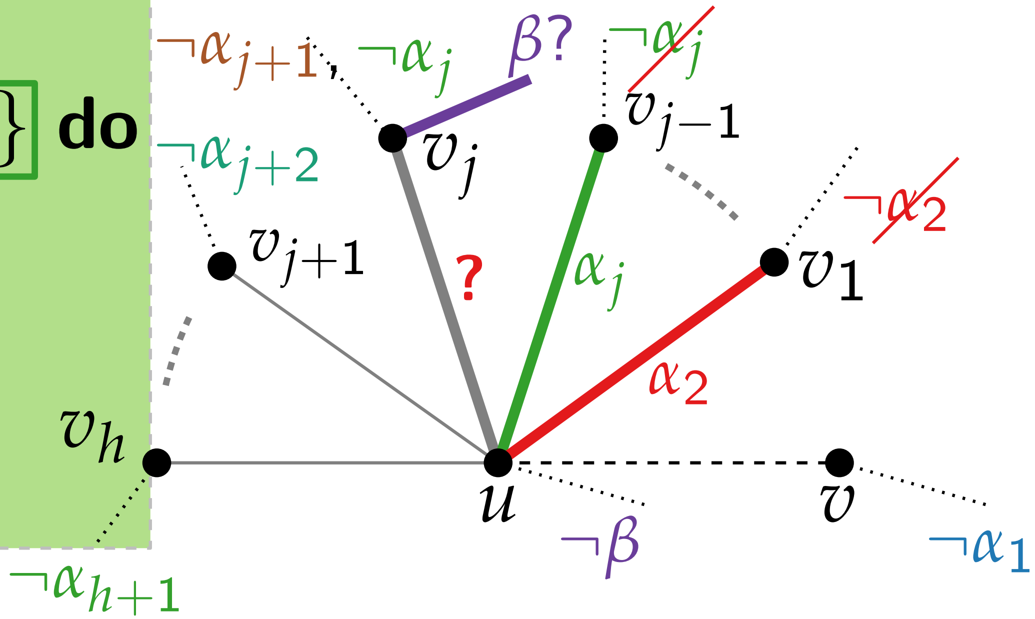
Lemma 2.
Let G be a graph with a $(\Delta + 1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

```
VizingRecoloring(G, c, u, α_1)
  i ← 1
  while ∃w ∈ N(u): c(uw) = α_i ∧ w ∉ {v_1, ..., v_{i-1}} do
      v_i ← w
      α_{i+1} ← min color missing at w
      i ← i + 1
  return v_1, ..., v_i; α_1, ..., α_{i+1}
```

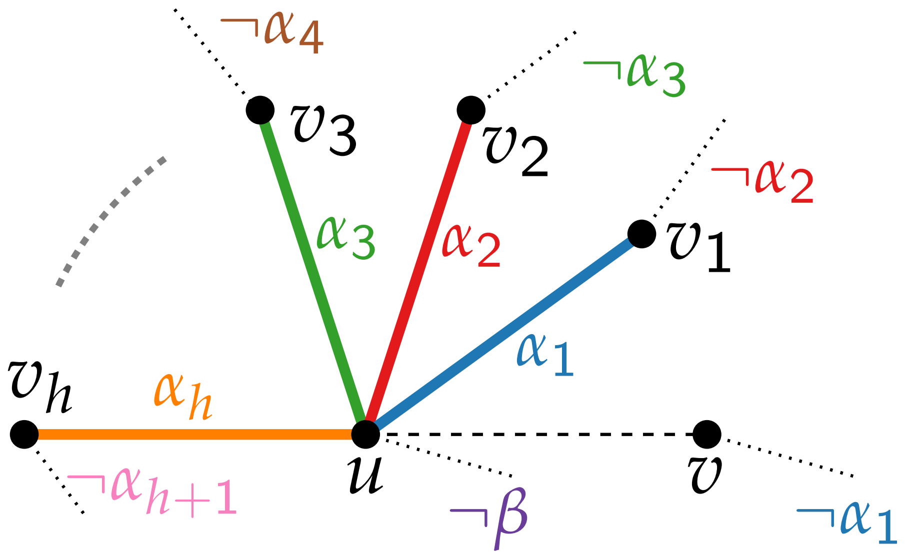
Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$.



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 β and v miss α_1; apply the following algorithm:



VizingRecoloring(G, c, u, α_1)

$i \leftarrow 1$

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

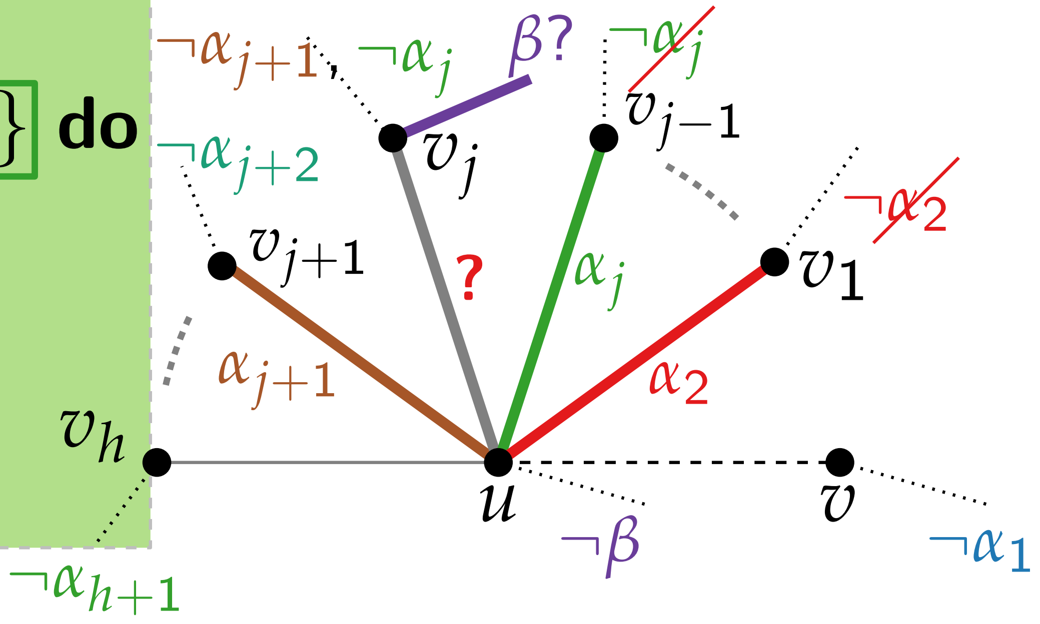
  $v_i \leftarrow w$

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

  $i \leftarrow i + 1$

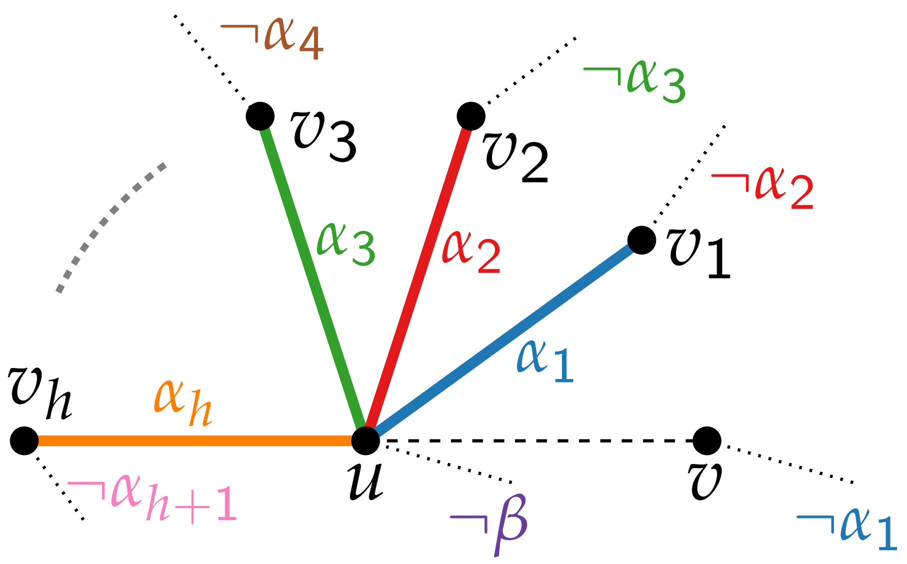
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum Edge Coloring – Recoloring

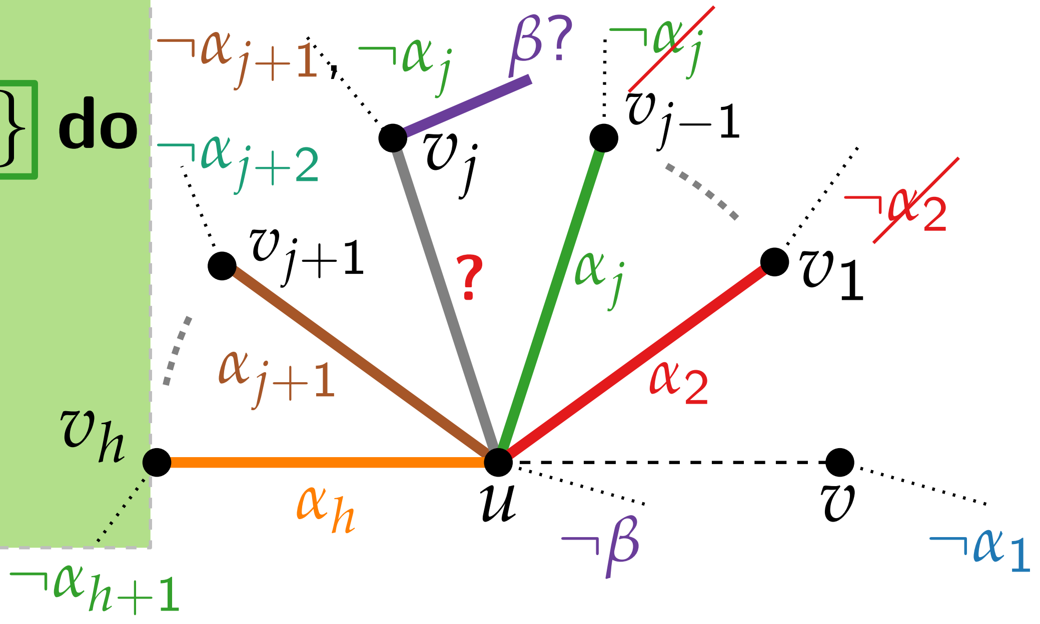
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

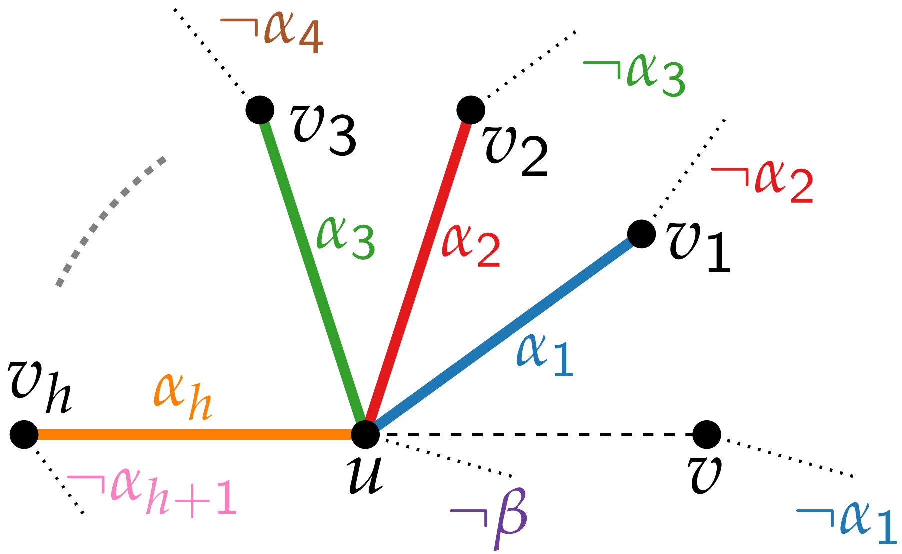
VizingRecoloring(G, c, u, α_1)
- $i \leftarrow 1$
- **while** $\exists w \in N(u) \colon c(uw) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i \leftarrow i + 1$
- **return** $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

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



Minimum Edge Coloring – Recoloring

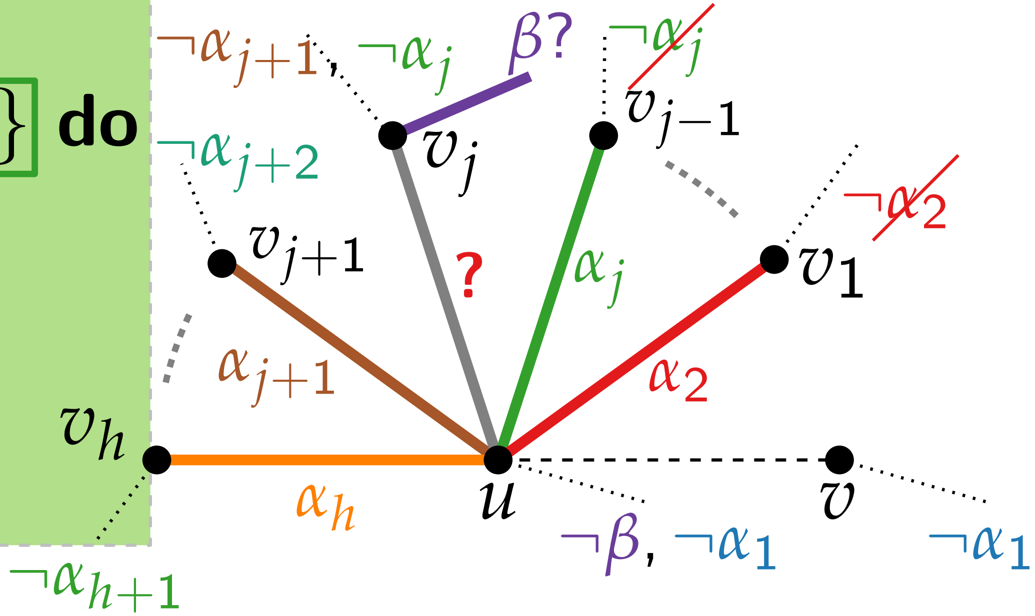
Lemma 2.
Let G be a graph with a $(\Delta+1)$-edge coloring c,
let u, v be non-adjacent vertices with $\deg(u), \deg(v) < \Delta$.
Then c can be changed s.t. u and v miss the same color.



Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

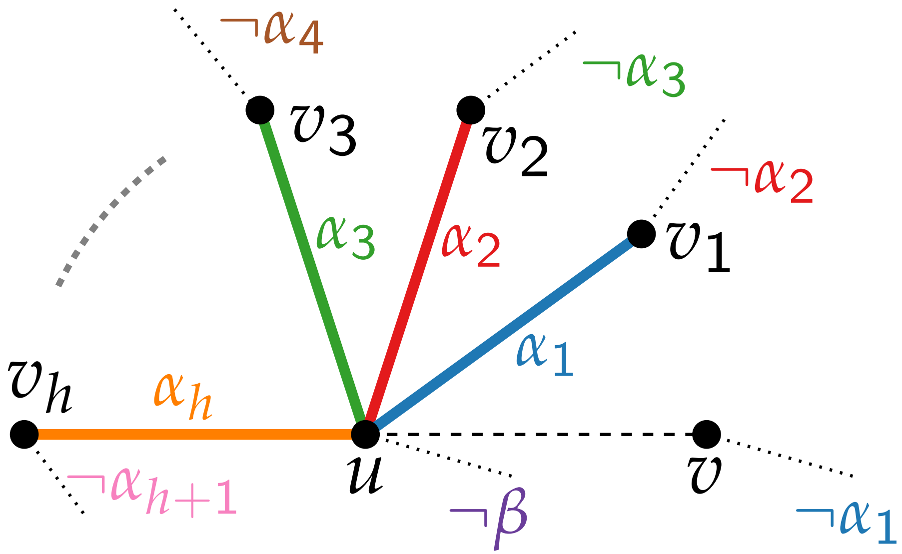
VizingRecoloring(G, c, u, α_1)
  $i \leftarrow 1$
  while $\exists w \in N(u)\colon c(uw) = \alpha_i \wedge w \notin \{v_1, \dots, v_{i-1}\}$ **do**
    $v_i \leftarrow w$
    $\alpha_{i+1} \leftarrow$ min color missing at w
    $i \leftarrow i+1$
  return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



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.

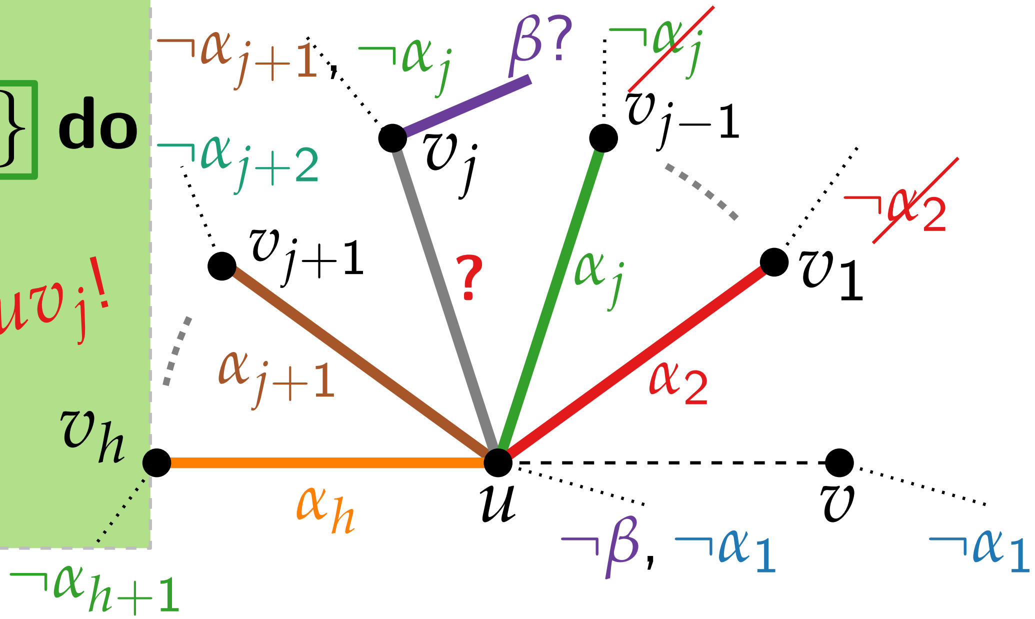


Proof. Note that every vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

```
VizingRecoloring(G, c, u, α_1)
  i ← 1
  while ∃w ∈ N(u): c(uw) = α_i ∧ w ∉ {v_1, ..., v_{i-1}} do
  |   v_i ← w
  |   α_{i+1} ← min color missing at w
  |   i ← i + 1
  return v_1, ..., v_i; α_1, ..., α_{i+1}
```

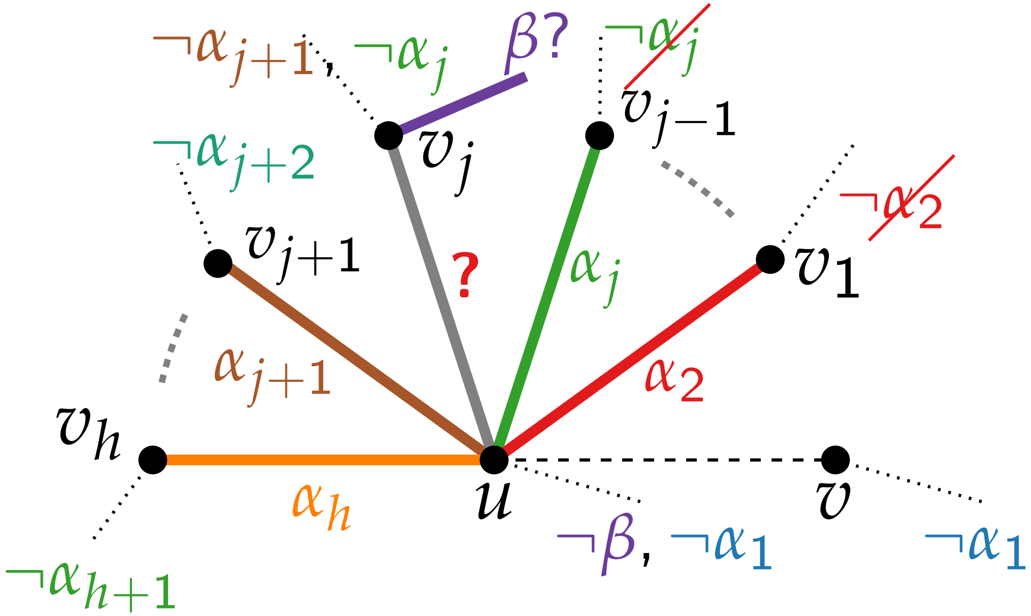
Need color for edge uv_j!

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



Minimum Edge Coloring – Recoloring

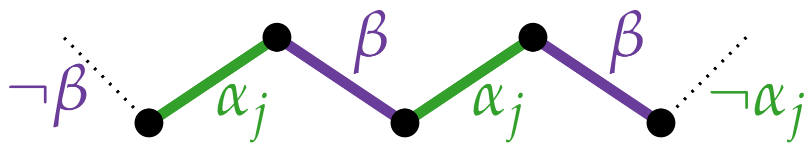
Proof continued for Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$, and we need to find a color for edge uv_j.

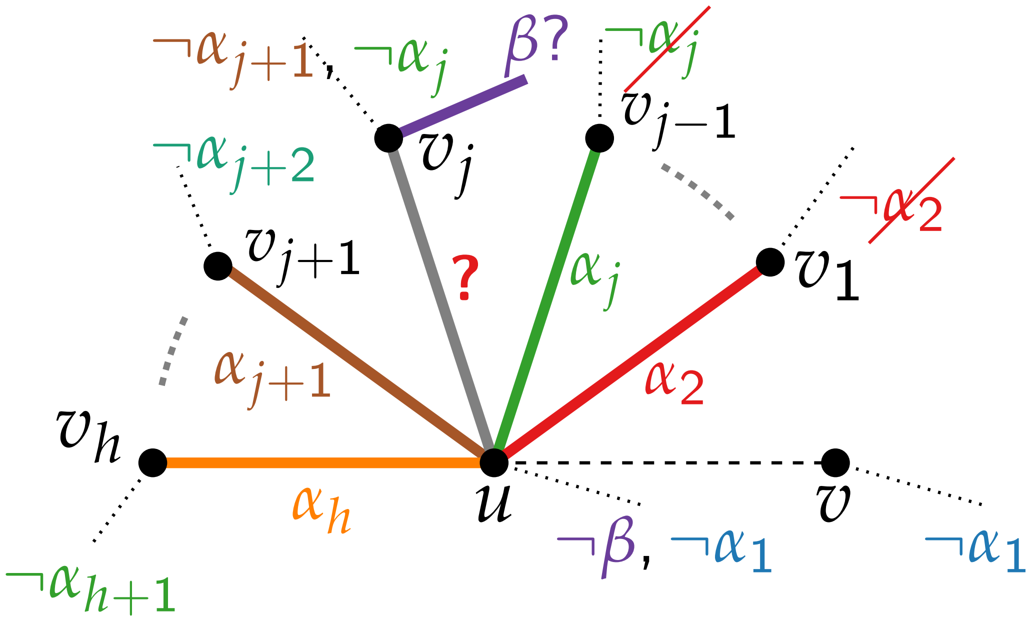


Minimum Edge Coloring – Recoloring

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 β and α_j.

