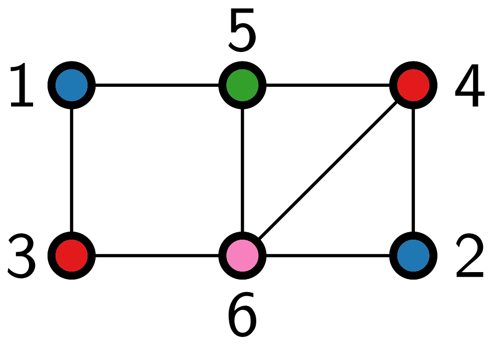
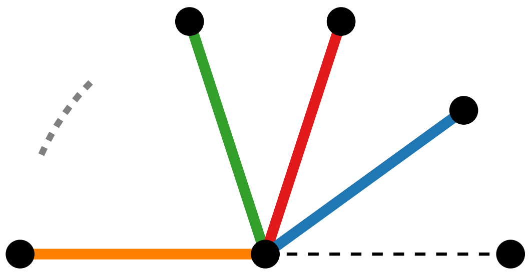
Advanced Algorithms

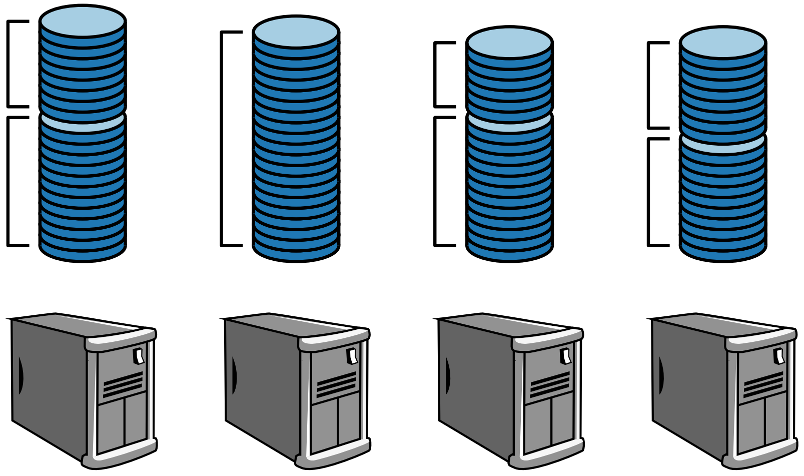
Approximation algorithms

Coloring and scheduling problems

Jonathan Klawitter · WS20



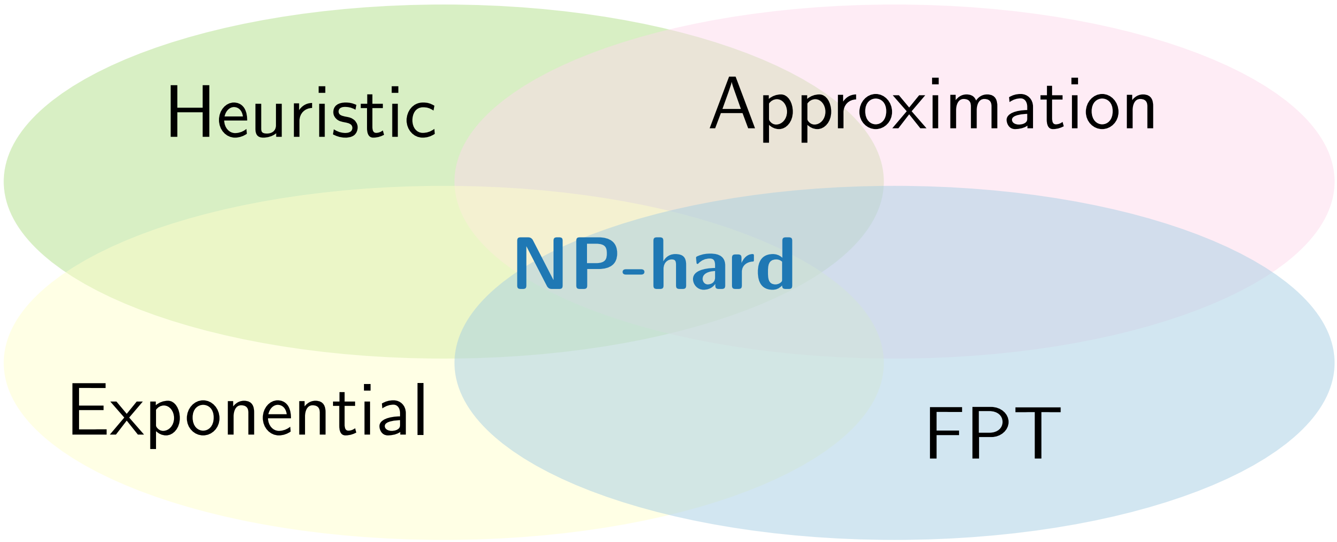




Dealing with NP-hard problems

What should we do?