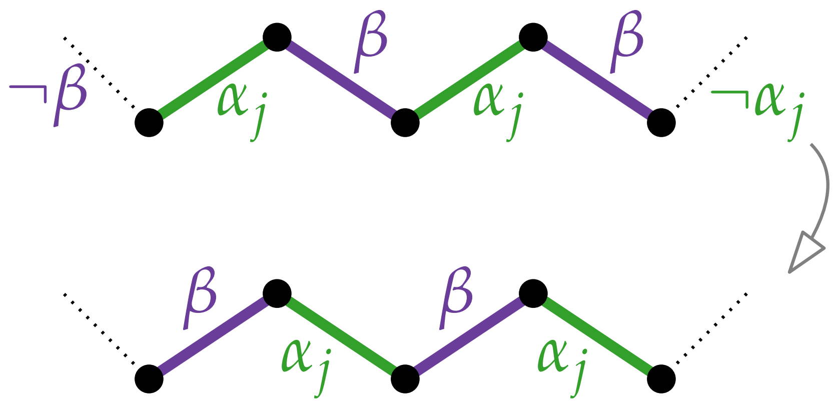


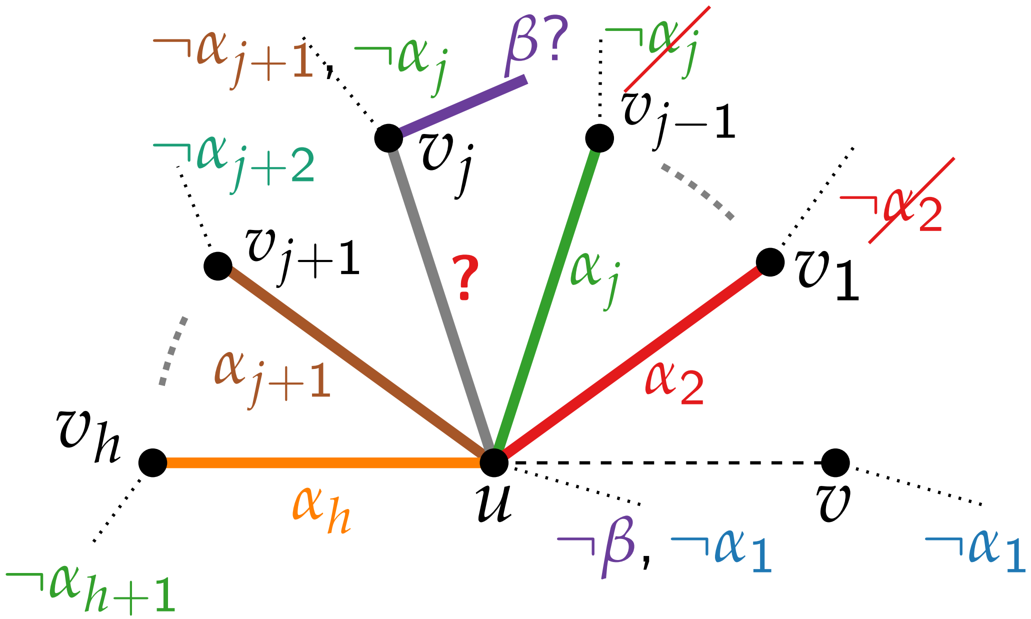


Minimum Edge Coloring – Recoloring

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

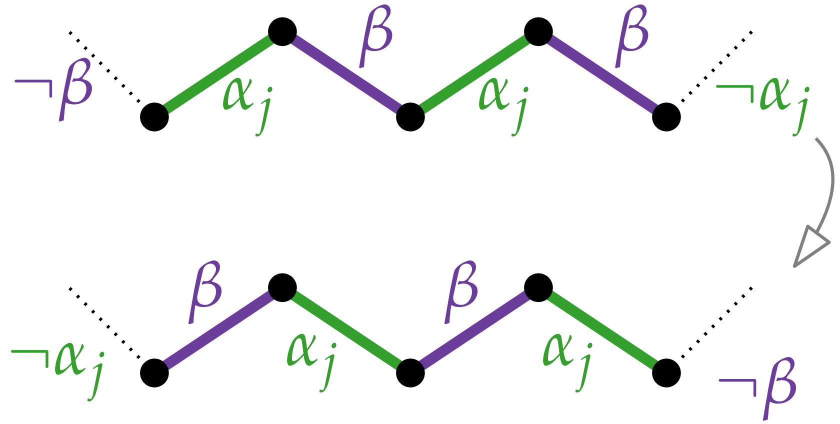


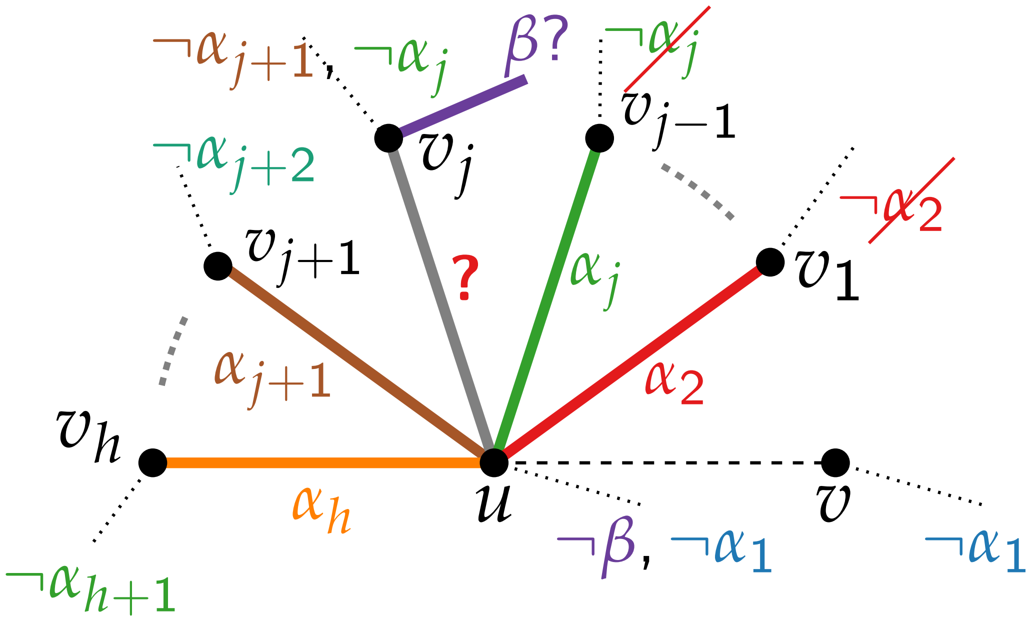


Minimum Edge Coloring – Recoloring

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

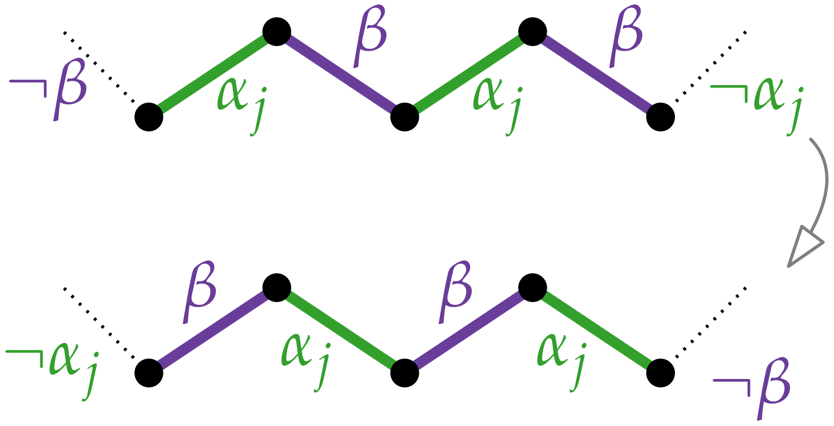


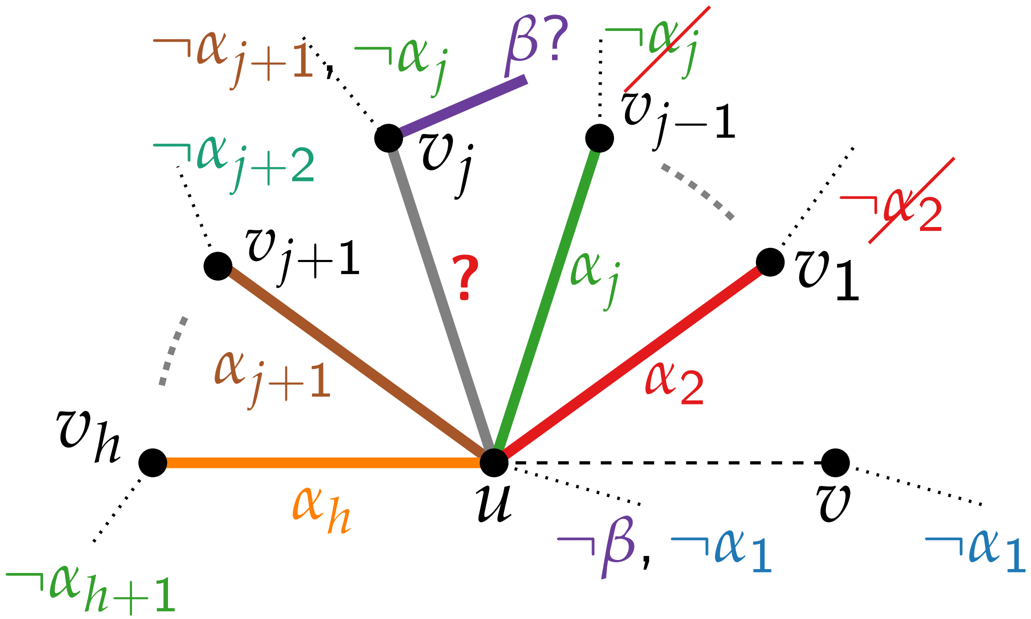


Minimum Edge Coloring – Recoloring

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 β and α_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'.

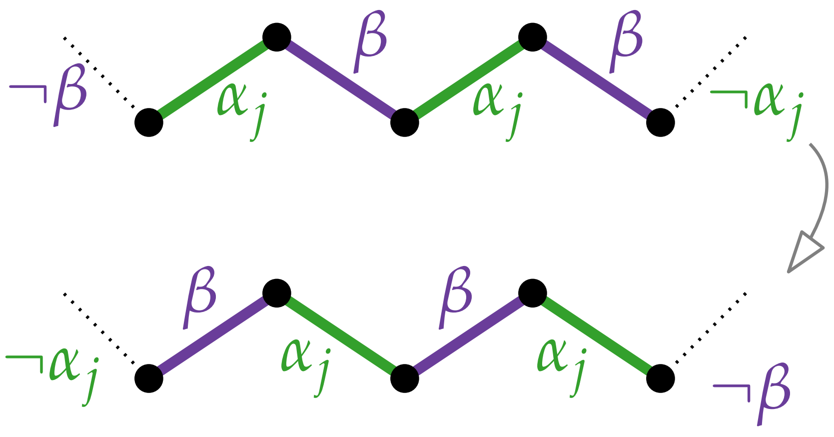


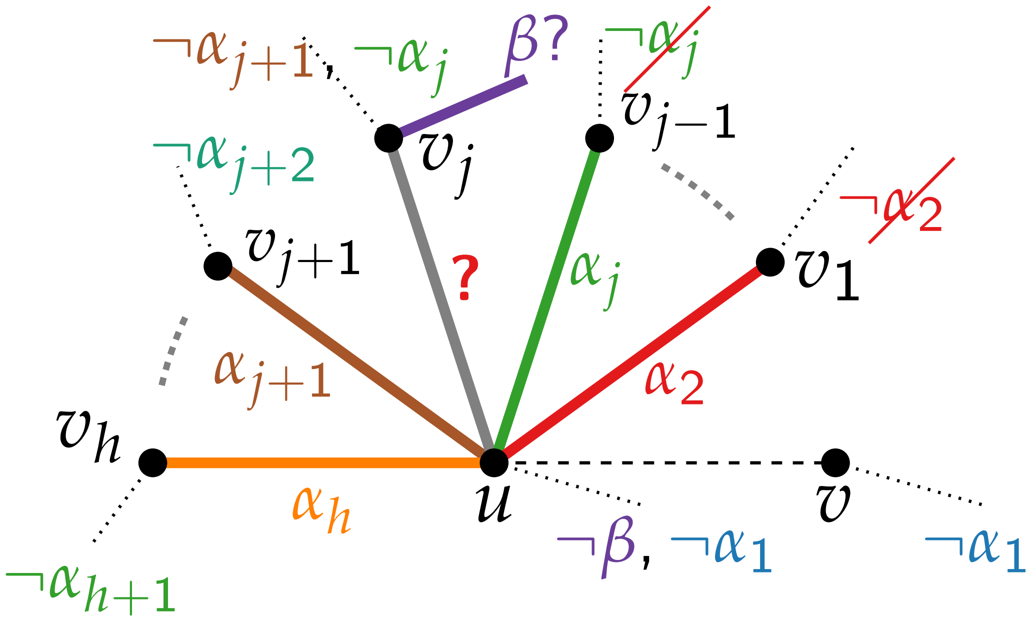


Minimum Edge Coloring – Recoloring

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 β and α_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:

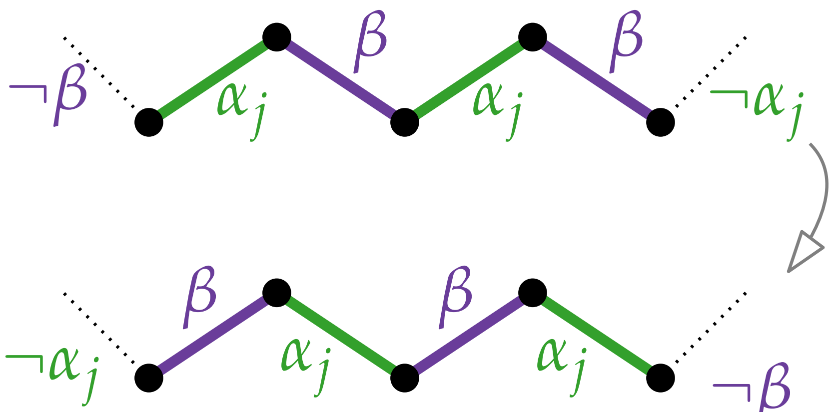


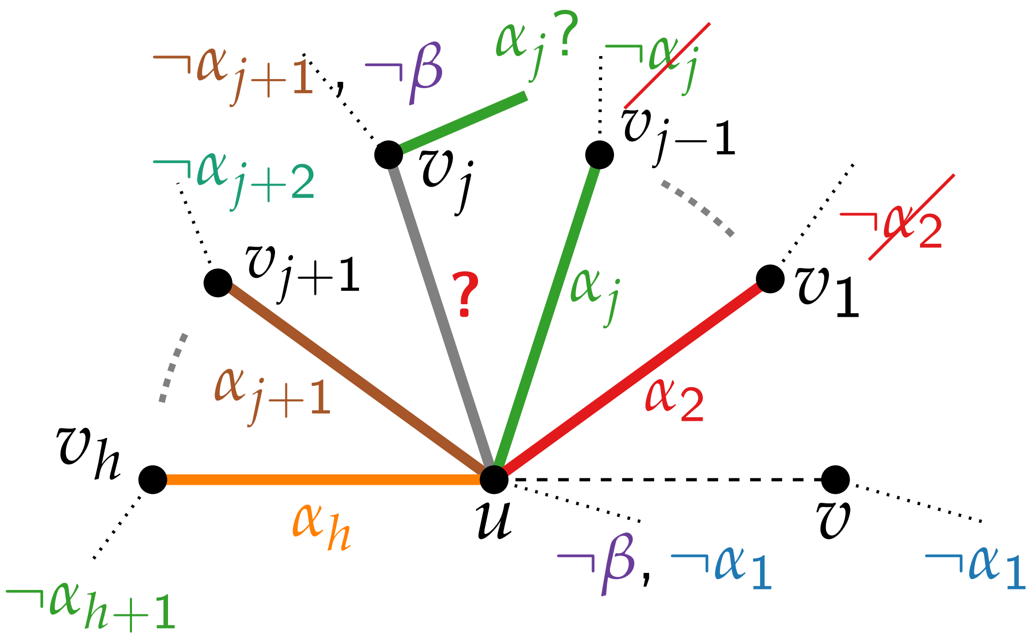


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,

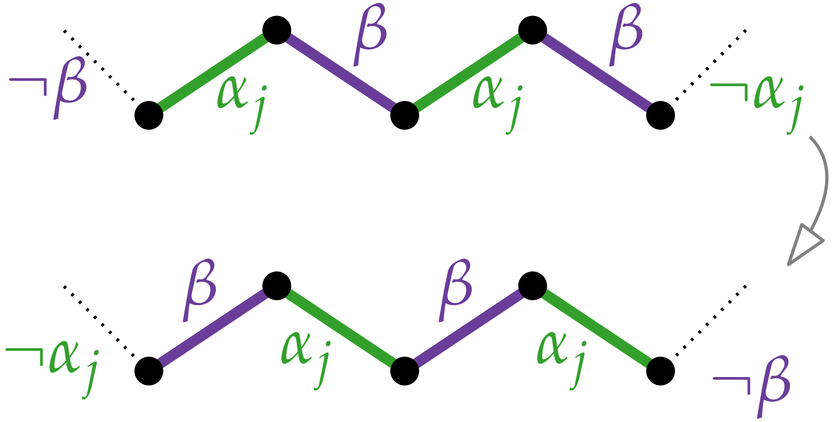


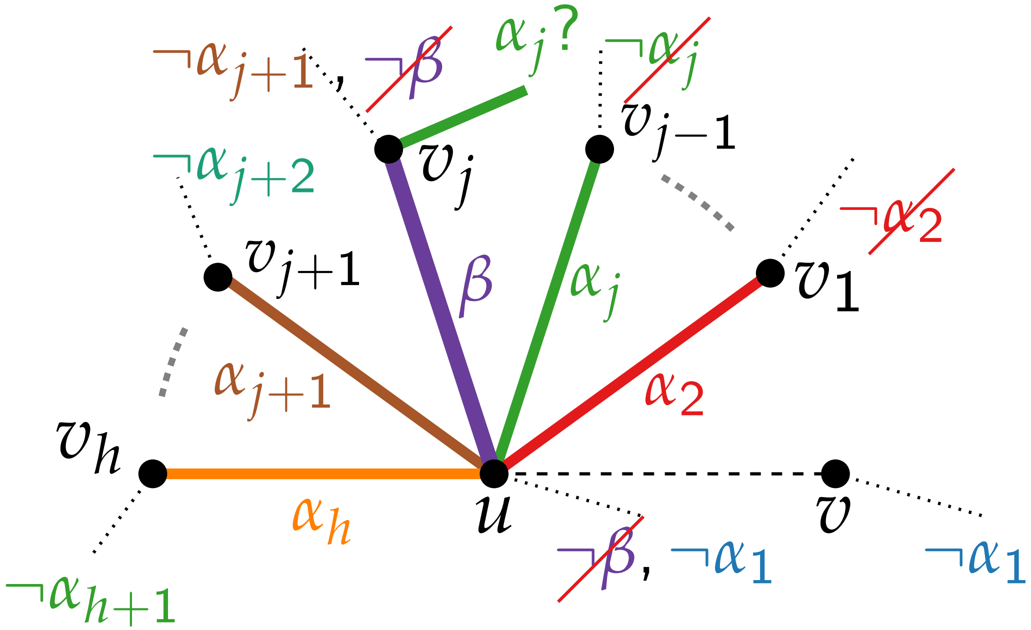


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.

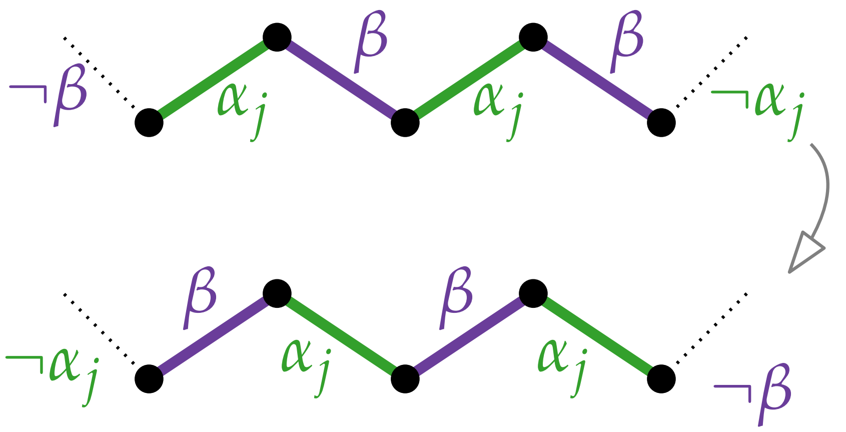


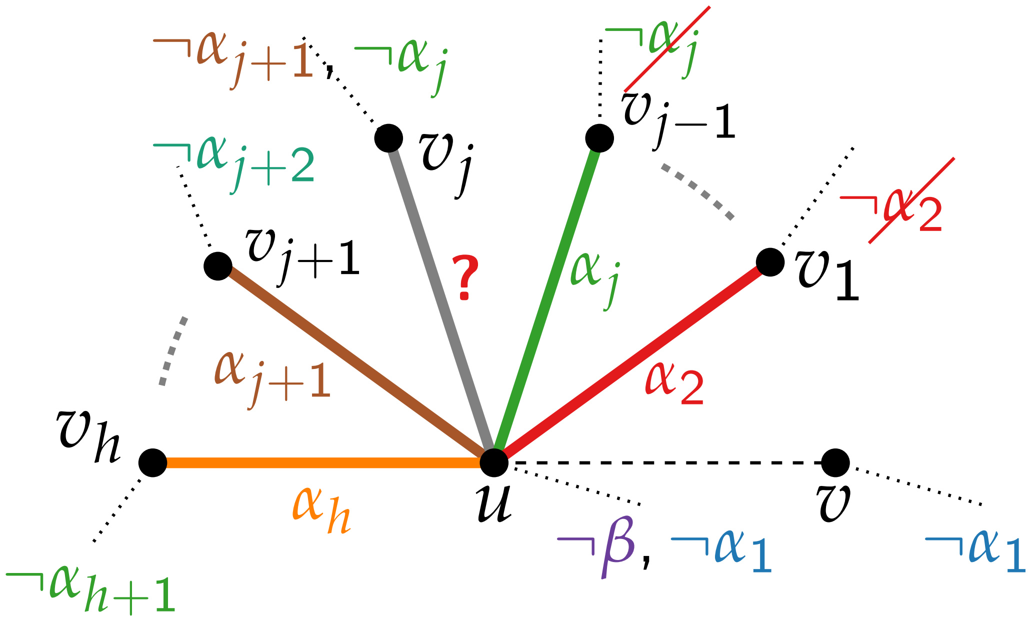


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?

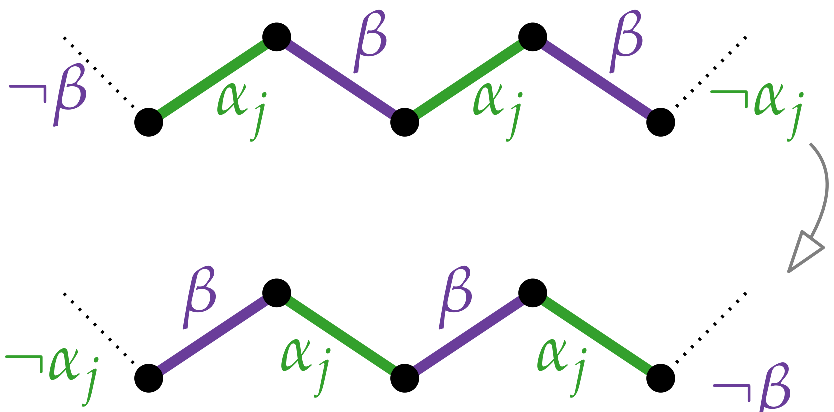


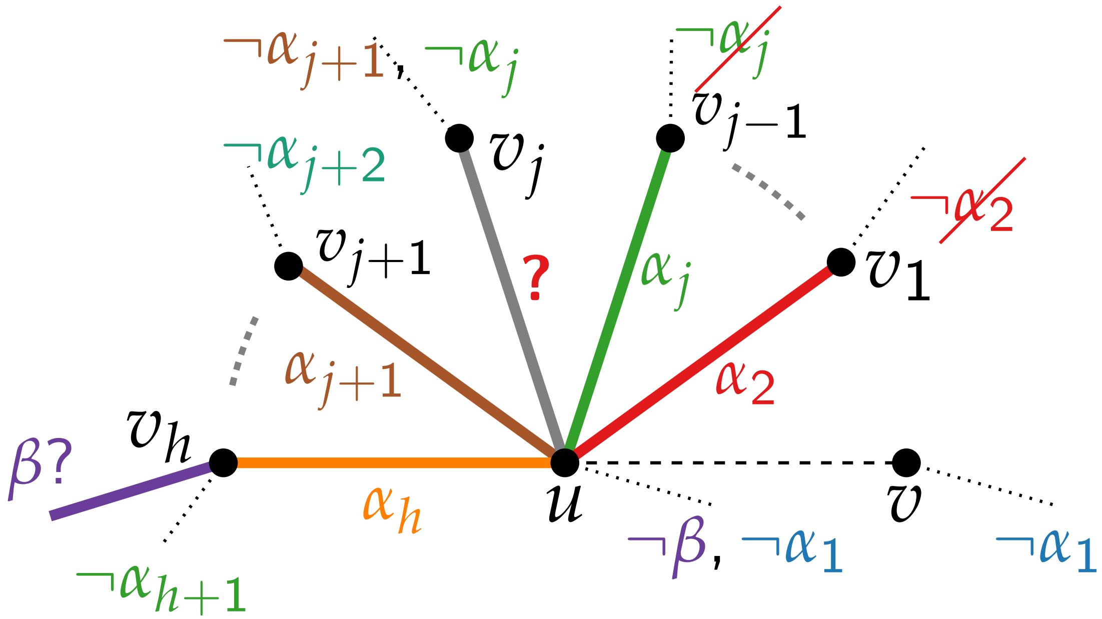


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?
  - re-color component ending at v_h if there is β

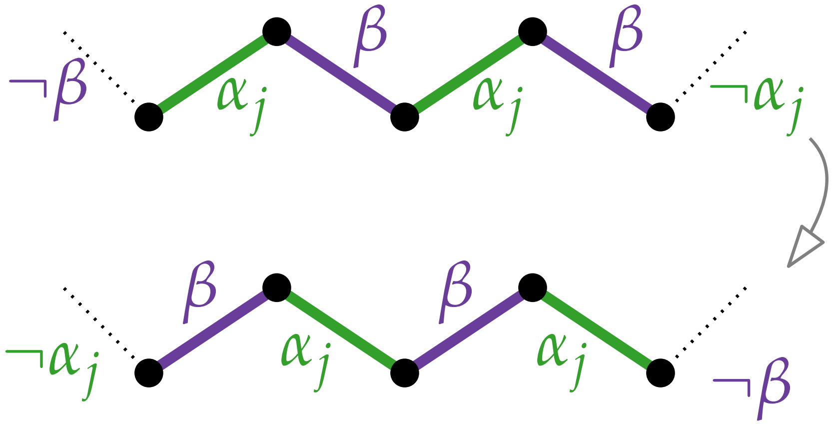


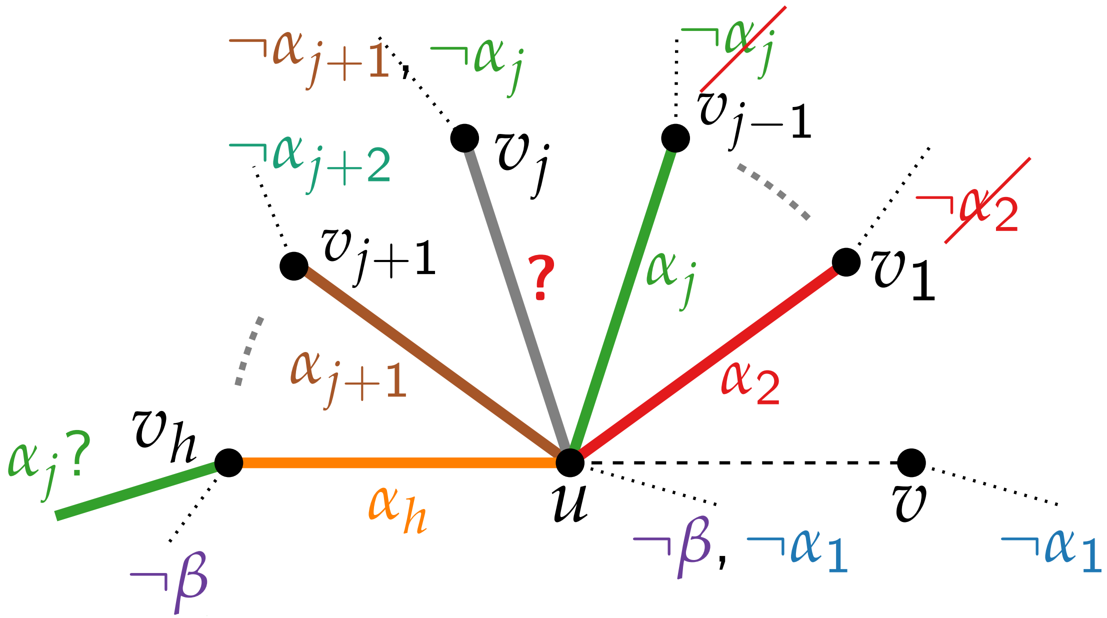


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?
  - re-color component ending at v_h if there is β

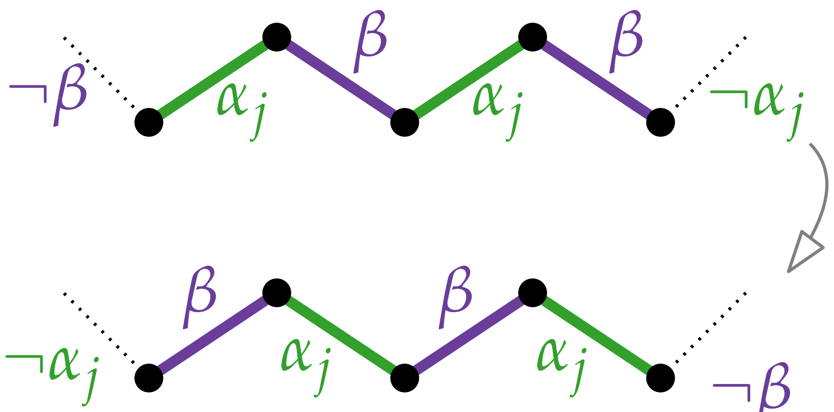


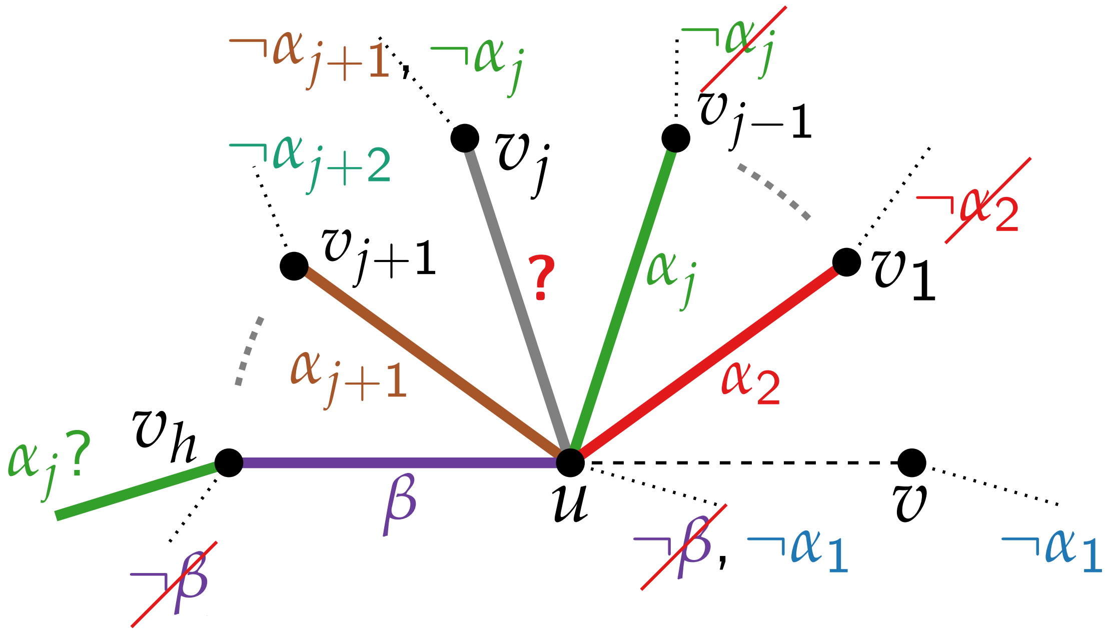


Minimum Edge Coloring – Recoloring

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 β and α_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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?
  - re-color component ending at v_h if there is β
  - color uv_h with β;

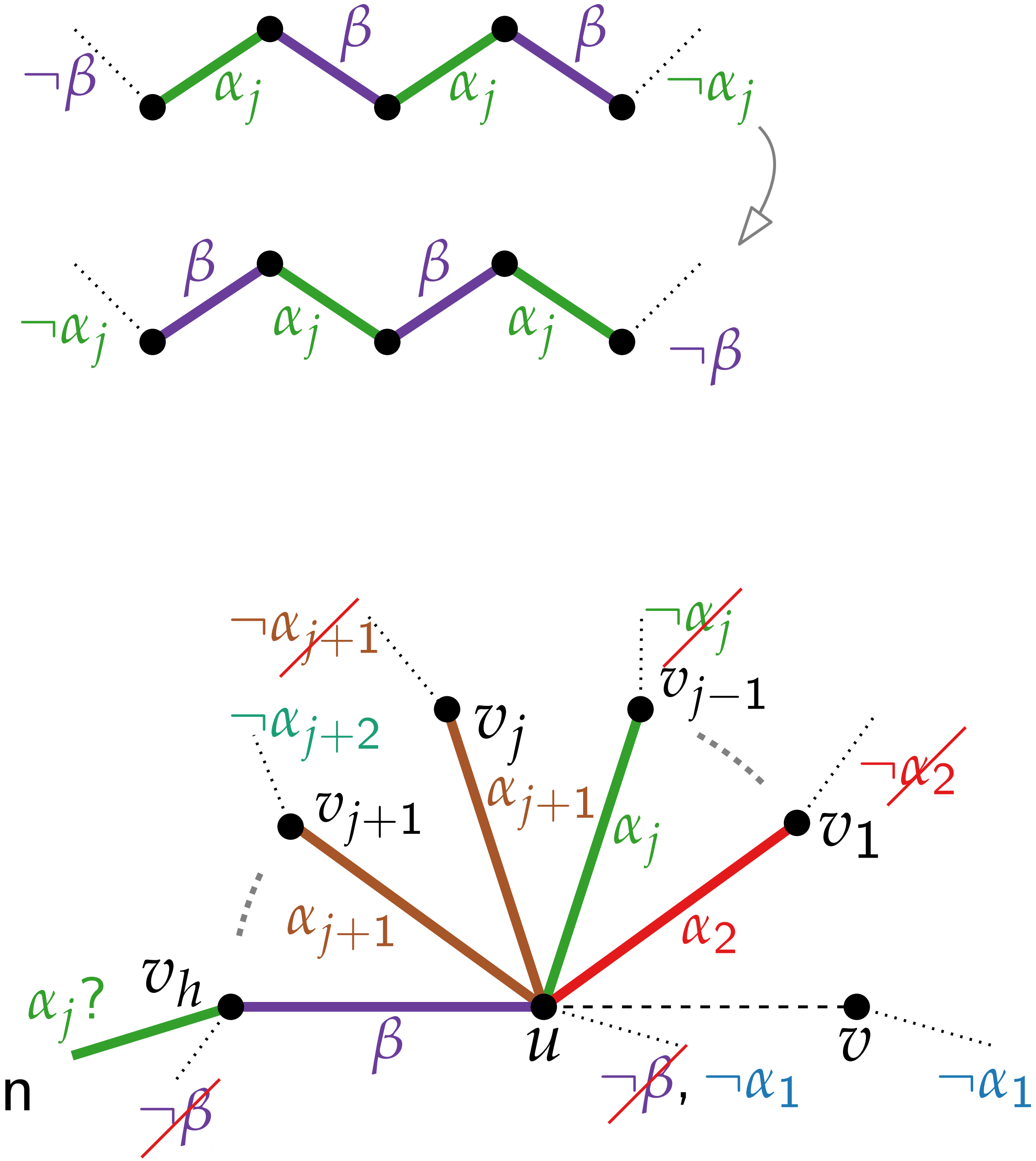




Minimum Edge Coloring – Recoloring

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

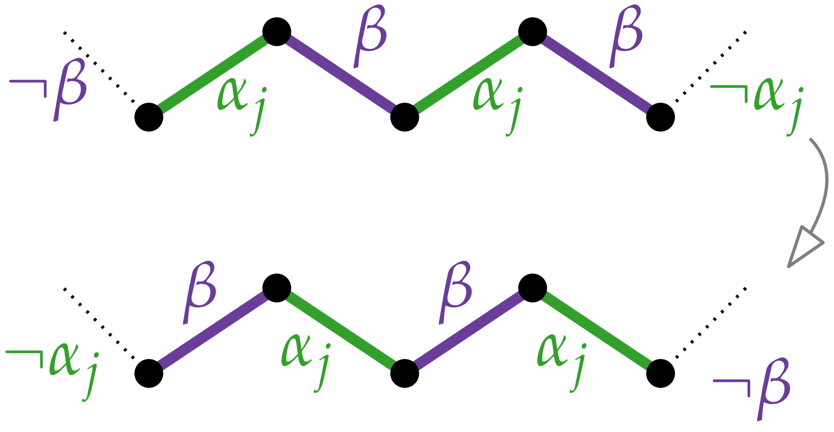
- Consider subgraph G' of G induced by the edges of colors β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- Nodes u, v_j, v_h are all leaves in G'.
  $\Rightarrow$ They are not all in the same component of G'.
- If u and v_j are not in the same component:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?
  - re-color component ending at v_h if there is β
  - color uv_h with β; color uv_j with α_{j+1} and so on

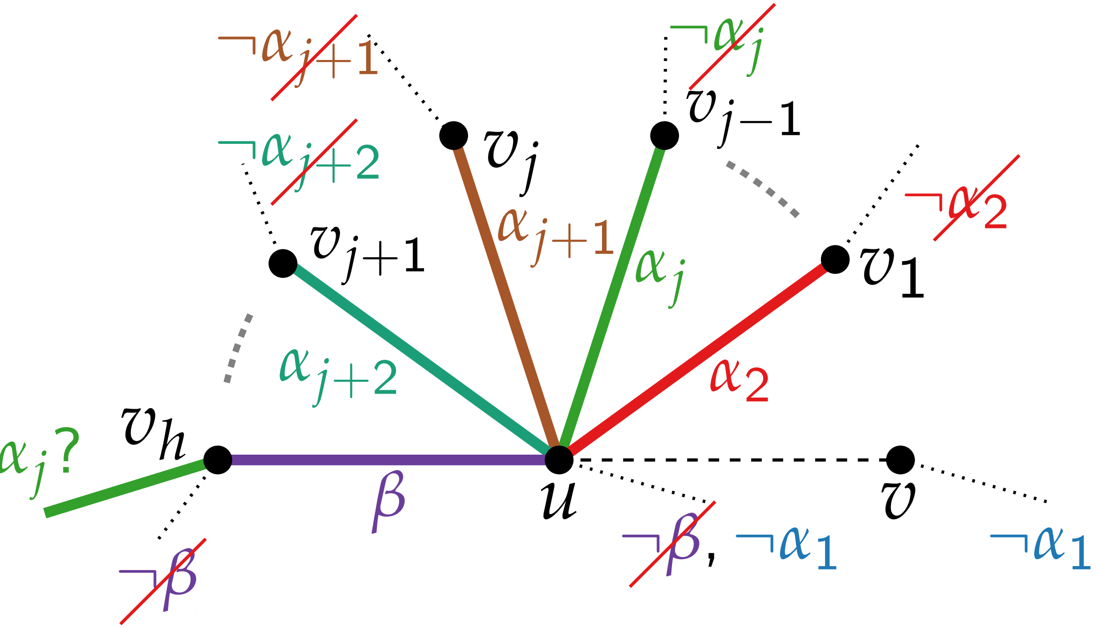


Minimum Edge Coloring – Recoloring

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 β and α_j.
- Since $\Delta(G') \le 2$, 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:
  - re-color component ending at v_j,
  - v_j now misses β; color uv_j with β.
- What if u and v_j are in the same component?
  - re-color component ending at v_h if there is β
  - color uv_h with β; color uv_j with α_{j+1} and so on





Minimum Edge Coloring – Algorithm

```
VizingEdgeColoring(graph G, coloring c ≡ 0)
  if E(G) ≠ ∅ then
      Let e = uv be an arbitrary edge of G.
      G_e ← G − e
      VizingEdgeColoring(G_e, c)
      if Δ(G_e) < Δ(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 ≡ 0)
  if E(G) ≠ ∅ then
    Let e = uv be an arbitrary edge of G.
    G_e ← G − e
    VizingEdgeColoring(G_e, c)
    if Δ(G_e) < Δ(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.
```

Theorem 4.
VizingEdgeColoring is an approximation algorithm with additive approximation guarantee $\mathrm{ALG}(G) - \mathrm{OPT}(G) \leq 1$.

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

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

We call α 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!

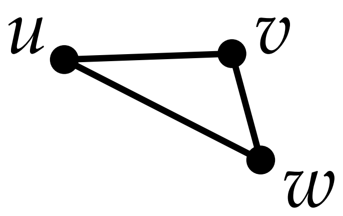
Definition.
maximization
Let Π be a minimization problem, and let $\alpha \in \mathbb{Q}^+$.
A **factor-α approximation algorithm** for Π is a polynomial-time algorithm $\mathcal{A}$ that computes, for every instance I of Π, a solution of value $\mathrm{ALG}(I)$ such that

$$\frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)} \overset{\geq}{\leq} \alpha.$$

We call α the **approximation factor** of $\mathcal{A}$.

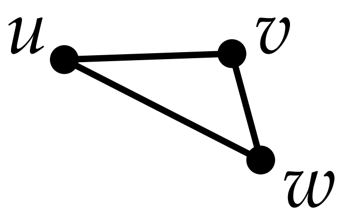
2-Approximation for Metric TSP (from AGT)

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



2-Approximation for Metric TSP (from AGT)

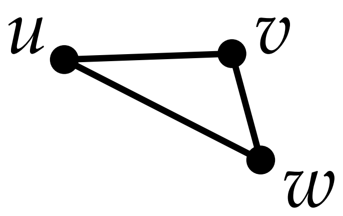
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

2-Approximation for Metric TSP (from AGT)

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

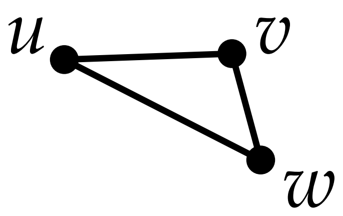


Output. A shortest Hamiltonian cycle in G.

Algorithm.

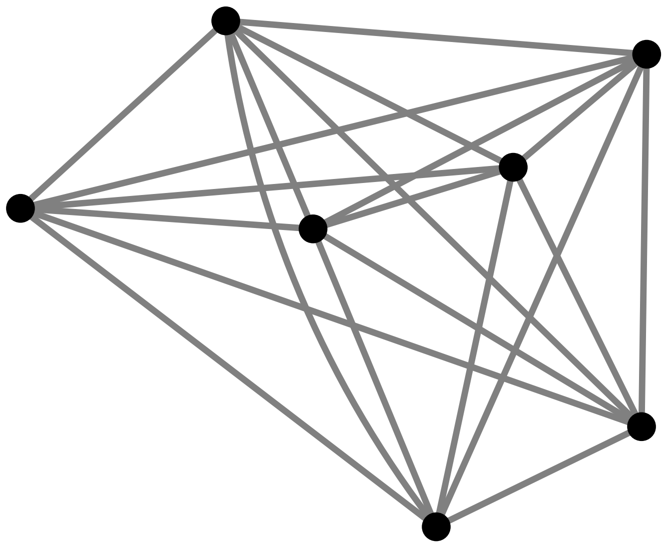
2-Approximation for Metric TSP (from AGT)

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



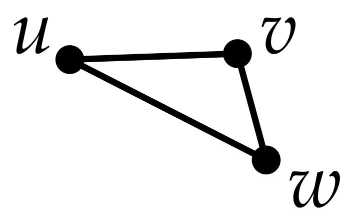
Output. A shortest Hamiltonian cycle in G.

Algorithm.



2-Approximation for Metric TSP (from AGT)

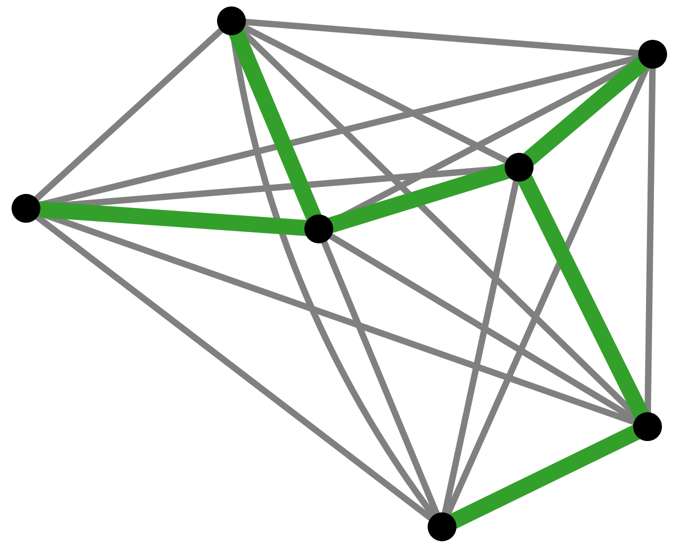
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

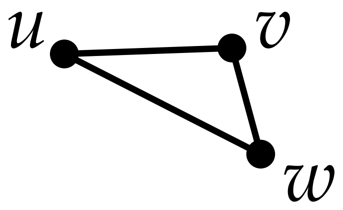
Algorithm.

- Compute MST.



2-Approximation for Metric TSP (from AGT)

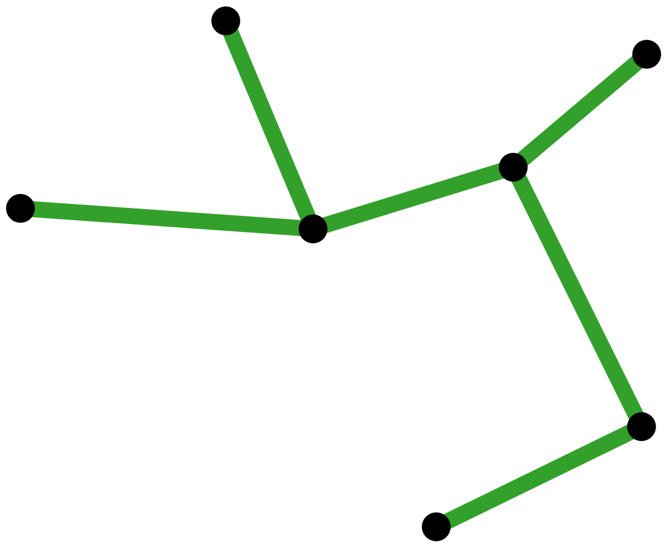
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

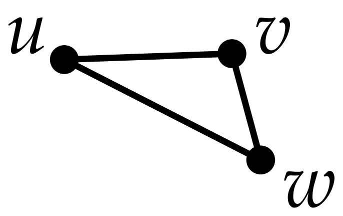
Algorithm.

- Compute MST.



2-Approximation for Metric TSP (from AGT)

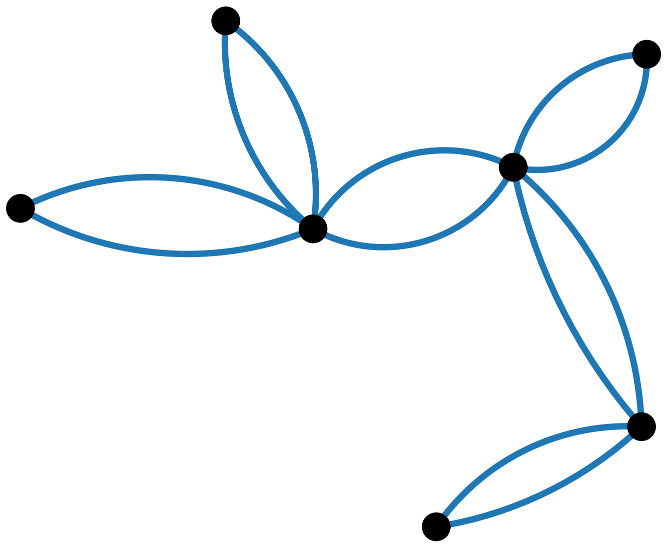
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u,w) \leq d(u,v) + d(v,w)$.



Output. A shortest Hamiltonian cycle in G.

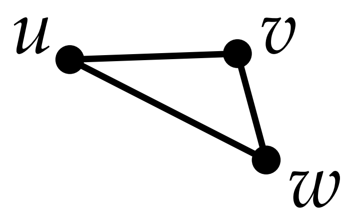
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle



2-Approximation for Metric TSP (from AGT)

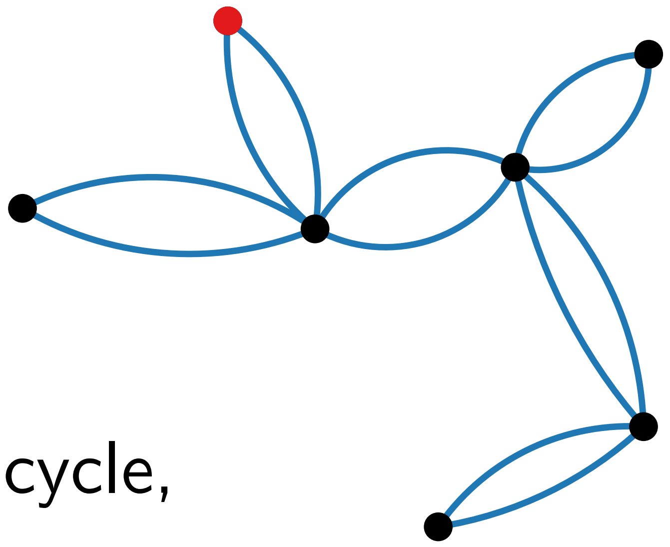
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u,w) \leq d(u,v) + d(v,w)$.



Output. A shortest Hamiltonian cycle in G.

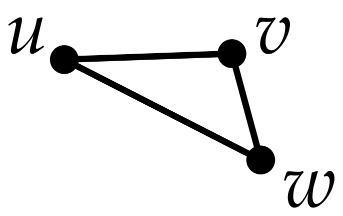
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,



2-Approximation for Metric TSP (from AGT)

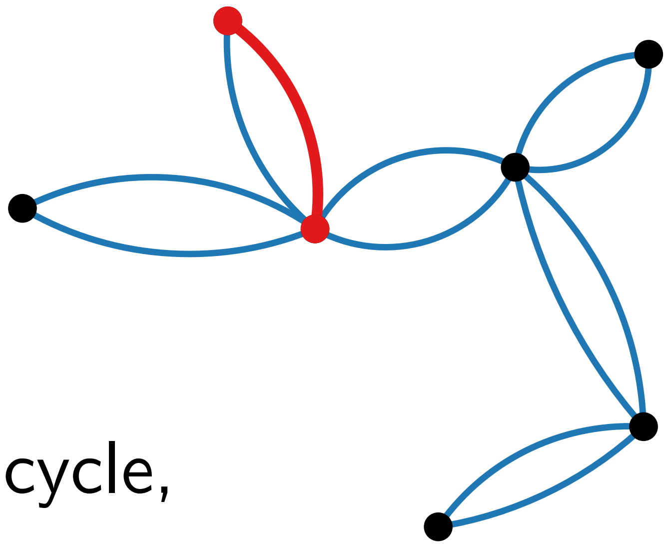
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

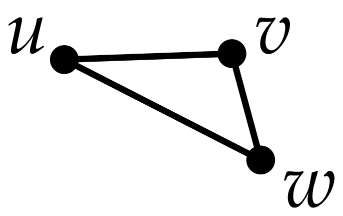
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,



2-Approximation for Metric TSP (from AGT)

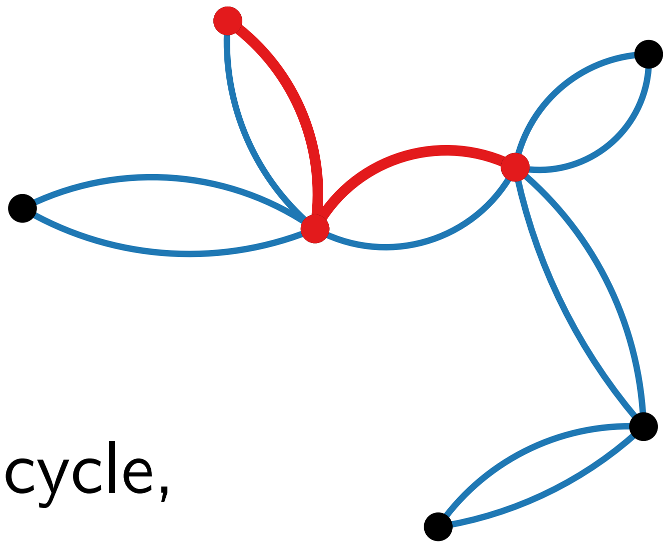
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

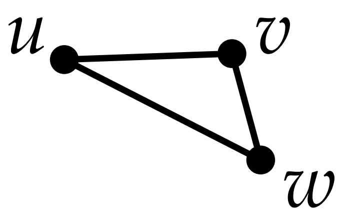
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,



2-Approximation for Metric TSP (from AGT)

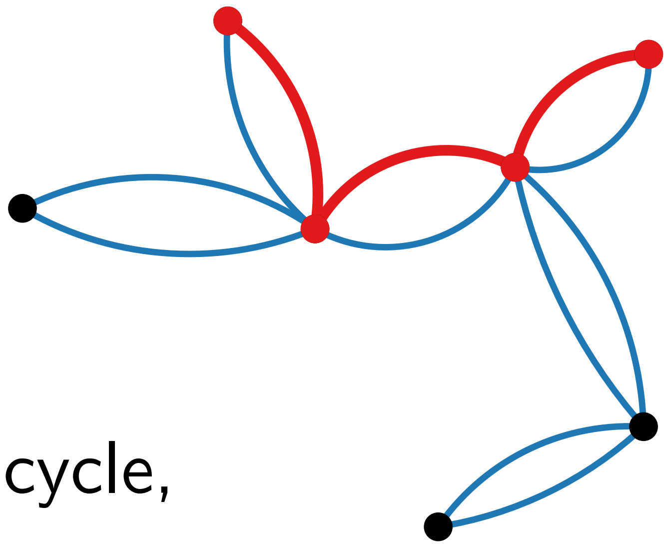
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

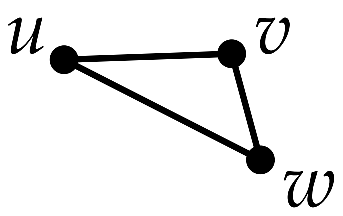
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,



2-Approximation for Metric TSP (from AGT)

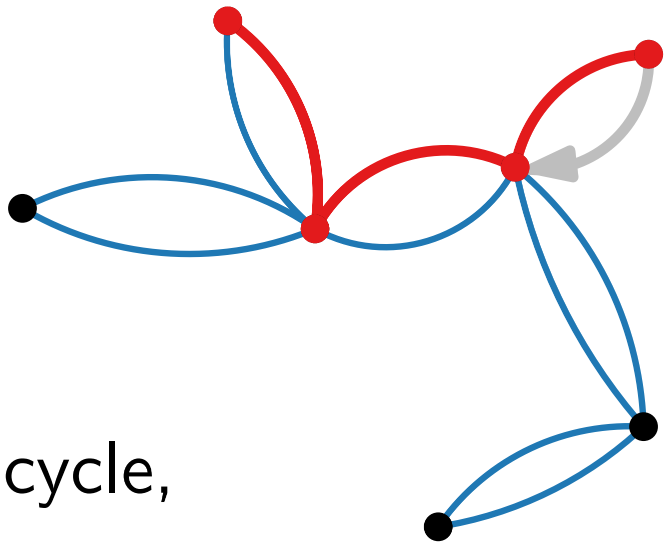
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

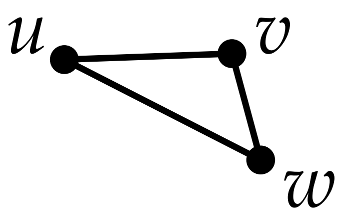
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices



2-Approximation for Metric TSP (from AGT)

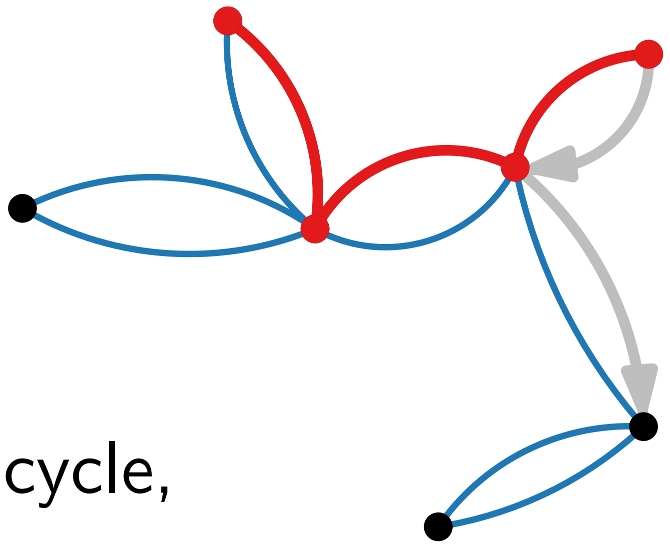
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

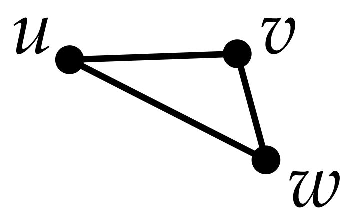
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices



2-Approximation for Metric TSP (from AGT)

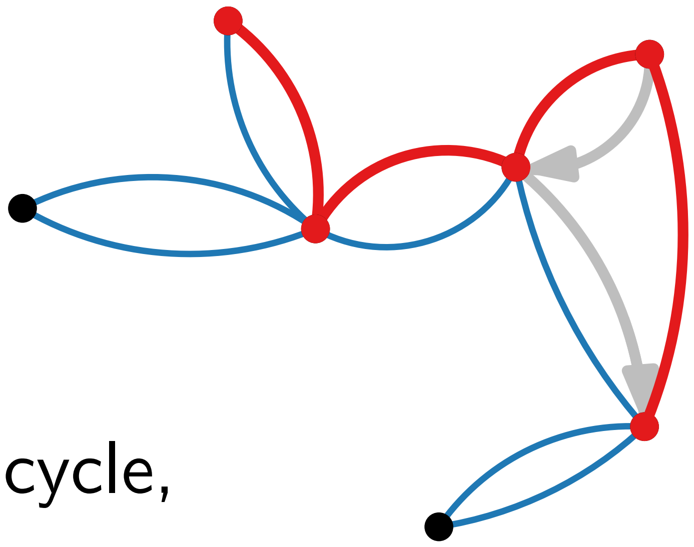
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

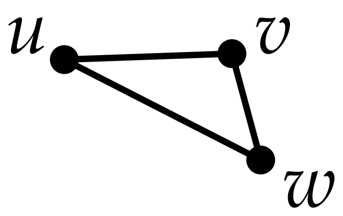
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

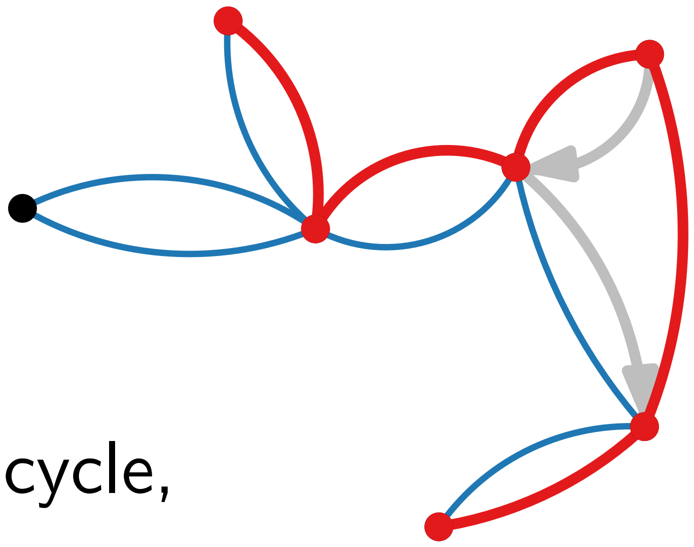
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

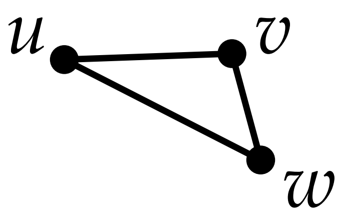
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

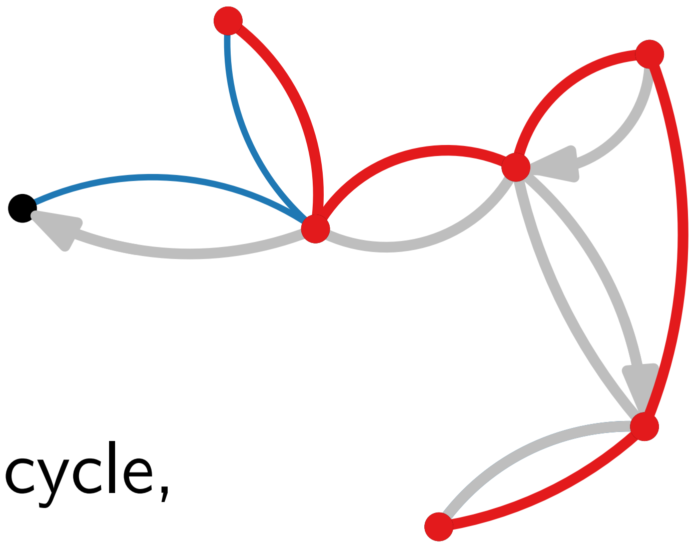
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

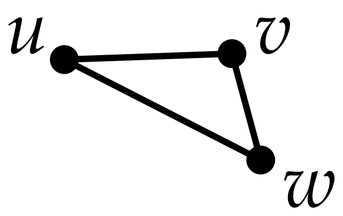
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

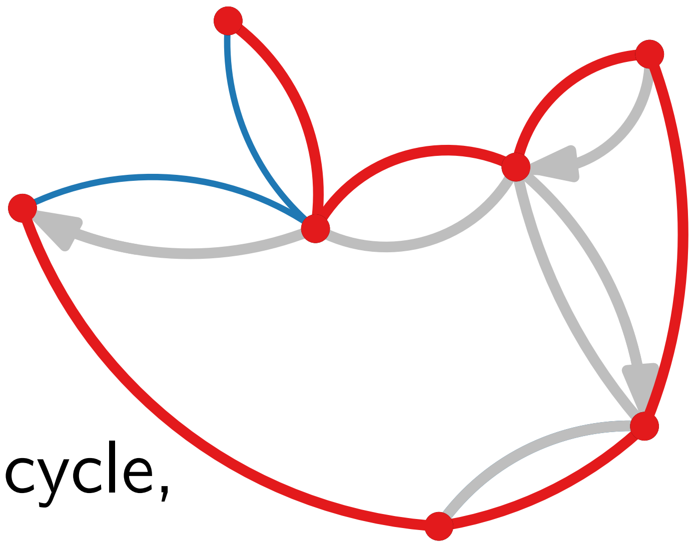
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

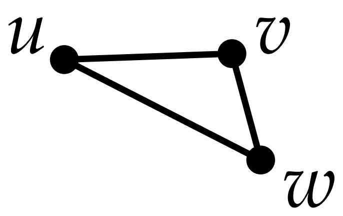
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

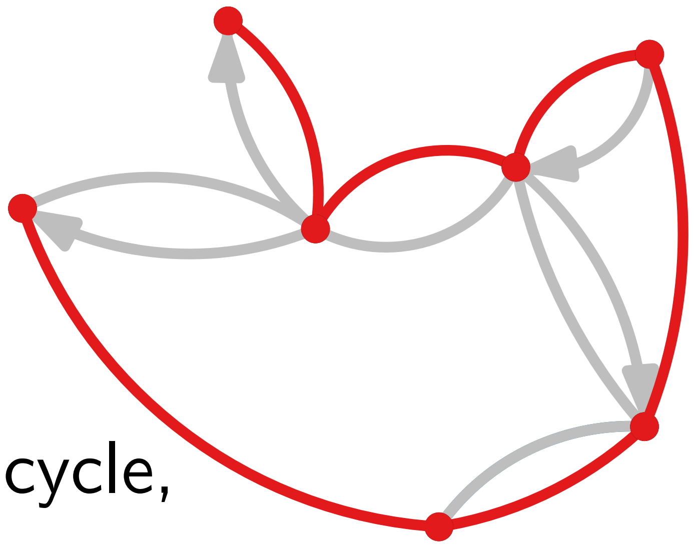
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u,w) \leq d(u,v) + d(v,w)$.



Output. A shortest Hamiltonian cycle in G.

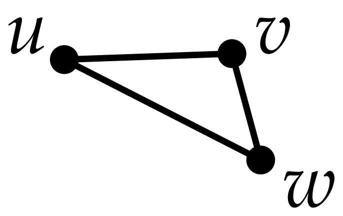
Algorithm.

- Compute MST.
- Double edges. $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

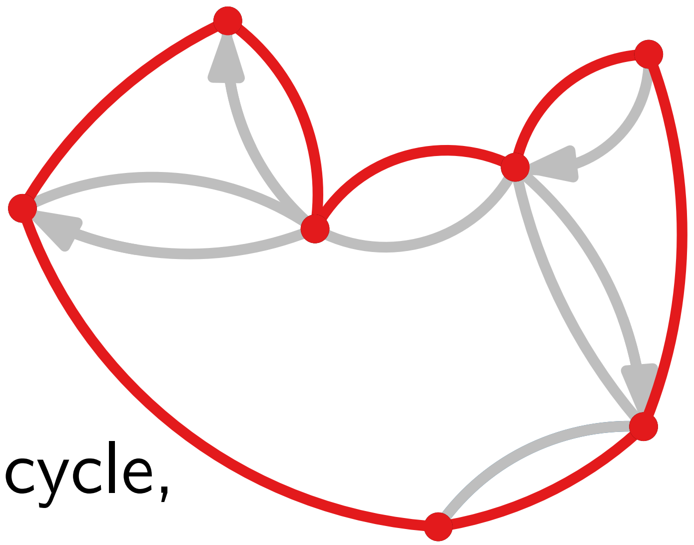
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u,w) \leq d(u,v) + d(v,w)$.



Output. A shortest Hamiltonian cycle in G.

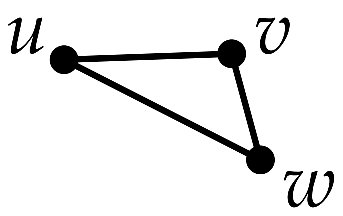
Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



2-Approximation for Metric TSP (from AGT)

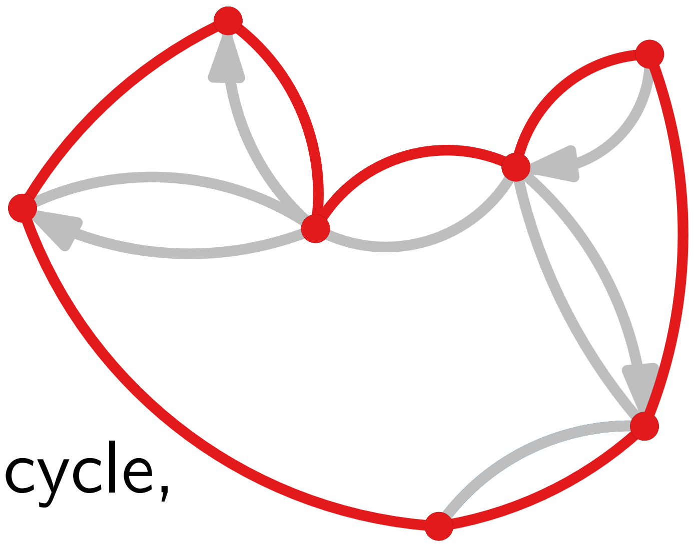
Input. Complete graph $G = (V, E)$ and a distance function $d\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V\colon d(u, w) \leq d(u, v) + d(v, w)$.



Output. A shortest Hamiltonian cycle in G.

Algorithm.

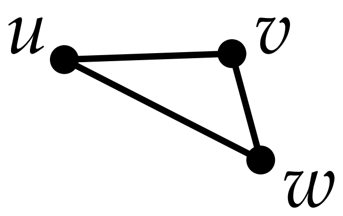
- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



Theorem 5.
The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.

2-Approximation for Metric TSP (from AGT)

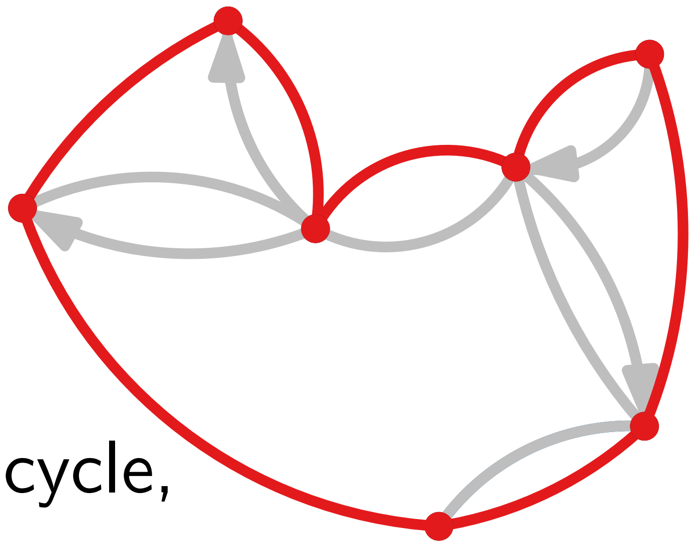
Input. Complete graph $G = (V, E)$ and a distance function $d \colon E \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, i.e., $\forall u, v, w \in V \colon d(u,w) \leq d(u,v) + d(v,w)$.



Output. A shortest Hamiltonian cycle in G.

Algorithm.

- Compute MST.
- Double edges.
  $\Rightarrow$ Eulerian cycle
- Walk along Eulerian cycle,
- skipping visited vertices
- and adding shortcuts.



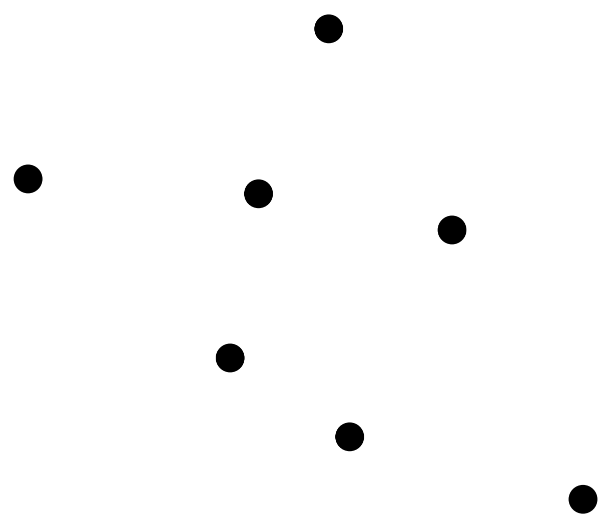
Theorem 5.
The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.

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

Nearest Addition Algorithm for Metric TSP

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

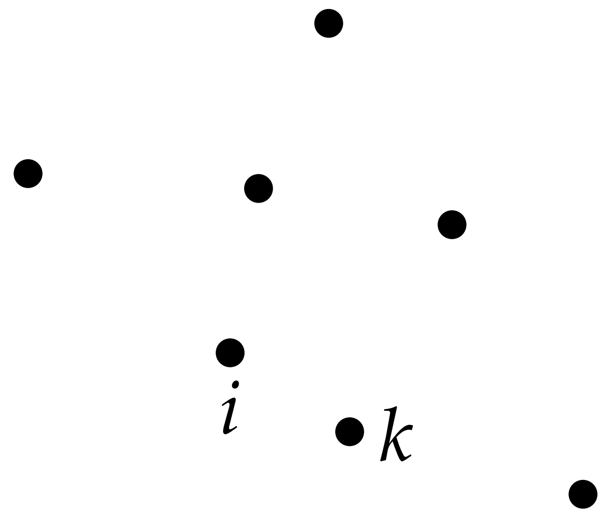
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

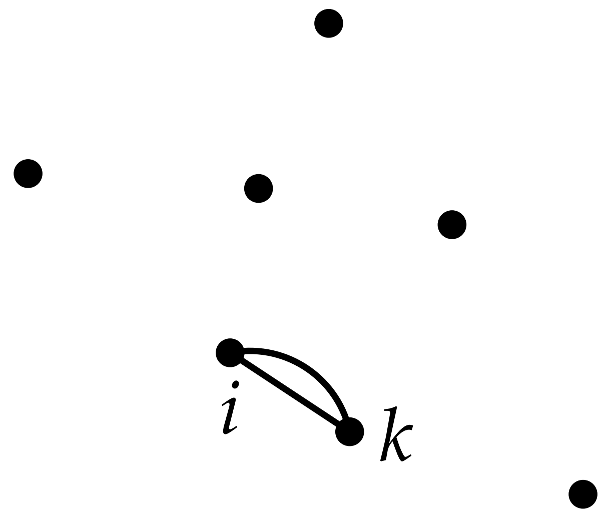
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
- Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
- Let k be vertex after i in T.
- Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

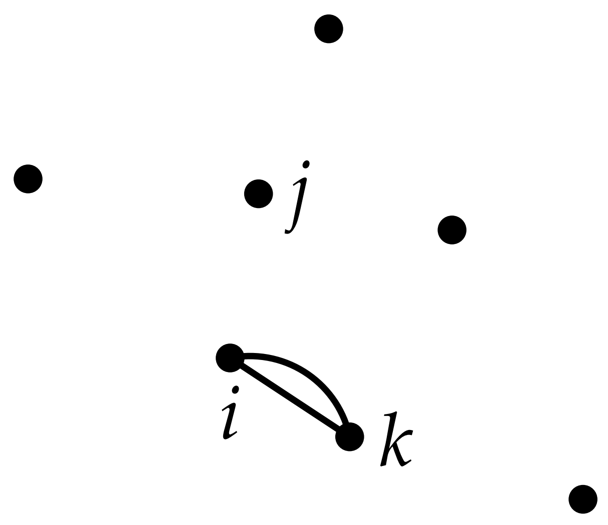
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

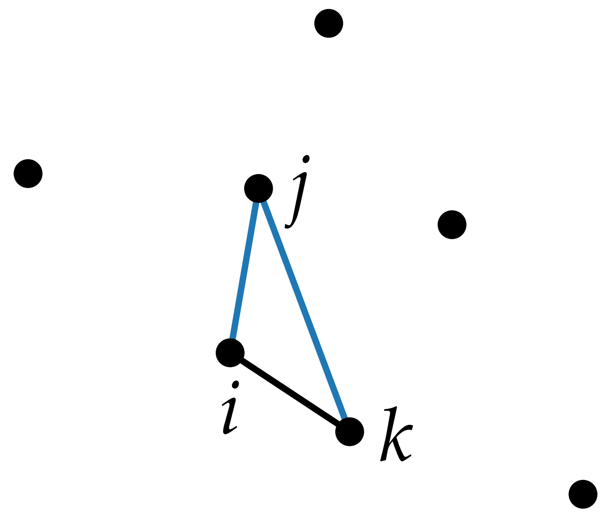
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

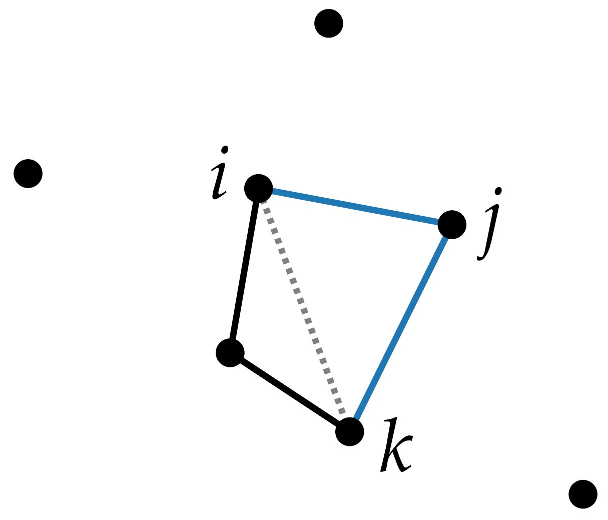
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
- Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
- Let k be vertex after i in T.
- Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

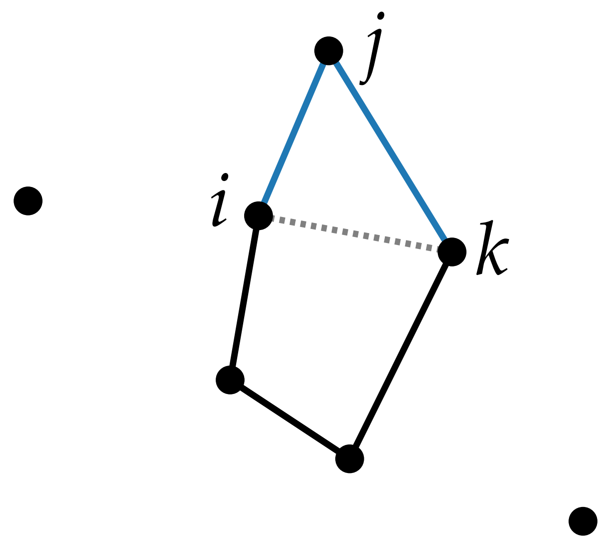
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

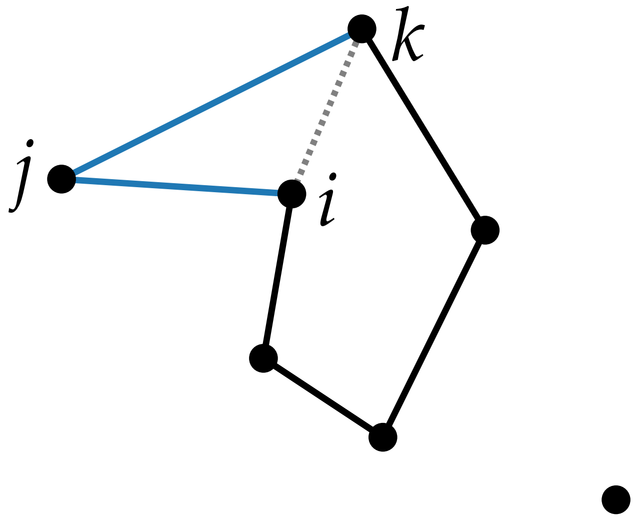
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

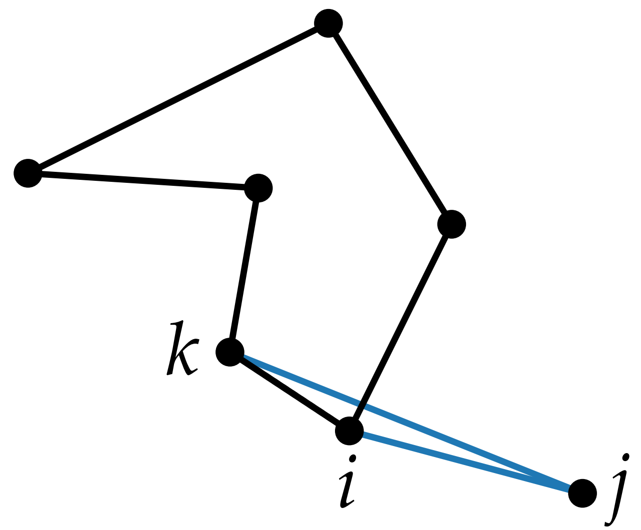
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
- Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
- Let k be vertex after i in T.
- Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

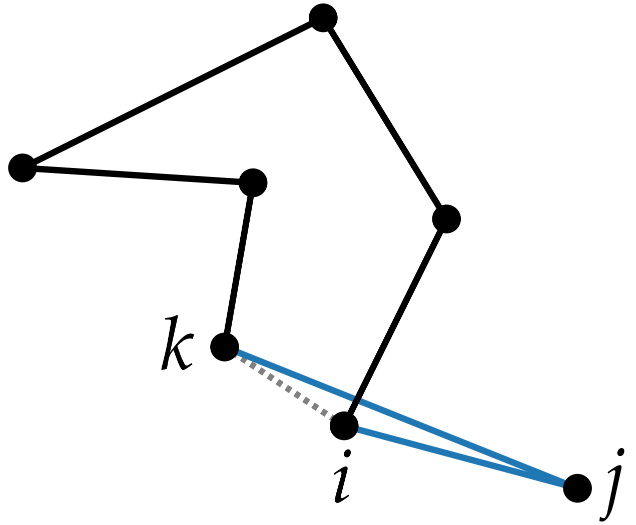
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
- Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
- Let k be vertex after i in T.
- Add j between i and k.



Nearest Addition Algorithm for Metric TSP

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

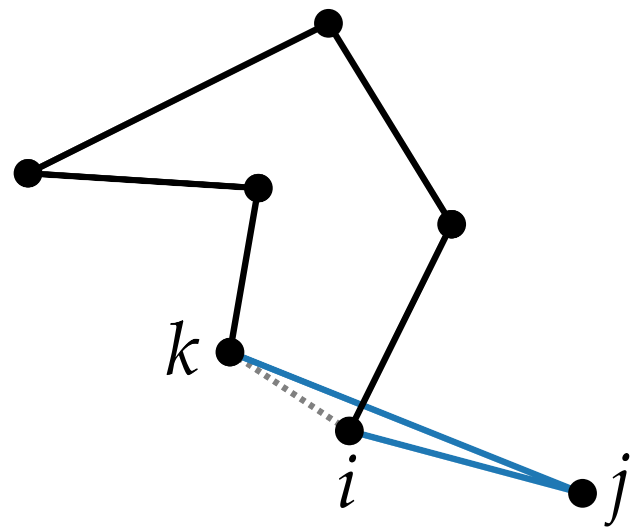
Find closest pair, say i and k.
Set tour T to go from i to k to i (clockwise).
while $T \subsetneq V$ **do**
  Find pair $(i, j) \in T \times (V \setminus T)$ minimizing $d(i, j)$.
  Let k be vertex after i in T.
  Add j between i and k.



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).
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.

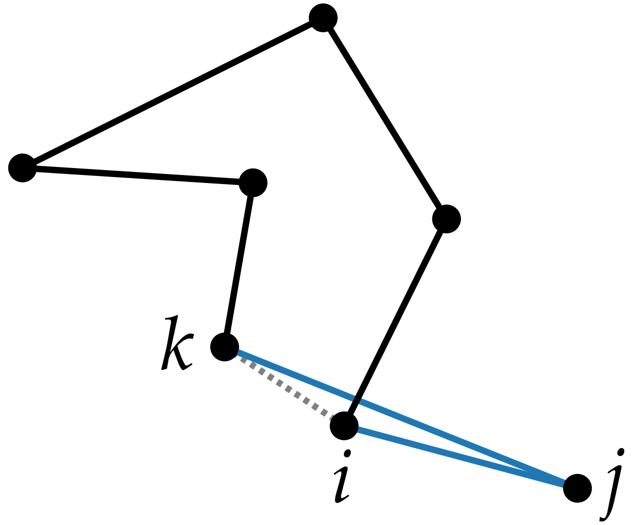


Theorem 6.
`NearestAdditionAlgorithm` is a 2-approximation algorithm for metric TSP.

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 $T \subsetneq V$ **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.



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

- $\mathrm{ALG}(I) \leq (1 + \varepsilon) \cdot \mathrm{OPT}(I)$, 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.
maximization
Let Π be a minimization problem. An algorithm $\mathcal{A}$ is called a **polynomial-time approximation scheme (PTAS)** if $\mathcal{A}$ computes, for every input (I, ε) (consisting of an instance I of Π 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 $\varepsilon > 0$.

Approximation Schemes

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

Definition.
maximization
Let Π be a minimization problem. An algorithm $\mathcal{A}$ is called a **polynomial-time approximation scheme (PTAS)** if $\mathcal{A}$ computes, for every input (I, ε) (consisting of an instance I of Π 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 $1/\varepsilon$.

Approximation Schemes

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

Definition. maximization

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

$\geq (1-\varepsilon)$

- ALG$(I) \leq (1+\varepsilon) \cdot$ 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.

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

Approximation Schemes

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

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

$\geq (1-\varepsilon)$

- $\mathrm{ALG}(I) \leq (1+\varepsilon) \cdot \mathrm{OPT}(I)$, 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.

- $\mathcal{O}\left(2^{\frac{n}{\varepsilon}}\right) \Rightarrow$ no PTAS
- $\mathcal{O}\left(n^2 + n^{\frac{1}{\varepsilon}}\right) \Rightarrow$
- $\mathcal{O}\left(n^2 \cdot 3^{\frac{1}{\varepsilon}}\right) \Rightarrow$
- $\mathcal{O}\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right) \Rightarrow$

Approximation Schemes

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

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

$\geq (1-\varepsilon)$

- $\mathrm{ALG}(I) \leq (1+\varepsilon) \cdot \mathrm{OPT}(I)$, 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.

- $\mathcal{O}\left(2^{\frac{n}{\varepsilon}}\right) \Rightarrow$ no PTAS
- $\mathcal{O}\left(n^2 + n^{\frac{1}{\varepsilon}}\right) \Rightarrow$ PTAS but no FPTAS
- $\mathcal{O}\left(n^2 \cdot 3^{\frac{1}{\varepsilon}}\right) \Rightarrow$
- $\mathcal{O}\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right) \Rightarrow$

Approximation Schemes

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

Definition. maximization

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

$\geq (1-\varepsilon)$

- $\mathsf{ALG}(I) \leq (1+\varepsilon) \cdot \mathsf{OPT}(I)$, 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.

- $\mathcal{O}\left(2^{\frac{n}{\varepsilon}}\right) \Rightarrow$ no PTAS
- $\mathcal{O}\left(n^2 + n^{\frac{1}{\varepsilon}}\right) \Rightarrow$ PTAS but no FPTAS
- $\mathcal{O}\left(n^2 \cdot 3^{\frac{1}{\varepsilon}}\right) \Rightarrow$ PTAS but no FPTAS
- $\mathcal{O}\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right) \Rightarrow$

Approximation Schemes

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

Definition.

maximization

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

$\geq (1-\varepsilon)$

- ALG$(I) \leq (1+\varepsilon) \cdot$ 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 $1/\varepsilon$.

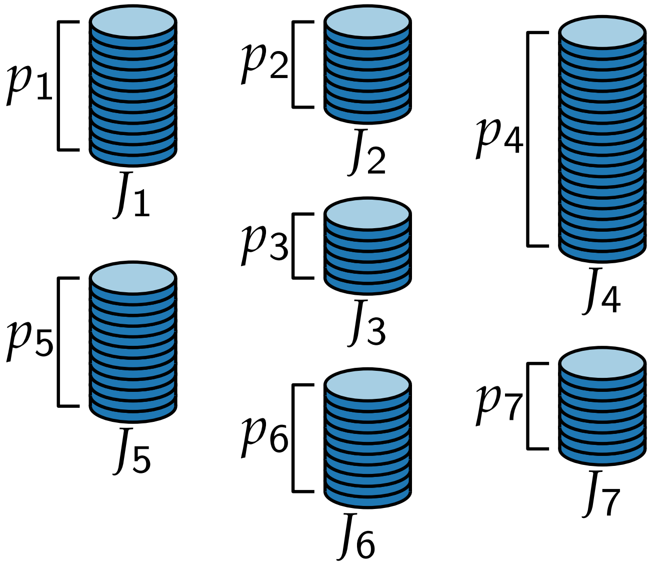
Examples.

- $\mathcal{O}\left(2^{\frac{n}{\varepsilon}}\right) \Rightarrow$ no PTAS
- $\mathcal{O}\left(n^2 + n^{\frac{1}{\varepsilon}}\right) \Rightarrow$ PTAS but no FPTAS
- $\mathcal{O}\left(n^2 \cdot 3^{\frac{1}{\varepsilon}}\right) \Rightarrow$ PTAS but no FPTAS
- $\mathcal{O}\left(n^4 \cdot \left(\frac{1}{\varepsilon}\right)^2\right) \Rightarrow$ FPTAS

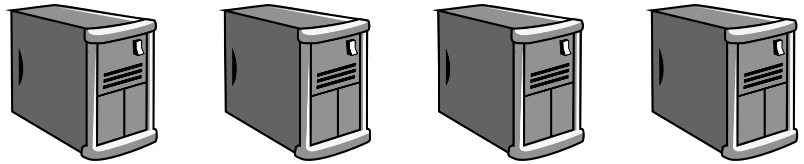
Multiprocessor Scheduling

Input.

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



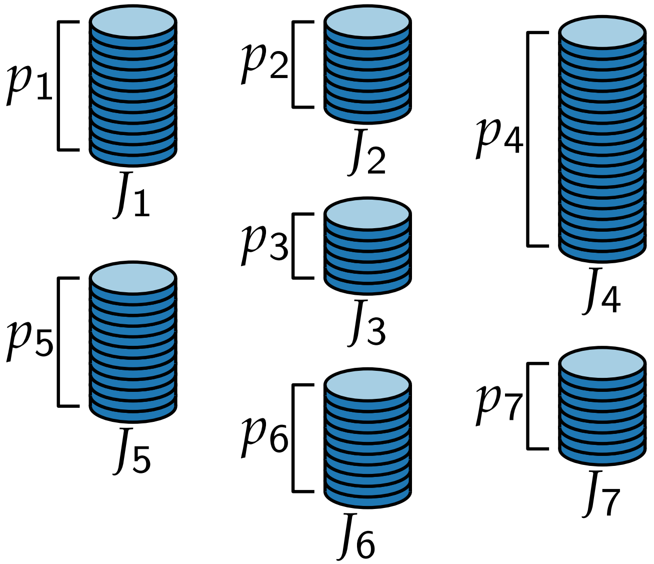
- m identical machines ($m < n$)



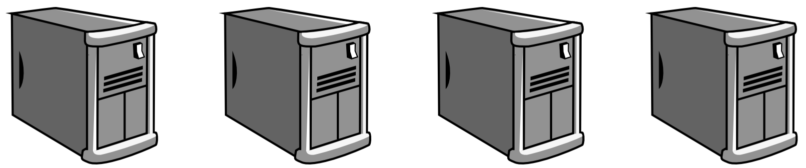
Multiprocessor Scheduling

Input.

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



- m identical machines $(m < n)$

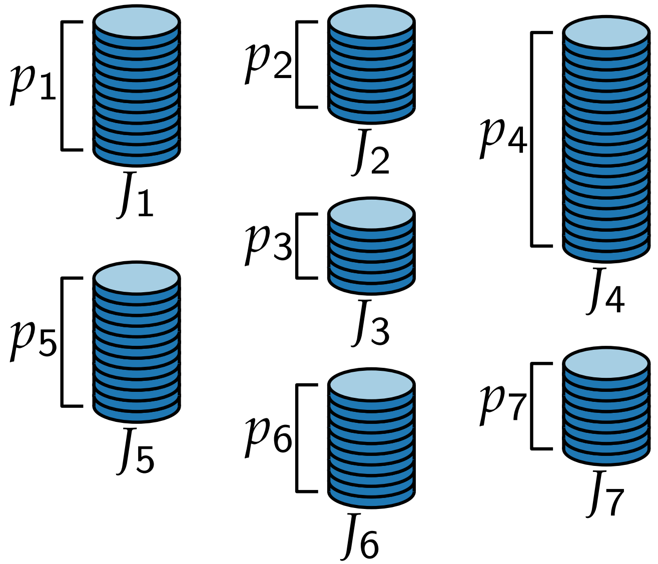


Output. Assignment of jobs to machines such that the time when all jobs have been processed is minimum.
This is called the **makespan** of the assignment.

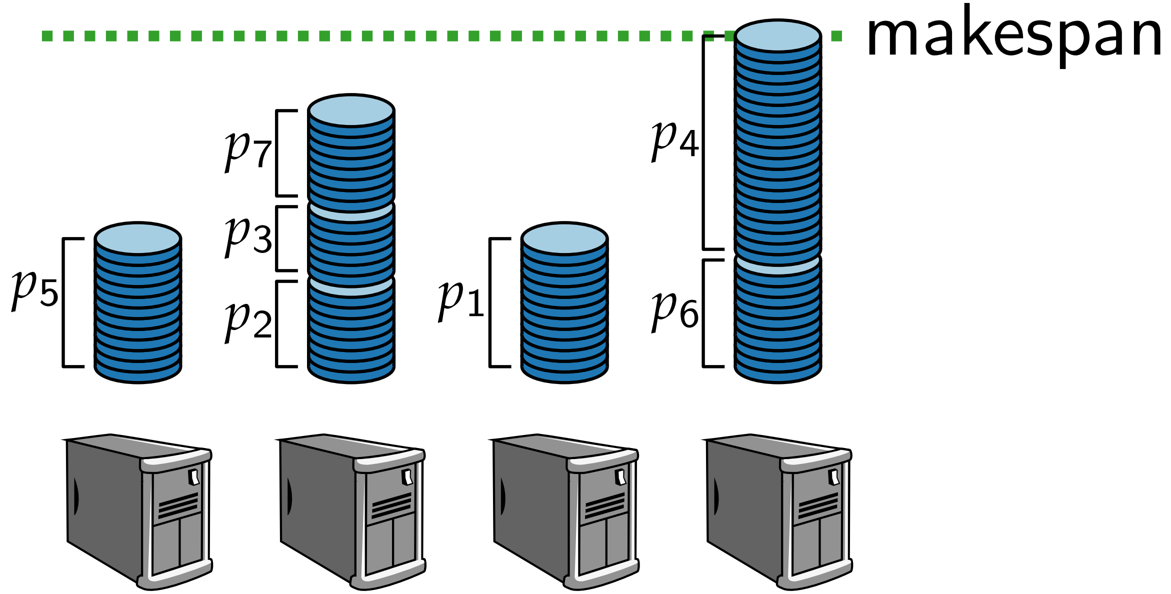
Multiprocessor Scheduling

Input.

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



- m identical machines ($m < n$)

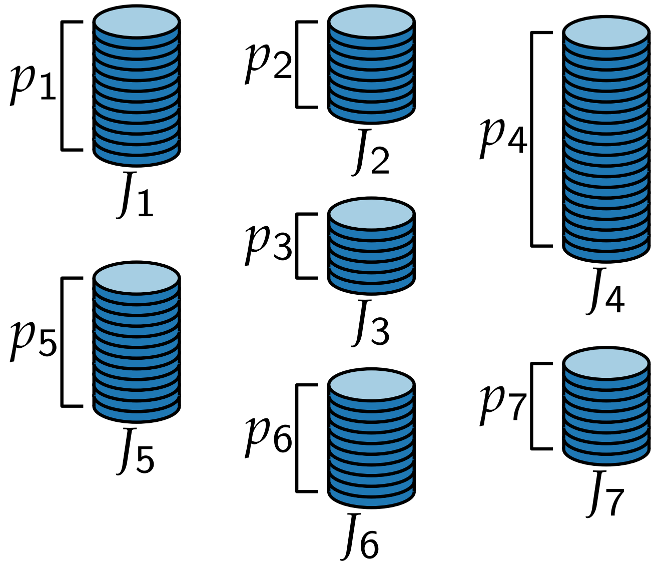


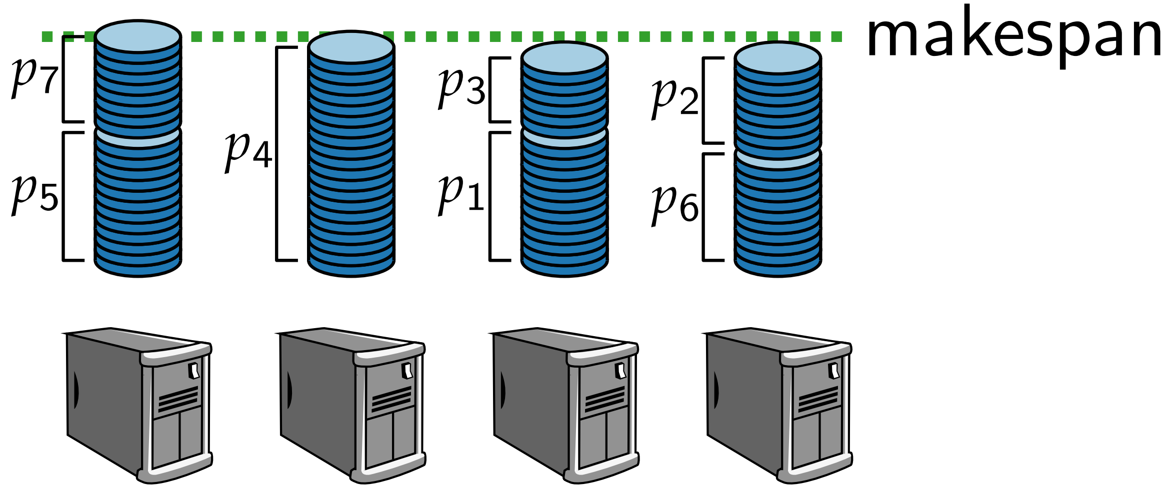
Output. Assignment of jobs to machines such that the time when all jobs have been processed is minimum.
This is called the **makespan** of the assignment.

Multiprocessor Scheduling

Input.

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



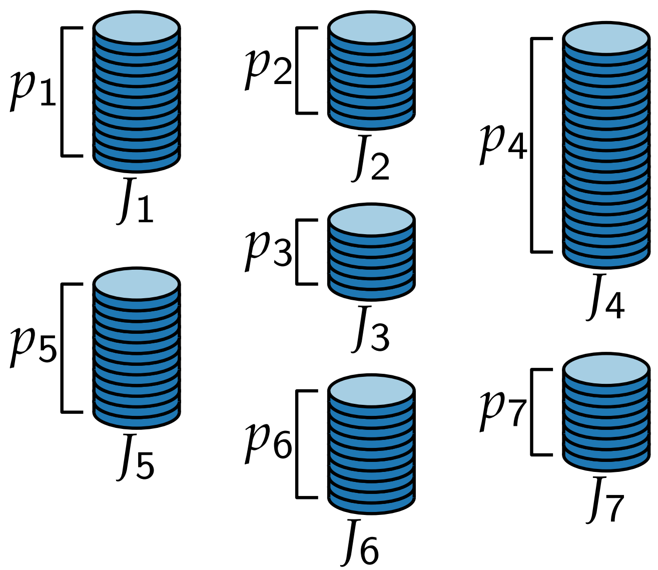


Output. Assignment of jobs to machines such that the time when all jobs have been processed is minimum.
This is called the **makespan** of the assignment.

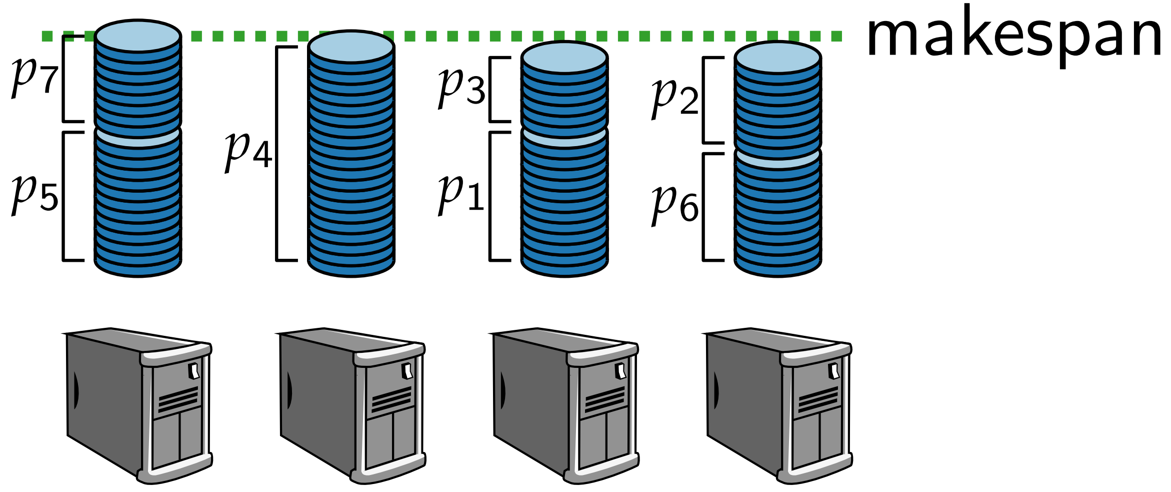
Multiprocessor Scheduling

Input.

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



- 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.

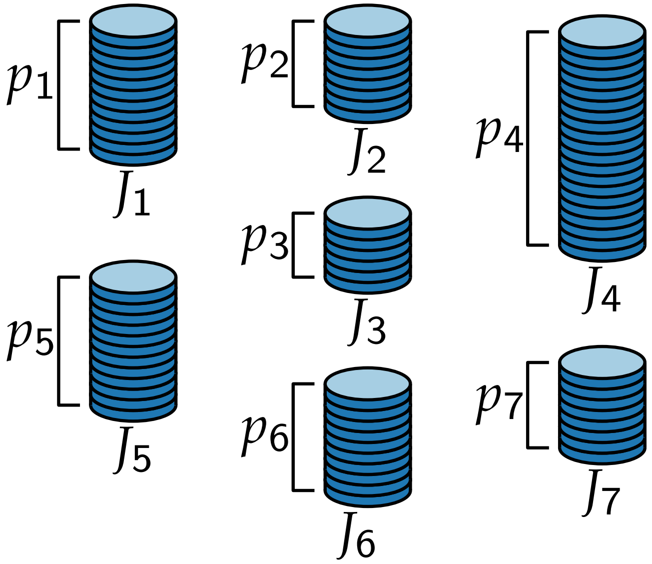
- Multiprocessor scheduling is NP-hard.

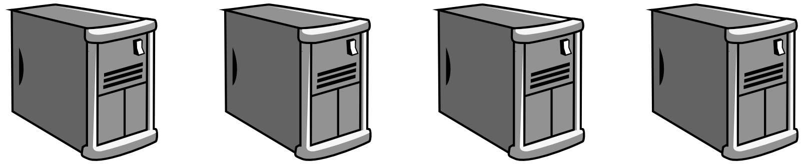
Multiprocessor Scheduling – List Scheduling

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

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

Example.





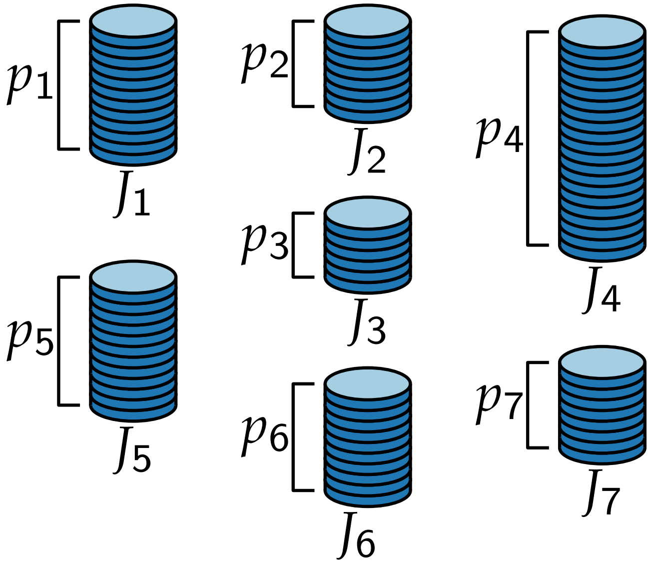
Multiprocessor Scheduling – List Scheduling

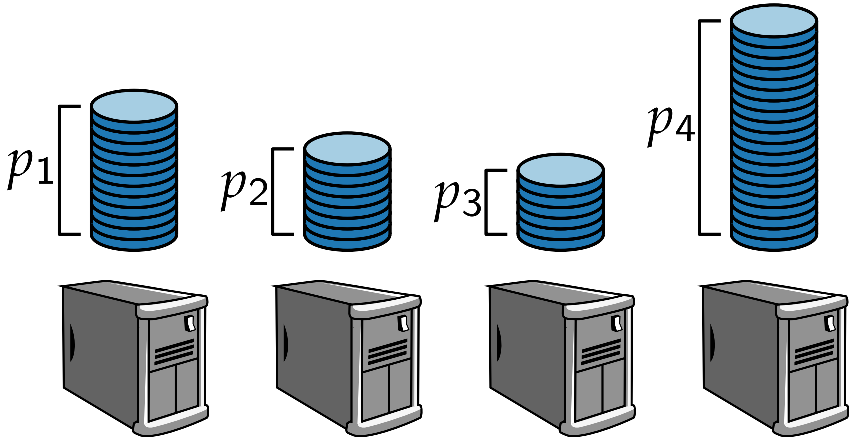
ListScheduling($J_1, \ldots, J_n, m$)

Put the first m jobs on the m machines.

Put the next job on the first free machine.

Example.





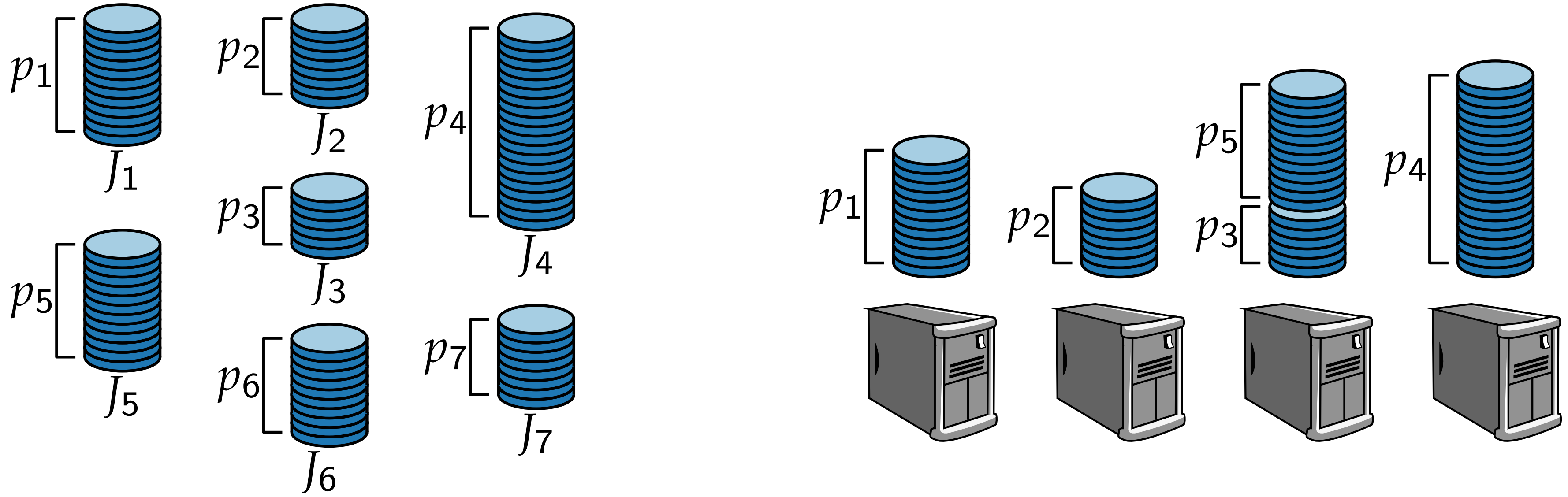
Multiprocessor Scheduling – List Scheduling

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.



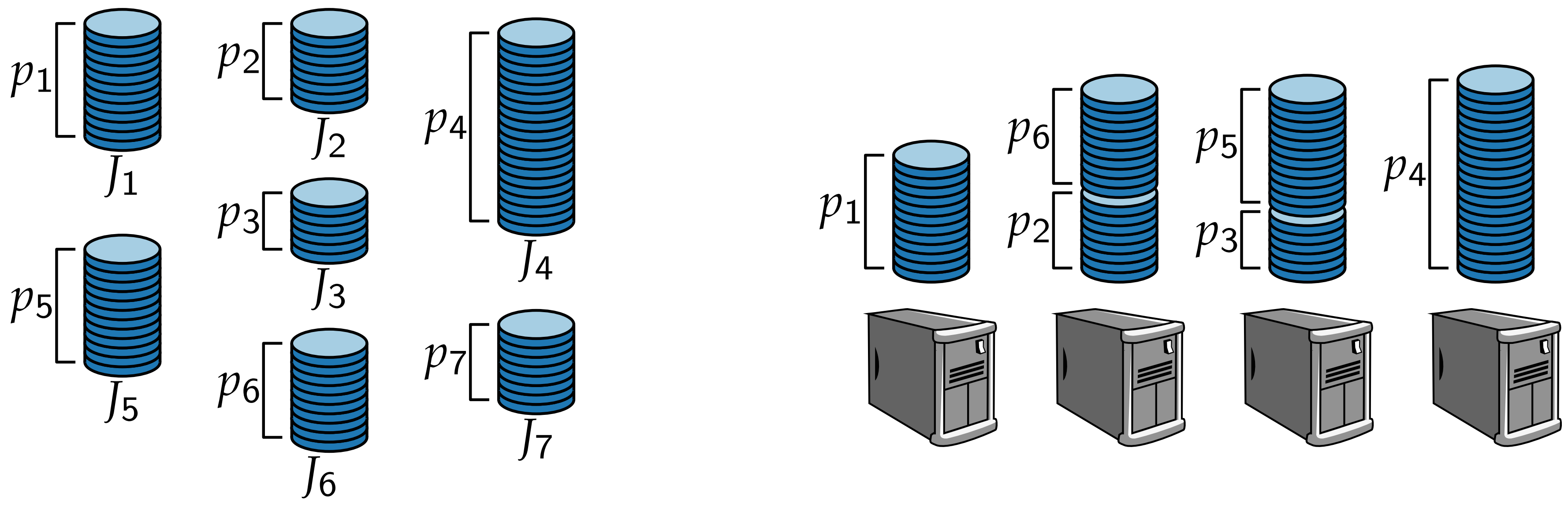
Multiprocessor Scheduling – List Scheduling

$\text{ListScheduling}(J_1, \dots, J_n, m)$

Put the first m jobs on the m machines.

Put the next job on the first free machine.

Example.



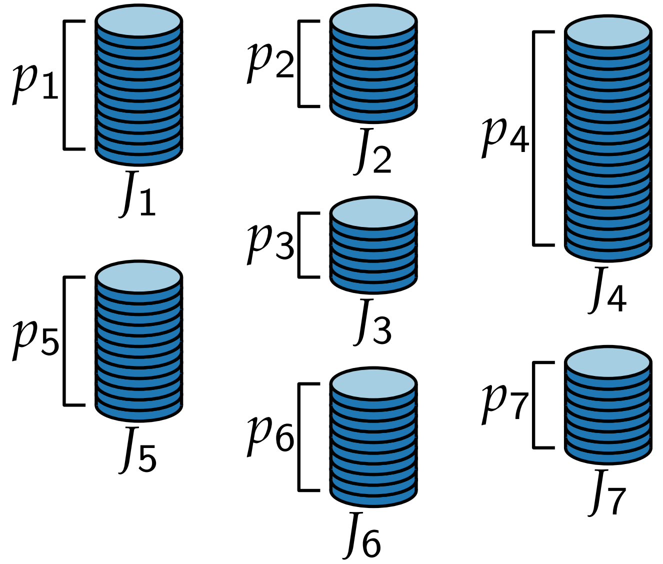
Multiprocessor Scheduling – List Scheduling

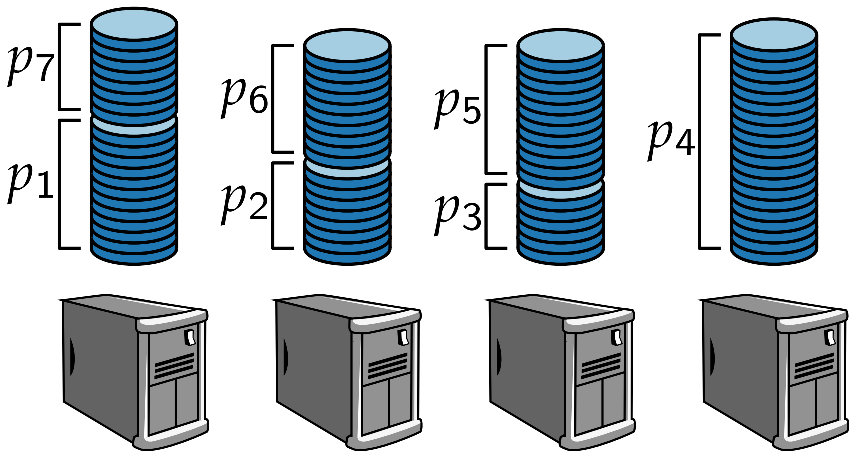
ListScheduling($J_1, \ldots, J_n, m$)

Put the first m jobs on the m machines.

Put the next job on the first free machine.

Example.



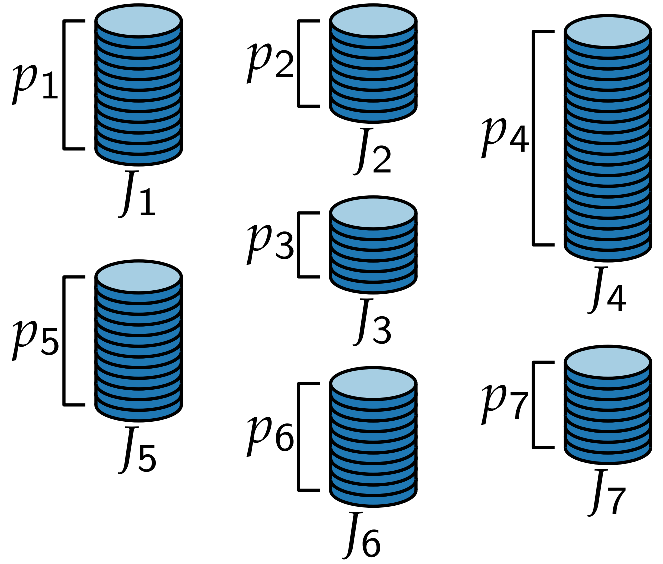


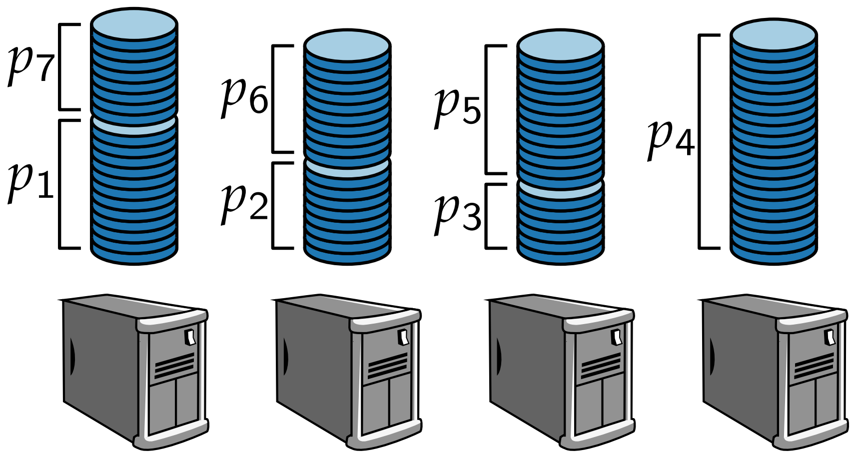
Multiprocessor Scheduling – List Scheduling

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

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

Example.





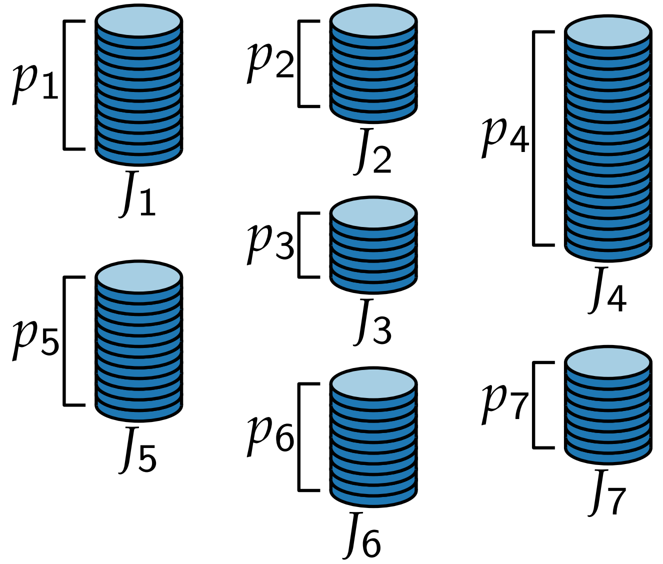
- ListScheduling runs in time.

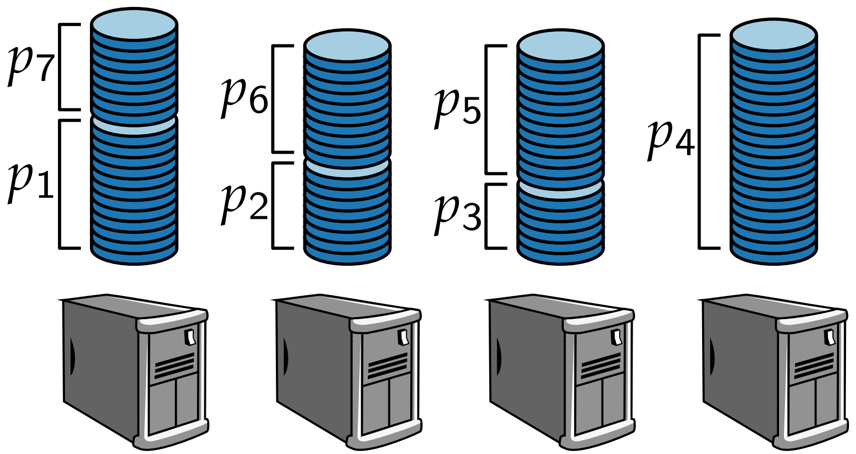
Multiprocessor Scheduling – List Scheduling

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.





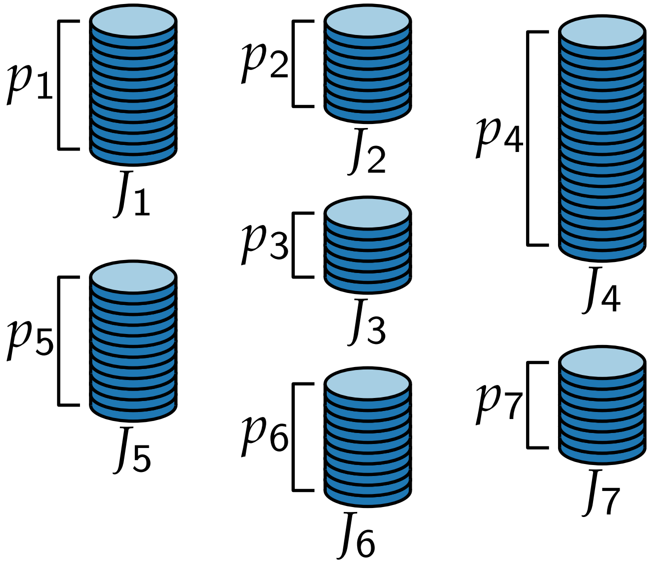
- ListScheduling runs in $\mathcal{O}(n \log m)$ time.

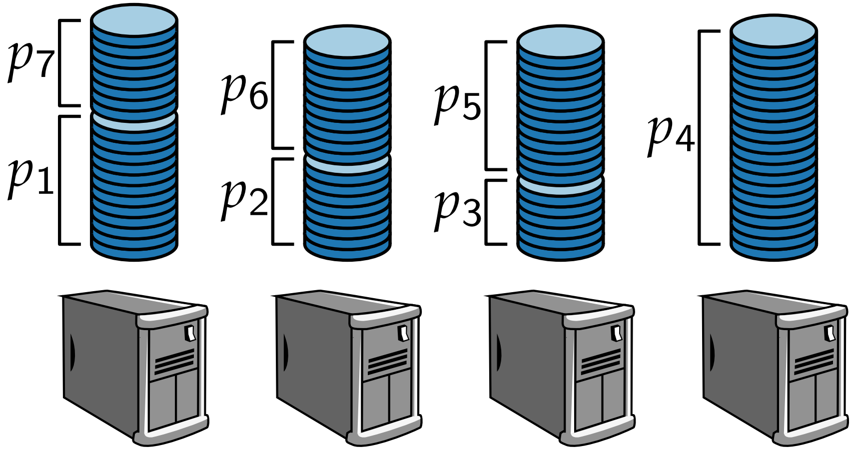
Multiprocessor Scheduling – List Scheduling

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 $\mathcal{O}(n \log m)$ time.

  Iterate over n jobs while maintaining a priority queue for the machines where each machine has its current completion time as its priority.

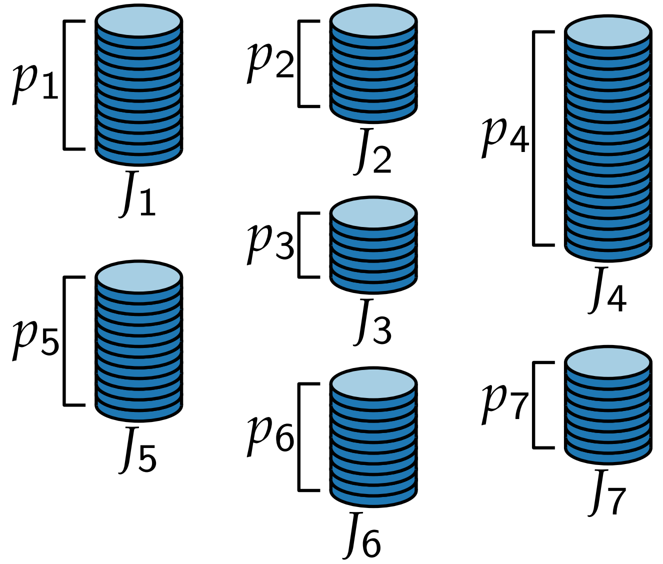
Multiprocessor Scheduling – List Scheduling

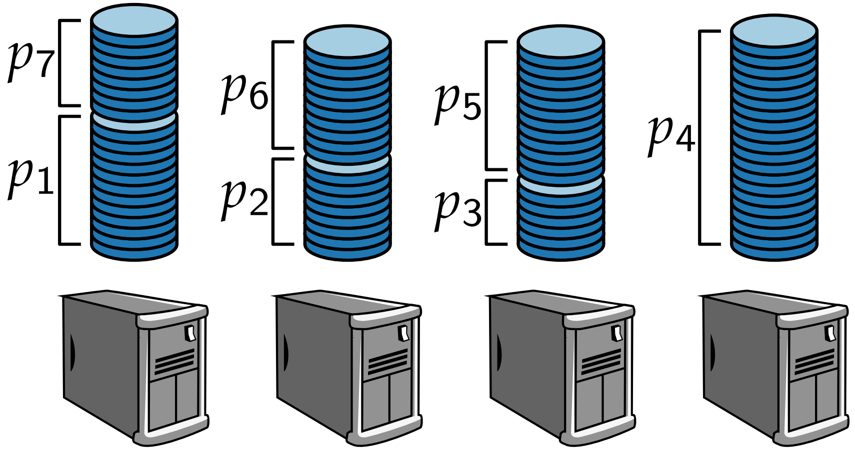
ListScheduling$(J_1, \ldots, J_n, m)$

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

Theorem 7.
ListScheduling is a factor-approximation algorithm.

Example.





- ListScheduling runs in $\mathcal{O}(n \log m)$ time.

  Iterate over n jobs while maintaining a priority queue for the machines where each machine has its current completion time as its priority.

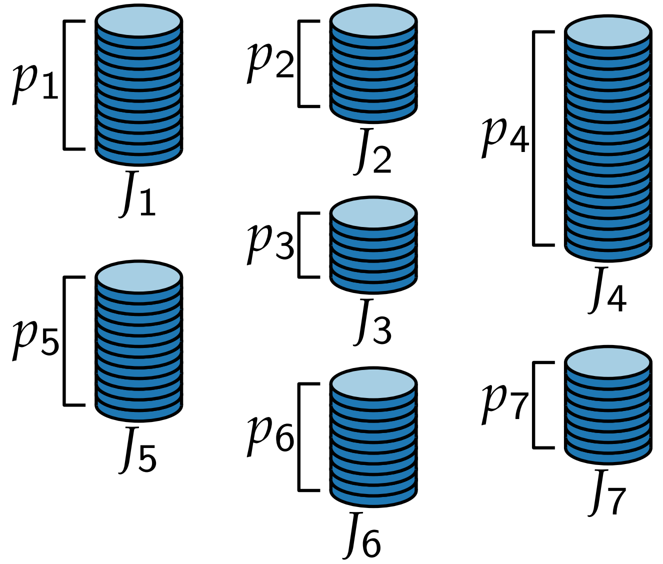
Multiprocessor Scheduling – List Scheduling

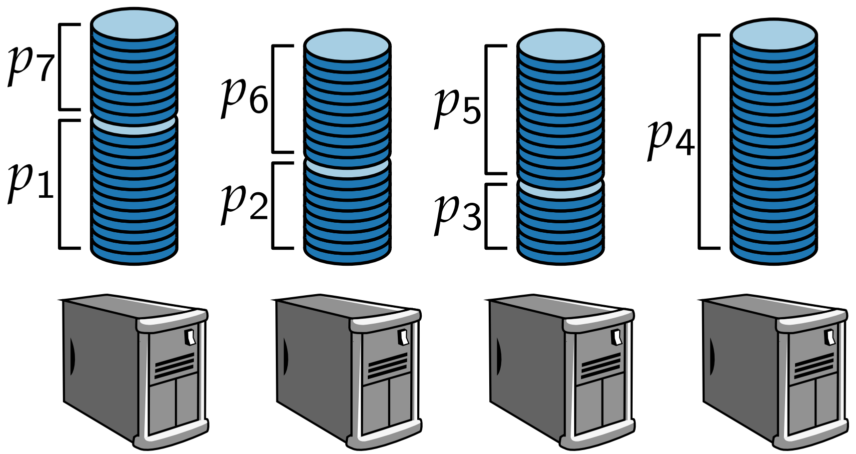
ListScheduling$(J_1, \ldots, J_n, m)$

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

Theorem 7.
ListScheduling is a factor-$\left(2 - \frac{1}{m}\right)$ approximation algorithm.

Example.





- ListScheduling runs in $\mathcal{O}(n \log m)$ time.

  Iterate over n jobs while maintaining a priority queue for the machines where each machine has its current completion time as its priority.

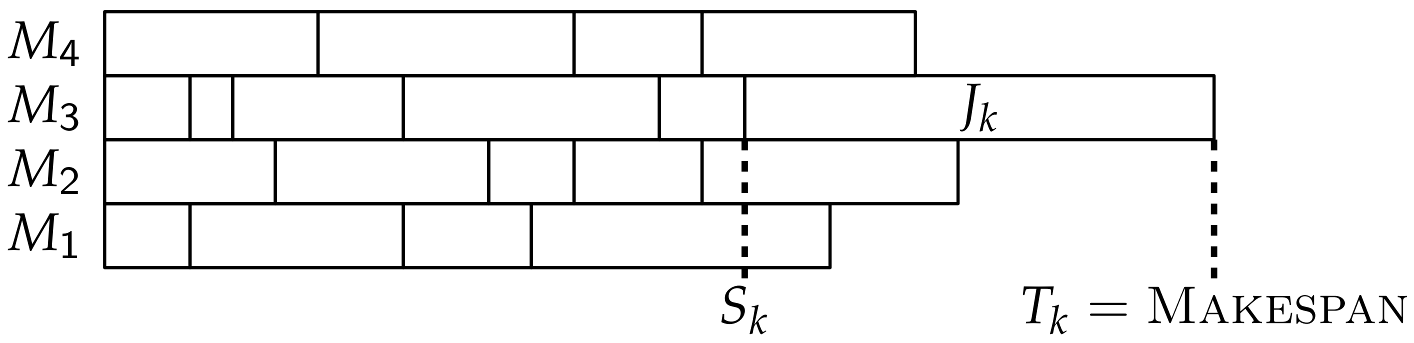
Multiprocessor Scheduling – List Scheduling (Proof)

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

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.



Multiprocessor Scheduling – List Scheduling (Proof)

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

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.

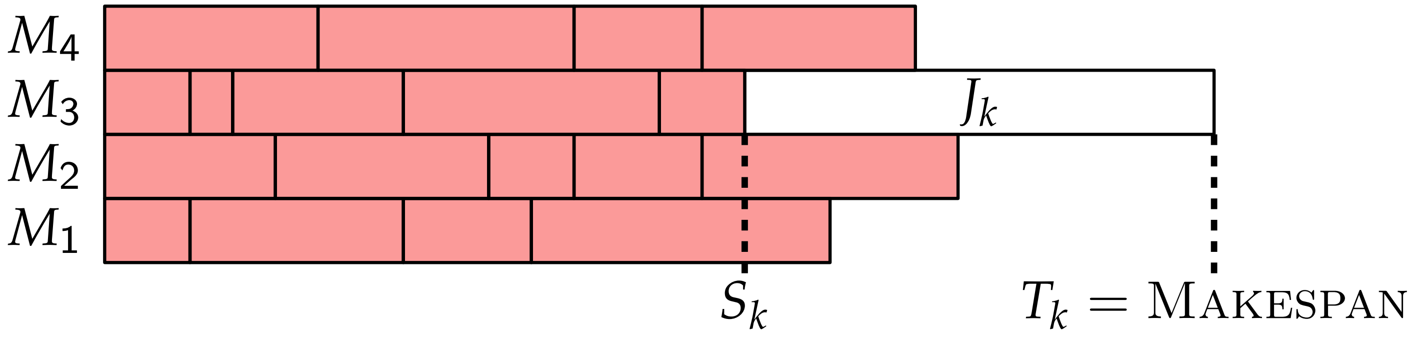
Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, T_k determines the makespan.

- No machine idles at time S_k.

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

weight of all jobs but J_k

evenly distributed on m machines



Multiprocessor Scheduling – List Scheduling (Proof)

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

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.

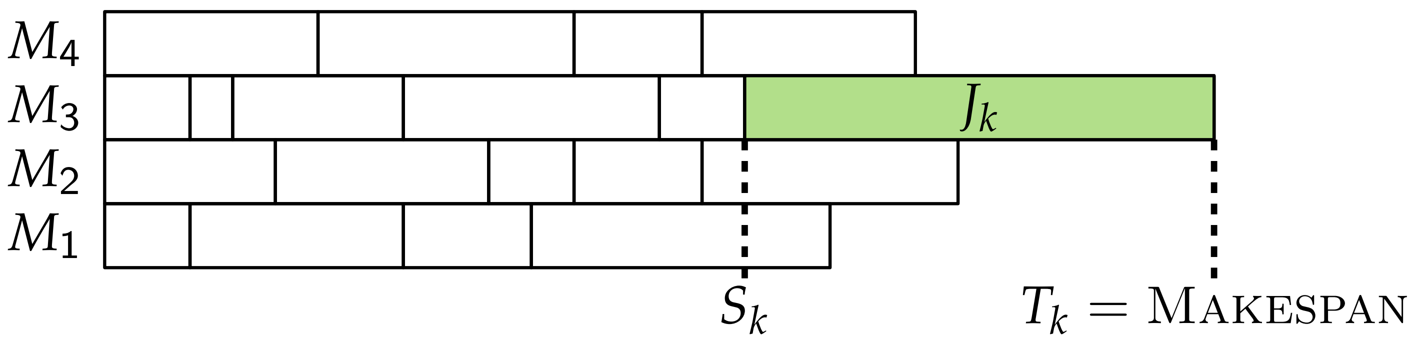
- No machine idles at time S_k.

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

weight of all jobs but J_k

evenly distributed on m machines

- For the optimal makespan T_{OPT}, we have:
- $T_{\mathrm{OPT}} \geq p_k$



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.

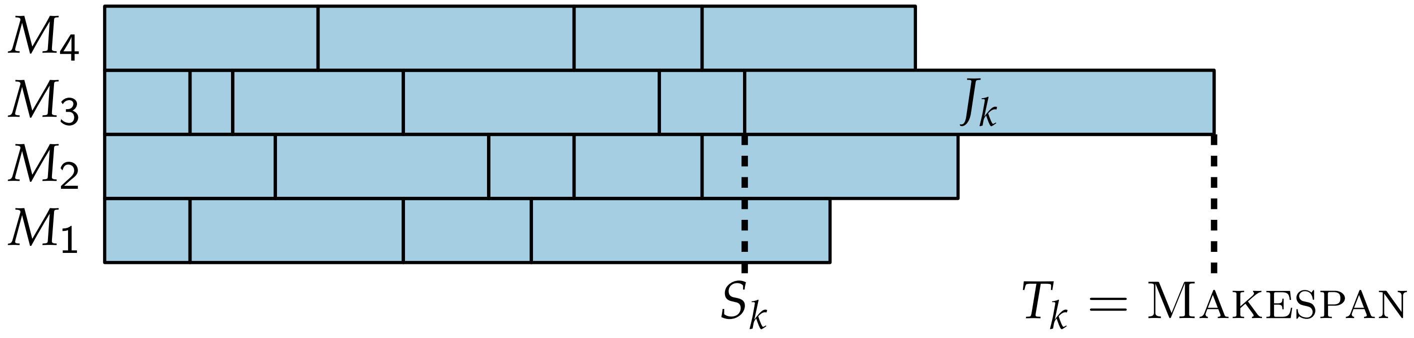
- No machine idles at time S_k.

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

weight of all jobs but J_k evenly distributed on m machines

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

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



Multiprocessor Scheduling – List Scheduling (Proof)

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

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.

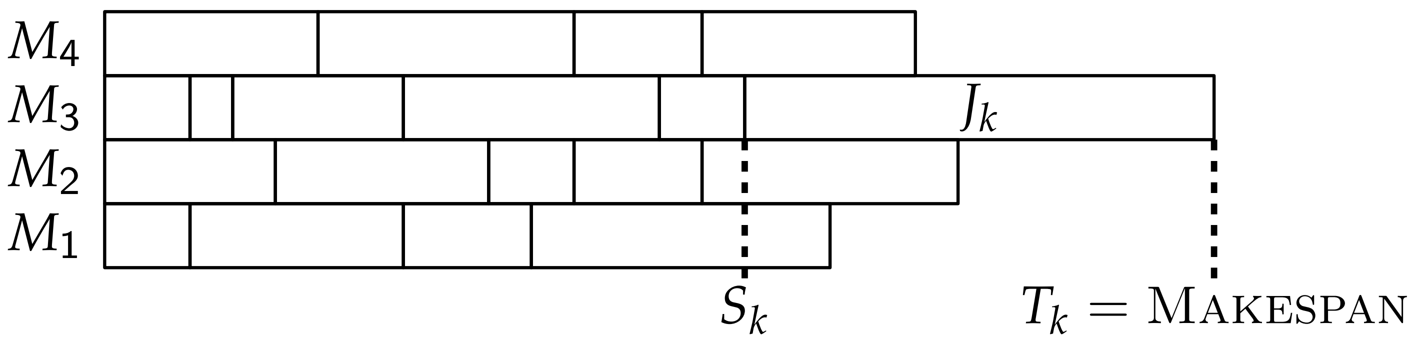
$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$$

weight of all jobs but J_k evenly distributed on m machines

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

- Hence:

$$T_k = S_k + p_k$$



Multiprocessor Scheduling – List Scheduling (Proof)

$\textsc{ListScheduling}(J_1, \dots, J_n, m)$

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

Theorem 7.
$\textsc{ListScheduling}$ 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.

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

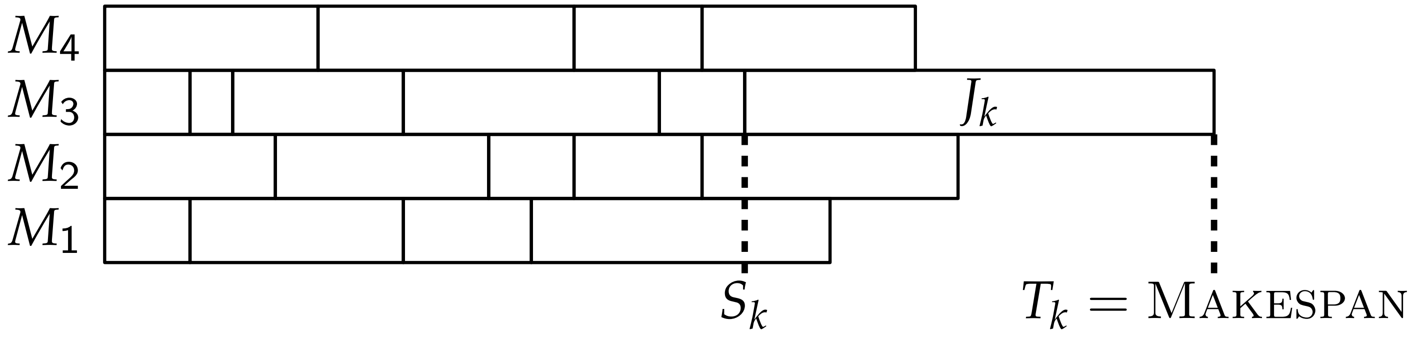
weight of all jobs but J_k evenly distributed on m machines

- Hence:

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

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

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



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.

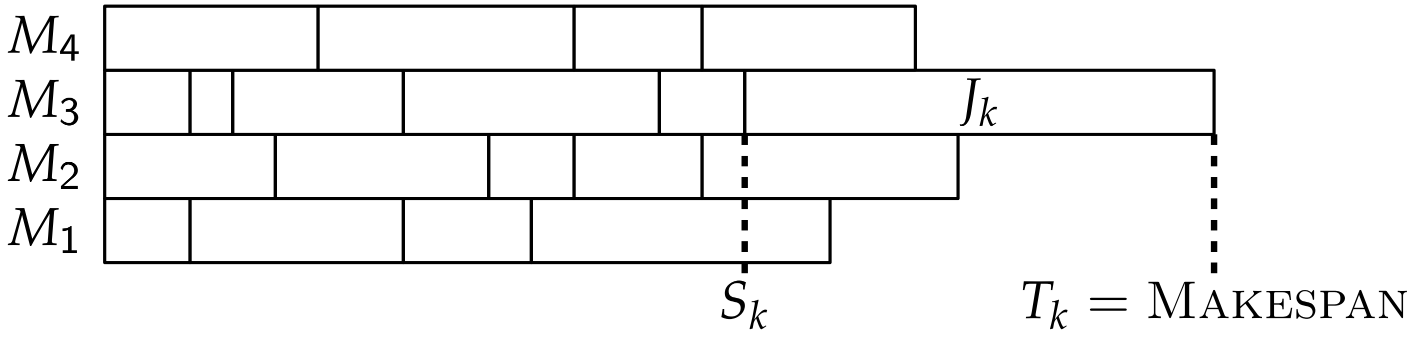
$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$$

weight of all jobs but J_k evenly distributed on m machines

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

- Hence:

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



□

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.

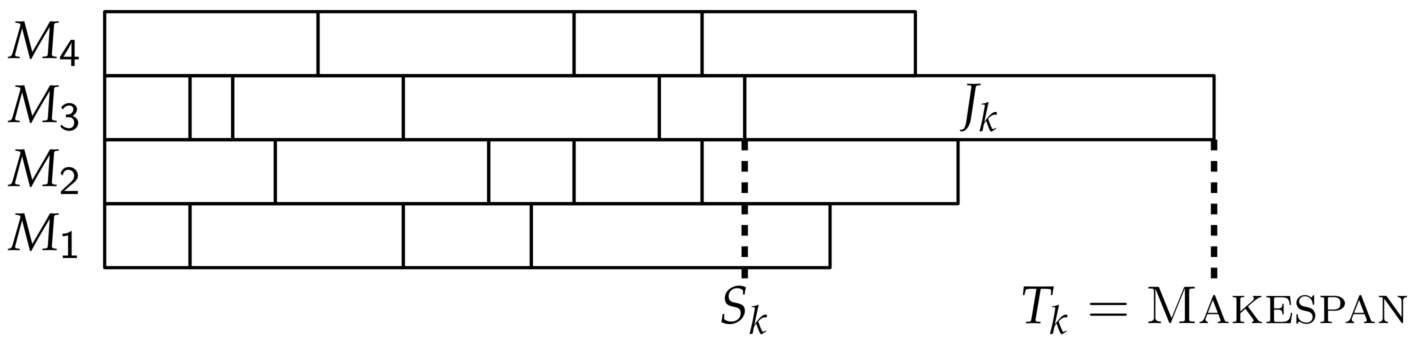
$$S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$$

weight of all jobs but J_k evenly distributed on m machines

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

- Hence:

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



Multiprocessor Scheduling – List Scheduling (Proof)

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

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

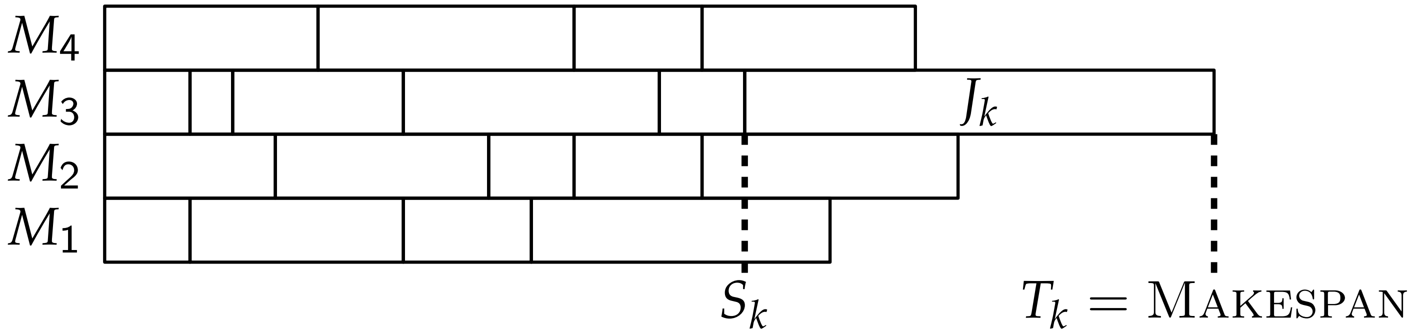
Theorem 7.
ListScheduling is a $(2 - \frac{1}{m})$-approximation alg.

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.

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

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



- Hence:

$$
\begin{aligned}
T_k &= S_k + p_k \\
&\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
&= \frac{1}{m} \cdot \sum_{i=1}^{n} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
&\leq T_{\mathrm{OPT}} + \left(1 - \frac{1}{m}\right) \cdot T_{\mathrm{OPT}} \\
&= \left(2 - \frac{1}{m}\right) \cdot T_{\mathrm{OPT}}
\end{aligned}
$$

□

Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.

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

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

Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$

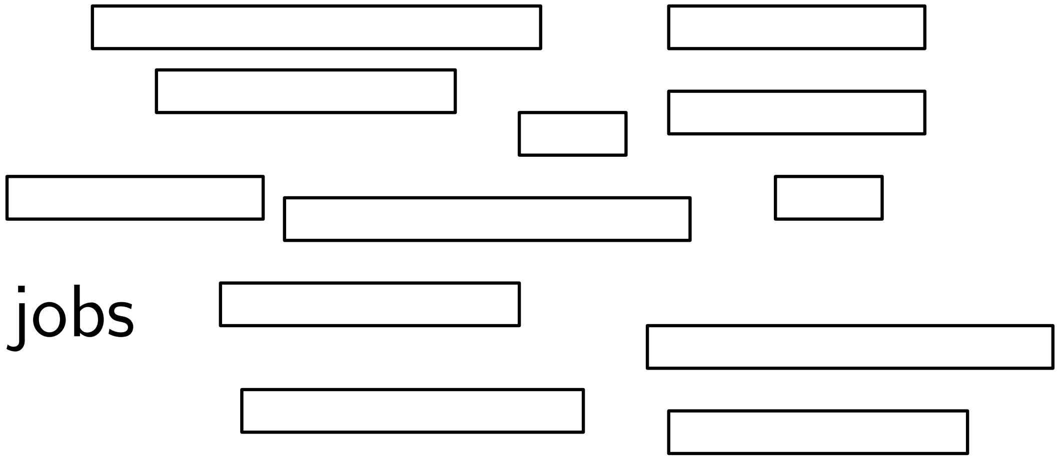
Sort jobs in descending order of runtime.

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

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

Example.

$\ell = 6$



Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$

Sort jobs in descending order of runtime.

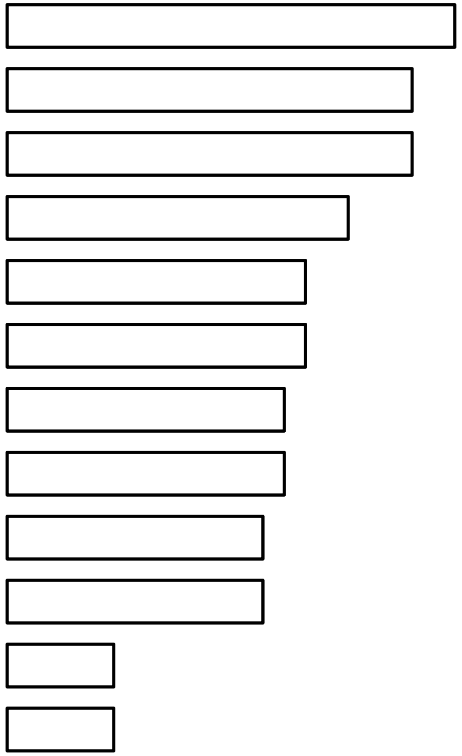
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally.

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

Example.

$\ell = 6$

sorted jobs



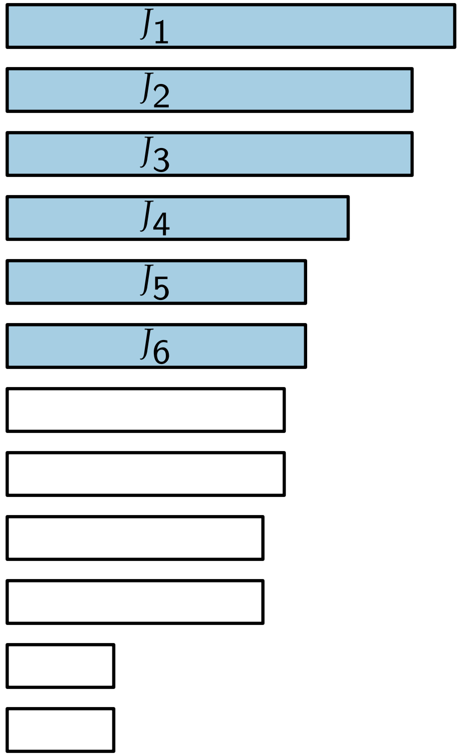
Multiprocessor Scheduling – PTAS

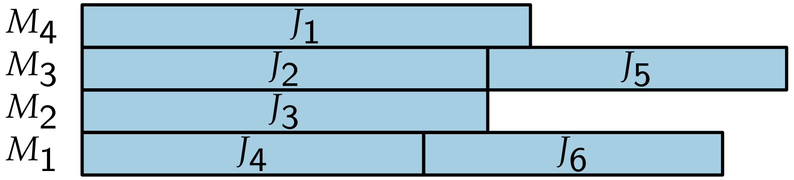
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
  Sort jobs in descending order of runtime.
  Schedule the ℓ 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

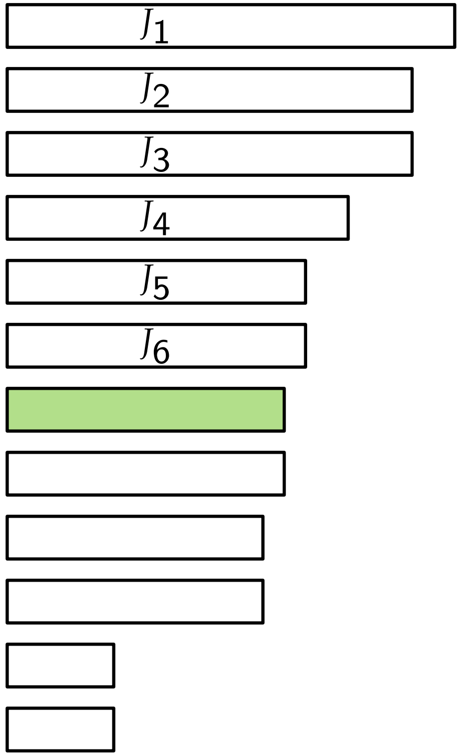
For a constant ℓ ($1 \leq \ell \leq n$) define the algorithm $\mathcal{A}_\ell$ as follows.

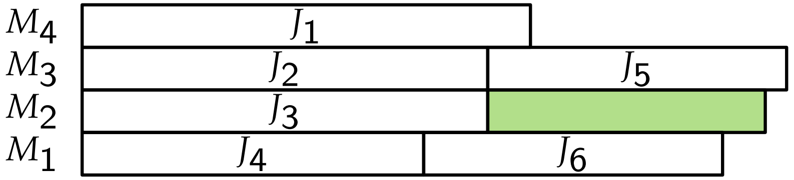
$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
- Sort jobs in descending order of runtime.
- Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally.
- Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs





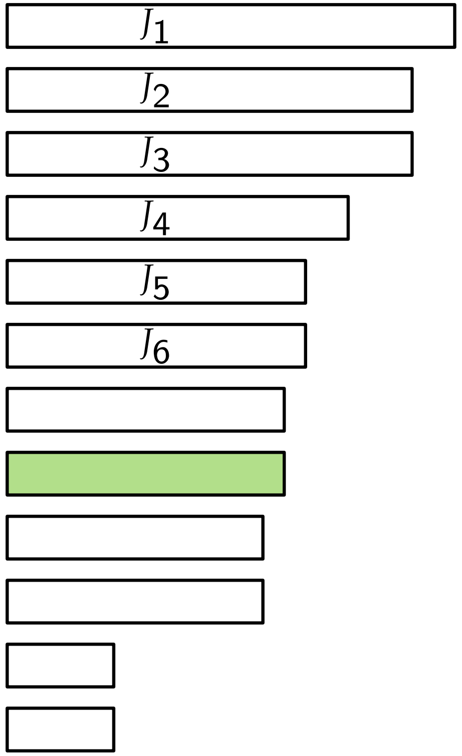
Multiprocessor Scheduling – PTAS

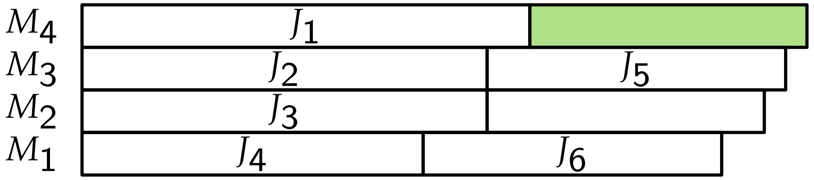
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ 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





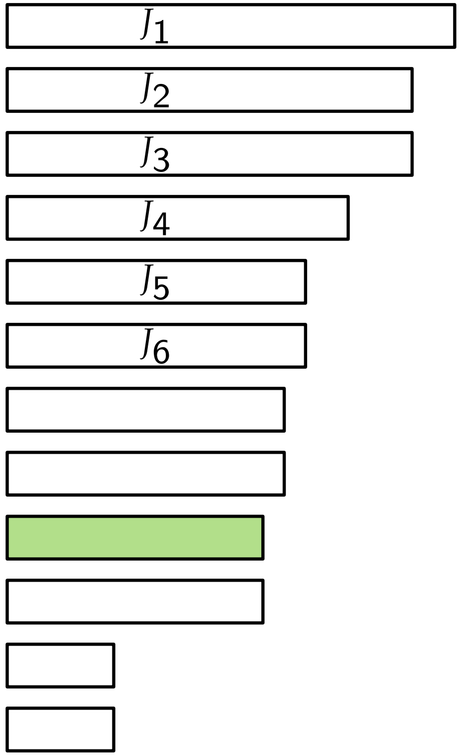
Multiprocessor Scheduling – PTAS

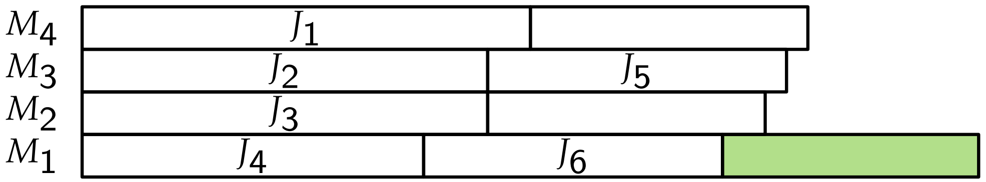
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
  Sort jobs in descending order of runtime.
  Schedule the ℓ 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





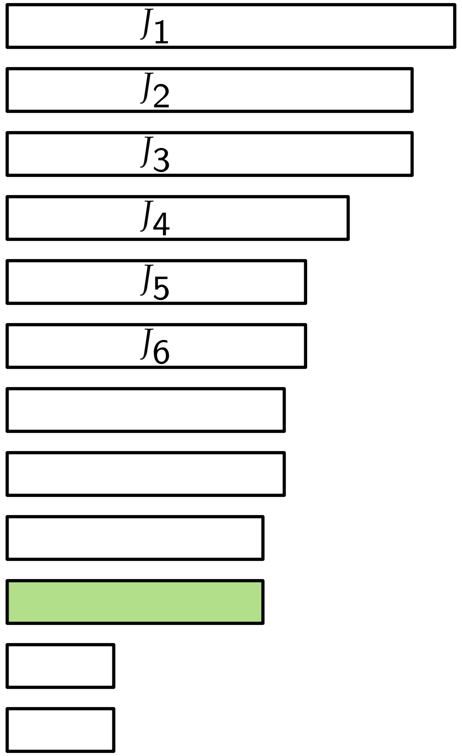
Multiprocessor Scheduling – PTAS

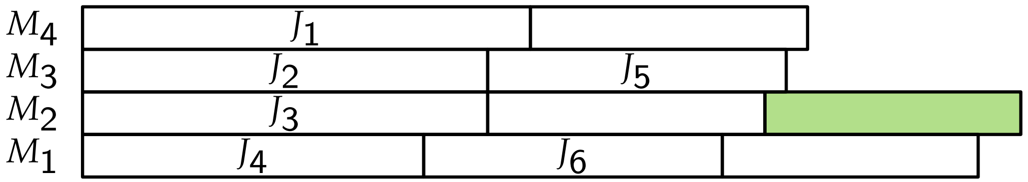
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ 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

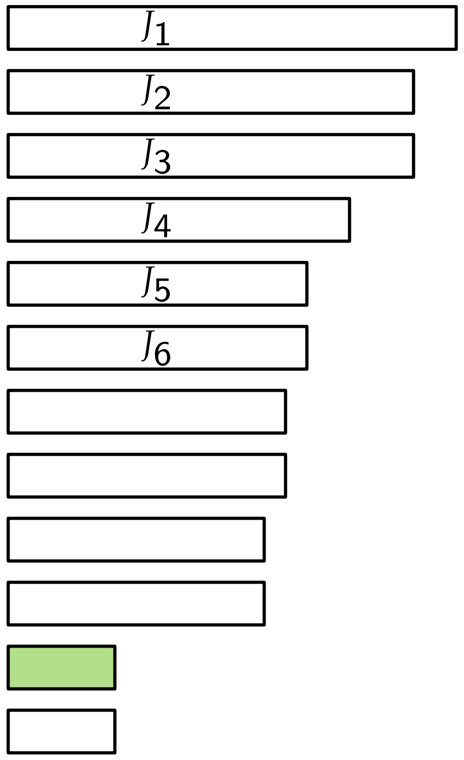
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

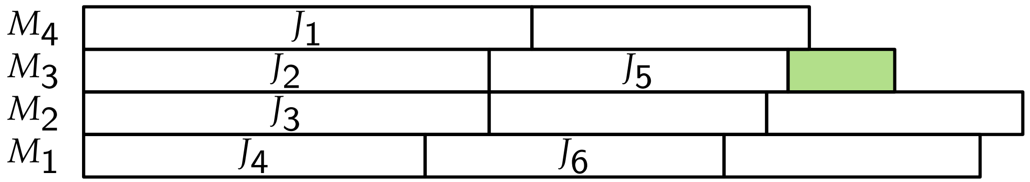
$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$.

Example.

$\ell = 6$

sorted jobs





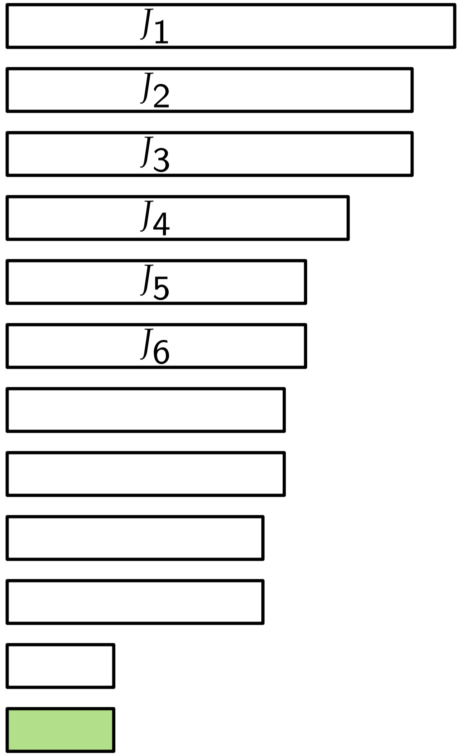
Multiprocessor Scheduling – PTAS

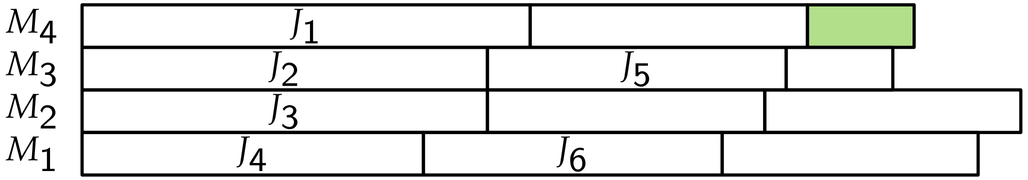
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ 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





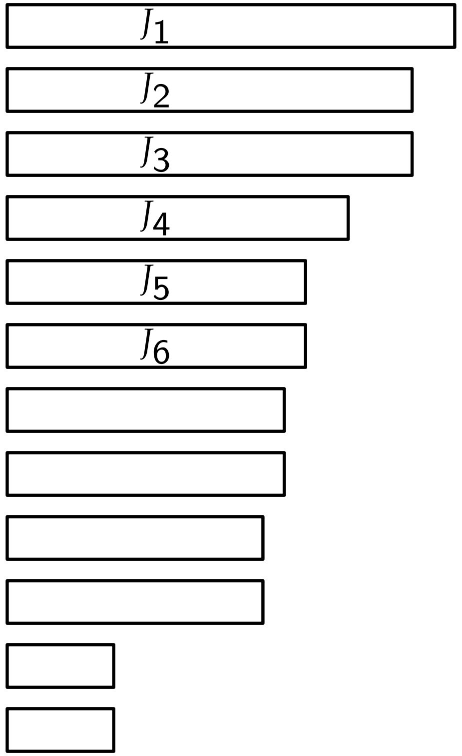
Multiprocessor Scheduling – PTAS

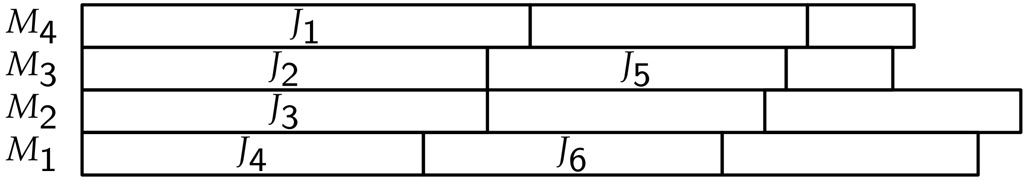
For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
  Sort jobs in descending order of runtime.
  Schedule the ℓ 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 ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

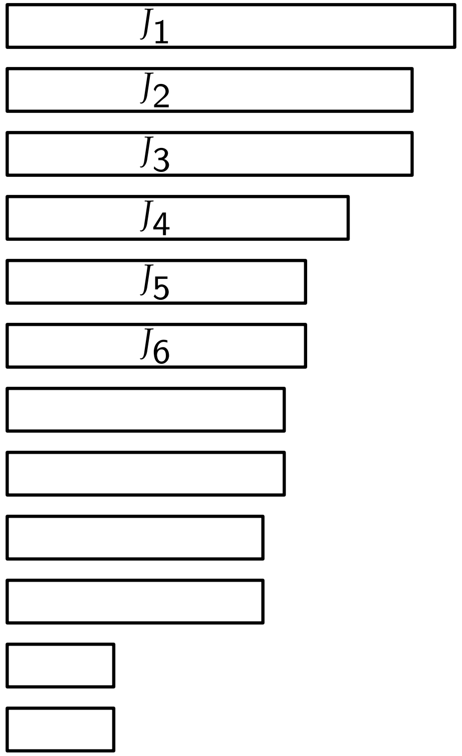
$\mathcal{A}_\ell(J_1, \ldots, J_n,\ m)$
- Sort jobs in descending order of runtime. $\mathcal{O}(n \log n)$
- Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally. $\mathcal{O}(m^\ell)$
- Use ListScheduling for the remaining jobs $J_{\ell+1}, \ldots, J_n$. $\mathcal{O}(n \log m)$

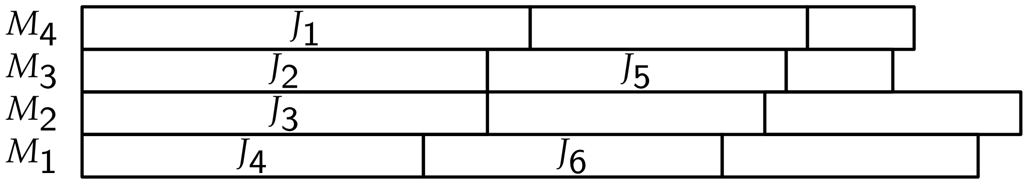
- Polynomial time for constant ℓ: $\mathcal{O}(m^\ell + n \log n)$

Example.

$\ell = 6$

sorted jobs





Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \ldots, J_n,\ m)$

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

- Polynomial time for constant ℓ: $\mathcal{O}(m^\ell + n \log n)$

Theorem 8.
For constant $\ell \in \{1, \ldots, n\}$, algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

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

- Polynomial time for constant ℓ: $\mathcal{O}(m^\ell + n \log n)$

Theorem 8.
For constant $\ell \in \{1, \ldots, n\}$, algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

- For $\varepsilon > 0$, choose ℓ such that $\mathcal{A}_\varepsilon = \mathcal{A}_{\ell(\varepsilon)}$ is a $(1 + \varepsilon)$-approximation algorithm.

Corollary 9.
For a constant number of machines, $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ is a PTAS.

Multiprocessor Scheduling – PTAS

For a constant ℓ $(1 \leq \ell \leq n)$ define the algorithm $\mathcal{A}_\ell$ as follows.

$\mathcal{A}_\ell(J_1, \dots, J_n, m)$
- Sort jobs in descending order of runtime. — $\mathcal{O}(n \log n)$
- Schedule the ℓ longest jobs $J_1, \dots, J_\ell$ optimally. — $\mathcal{O}(m^\ell)$
- Use ListScheduling for the remaining jobs $J_{\ell+1}, \dots, J_n$. — $\mathcal{O}(n \log m)$

- Polynomial time for constant ℓ: $\mathcal{O}(m^\ell + n \log n)$

Theorem 8.
For constant $\ell \in \{1, \dots, n\}$, algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

- For $\varepsilon > 0$, choose ℓ such that $\mathcal{A}_\varepsilon = \mathcal{A}_{\ell(\varepsilon)}$ is a $(1+\varepsilon)$-approximation algorithm.
- $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ is not an FPTAS since the running time is not polynomial in $\frac{1}{\varepsilon}$.

Corollary 9.
For a constant number of machines, $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ is a PTAS.

Multiprocessor Scheduling – PTAS (Proof)

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

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ 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.

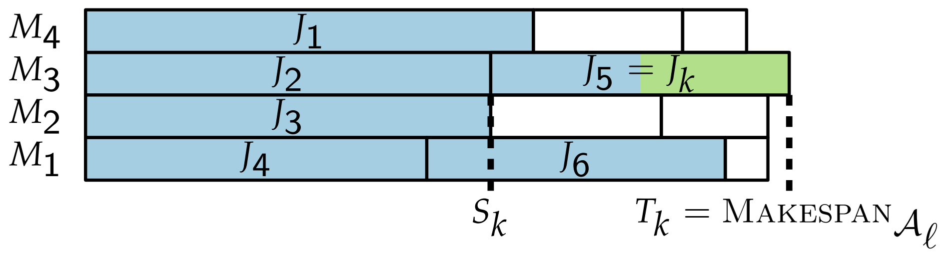
Multiprocessor Scheduling – PTAS (Proof)

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

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

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 ℓ jobs $J_1, \dots, J_\ell$.



Multiprocessor Scheduling – PTAS (Proof)

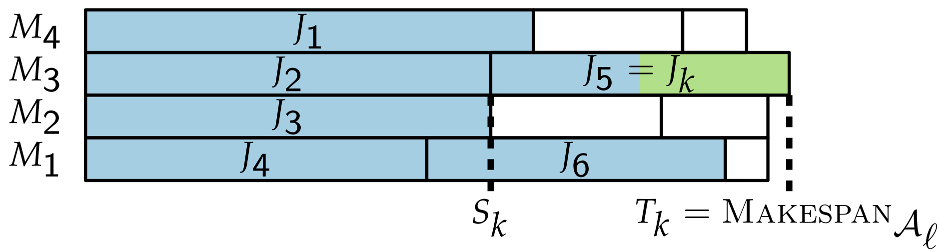
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ 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.

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

- Solution is optimal for $J_1, \ldots, J_k$
- Hence, solution is optimal for $J_1, \ldots, J_n$



Multiprocessor Scheduling – PTAS (Proof)

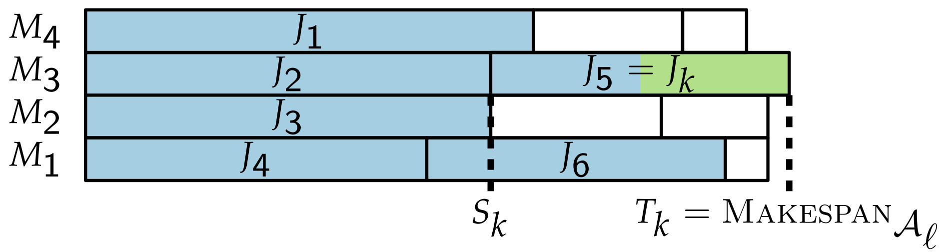
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

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

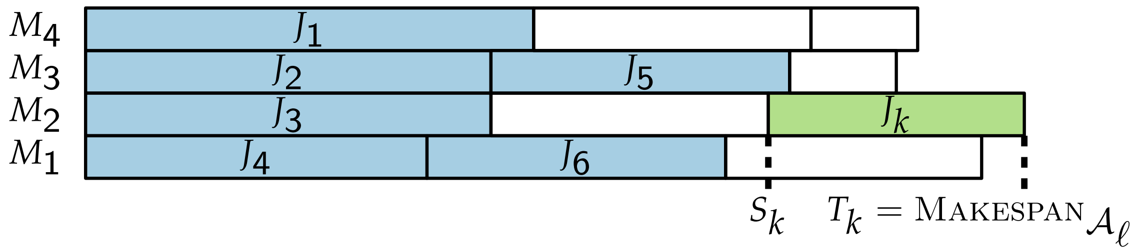
Proof. Let $J_k = (S_k, T_k)$ be the last job, that is, T_k determines the makespan.

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

- Solution is optimal for $J_1, \dots, J_k$
- Hence, solution is optimal for $J_1, \dots, J_n$



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



Multiprocessor Scheduling – PTAS (Proof)

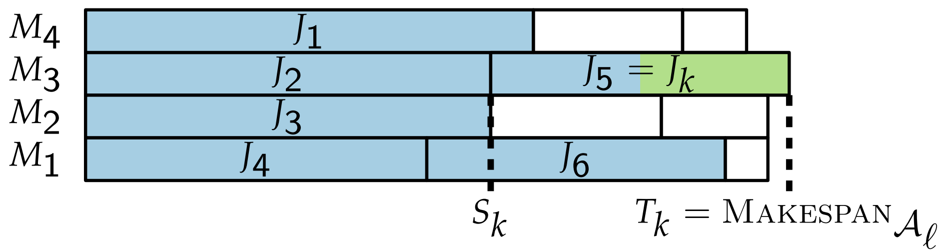
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

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

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

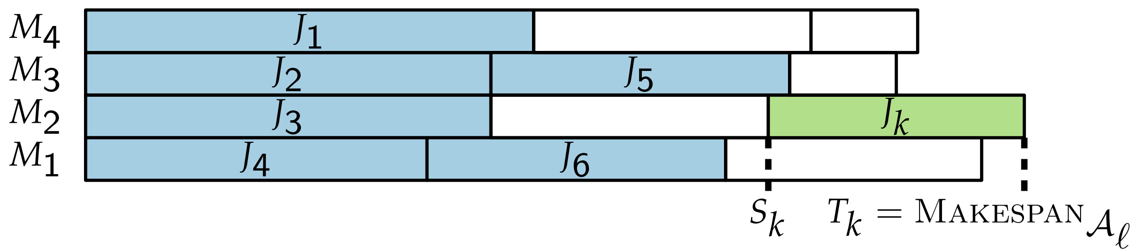
Case 1. J_k is one of the longest ℓ jobs $J_1, \dots, J_\ell$.

- Solution is optimal for $J_1, \dots, J_k$
- Hence, solution is optimal for $J_1, \dots, J_n$



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

- Similar analysis to ListScheduling
- Use that there are $\ell + 1$ jobs that are at least as long as J_k (including J_k).



Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of **Case 2**.

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

$$T_k = S_k + p_k$$

Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of **Case 2**.

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

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

Multiprocessor Scheduling – PTAS (Proof)

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

$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$

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

Proof of **Case 2**.

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

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

Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of **Case 2**.

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

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

Multiprocessor Scheduling – PTAS (Proof)

Theorem 8.
For constant $1 \le \ell \le n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1 - \frac{1}{m}}{1 + \lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

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

Proof of **Case 2**.

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

$$\begin{aligned}
T_k &= S_k + p_k \\
&\le \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
&= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
&\le T_{\text{OPT}} + \left(1 - \frac{1}{m}\right) \cdot T_{\text{OPT}}
\end{aligned}$$

can we do better?

Multiprocessor Scheduling – PTAS (Proof)

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

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

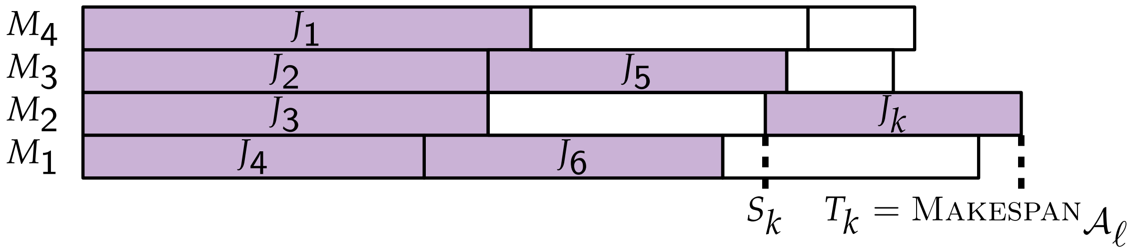
Proof of **Case 2**.

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

$$T_{\text{OPT}} \geq p_k \cdot$$

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

can we do better?



Multiprocessor Scheduling – PTAS (Proof)

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

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

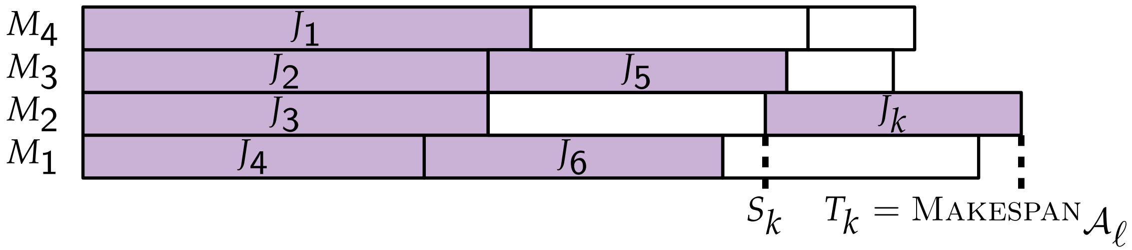
Proof of **Case 2**.

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

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

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

can we do better?



Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of **Case 2**.

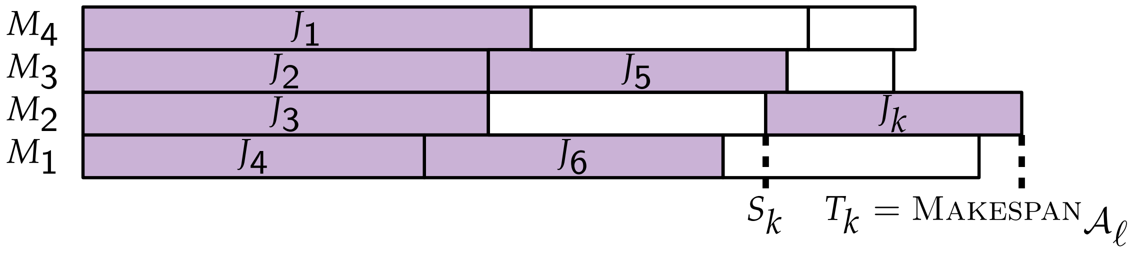
- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{\text{OPT}} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- Consider only $J_1, \dots, J_\ell, J_k$:

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

one machine has this many jobs*

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

can we do better?



Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of Case 2.

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

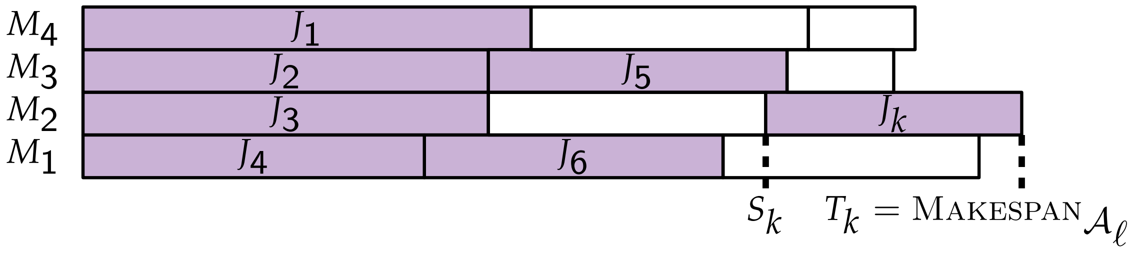
$$T_{\mathrm{OPT}} \geq p_k \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right)$$

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

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

can we do better?



Multiprocessor Scheduling – PTAS (Proof)

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

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

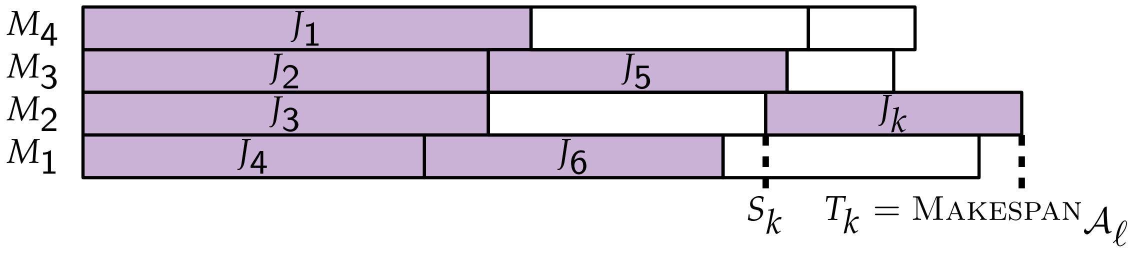
Proof of Case 2.

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

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

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



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

can we do better?

Multiprocessor Scheduling – PTAS (Proof)

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

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

Proof of Case 2.

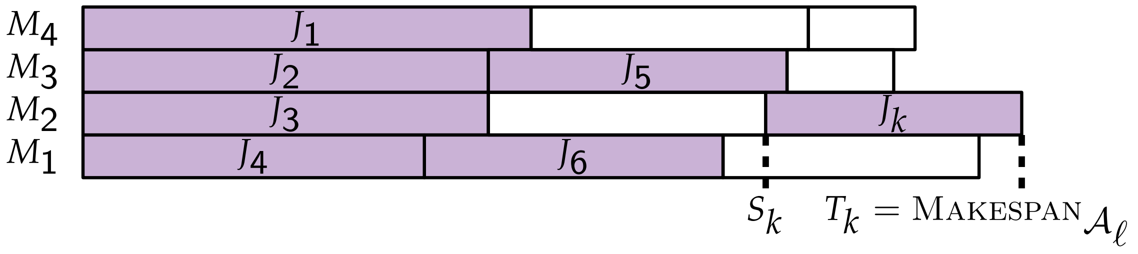
- $S_k \leq \frac{1}{m} \sum_{i \neq k} p_i$
- $T_{\text{OPT}} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$
- Consider only $J_1, \ldots, J_\ell, J_k$:

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

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



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

can we do better?

Multiprocessor Scheduling – PTAS (Proof)

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

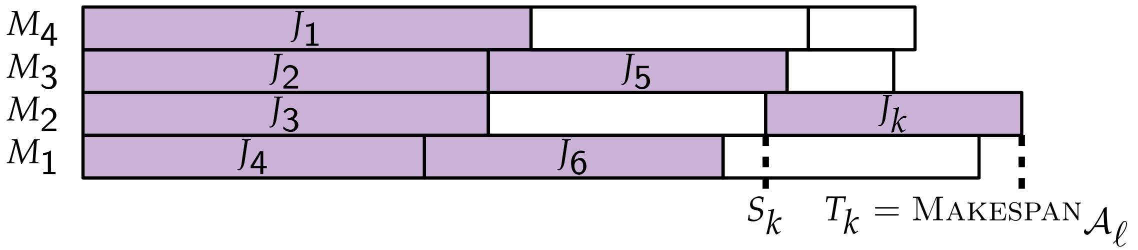
$\mathcal{A}_\ell(J_1, \dots, J_n, m)$
Sort jobs in descending order of runtime.
Schedule the ℓ longest jobs $J_1, \dots, J_\ell$ optimally.
Use ListScheduling for the remaining jobs $J_{\ell+1}, \dots, J_n$.

Proof of **Case 2**.

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

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



$$
\begin{aligned}
T_k &= S_k + p_k \\
&\leq \frac{1}{m} \cdot \sum_{i \neq k} p_i + p_k \\
&= \frac{1}{m} \cdot \sum_{i=1}^{m} p_i + \left(1 - \frac{1}{m}\right) \cdot p_k \\
&\leq T_{\mathrm{OPT}} + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor} \cdot T_{\mathrm{OPT}}
\end{aligned}
$$

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 more fine-grained classifications.

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 more fine-grained classifications.
- 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 (more later in this 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 more fine-grained classifications.
- 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 (more later in this 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.

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 more fine-grained classifications.
- 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 (more later in this 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.
- There is a whole lecture on approximation algorithms this semester!
  `https://wuecampus.uni-wuerzburg.de/moodle/course/view.php?id=62943`

Literature

Main references

- [Jansen & Margraf, 2008: Ch3] "Approximative Algorithmen und Nichtapproximierbarkeit"
- [Williamson & Shmoys, 2011: Ch3] "The Design of Approximation Algorithms"

Another book recommendation:

- [Vazirani, 2013] "Approximation Algorithms"