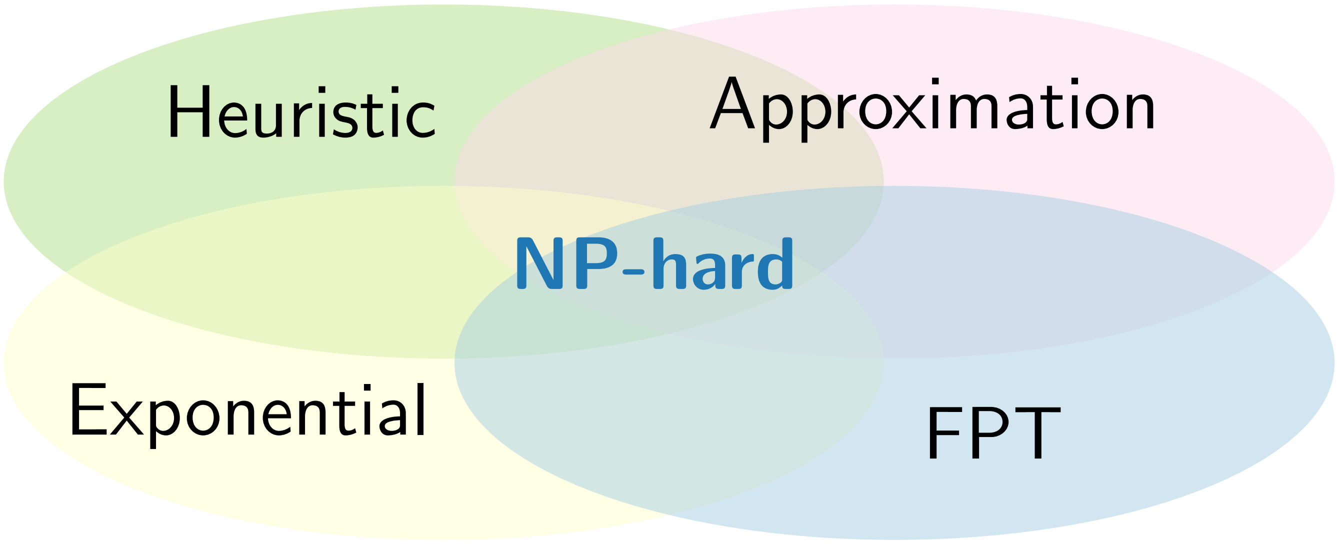
- Sacrifice optimality for speed
  - Heuristics
  - Approximation Algorithms
- Optimal Solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms



Dealing with NP-hard 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 optimisation problems, we cannot compute the optimal solution of each instance efficiently (unless $\mathrm{P} = \mathrm{NP}$).
- Heuristics offer no guarantee on the quality of their solutions.

Approximation algorithms

Problem.

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

Goal.

- Design **approximation algorithms** that
  - 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 optimisation problems, we cannot compute the optimal solution of each instance efficiently (unless $\mathrm{P} = \mathrm{NP}$).
- Heuristics offer no guarantee on the quality of their solutions.

Goal.

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

Overview.

- Approximation algorithms that compute solutions with/that are
  - additive guarantee,
  - relative guarantee,
  - "arbitraility good".

Approximation with additive guarantee

Definition.
Let Π be an optimisation problem and let $\mathcal{A}$ be a polynomial-time algorithm that computes the value $\mathcal{A}(I)$ for an instance I of Π.
$\mathcal{A}$ is called an **approximation algorithm with additive guarantee** δ if

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

for every instance I of Π.

Approximation with additive guarantee

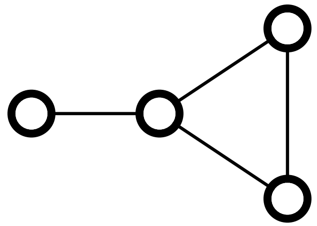
Definition.
Let Π be an optimisation problem and let $\mathcal{A}$ be a polynomial-time algorithm that computes the value $\mathcal{A}(I)$ for an instance I of Π.
$\mathcal{A}$ is called an **approximation algorithm with additive guarantee** δ if

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

for every instance I of Π.

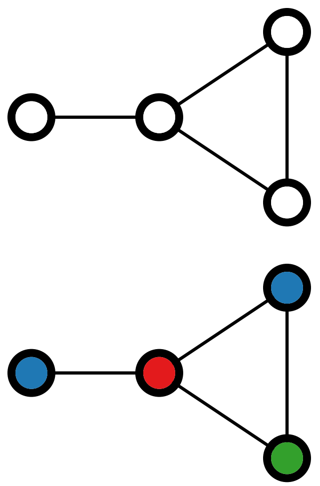
- Most problems do not admit an approximation algorithm with additive guarantee.

Minimum vertex coloring



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

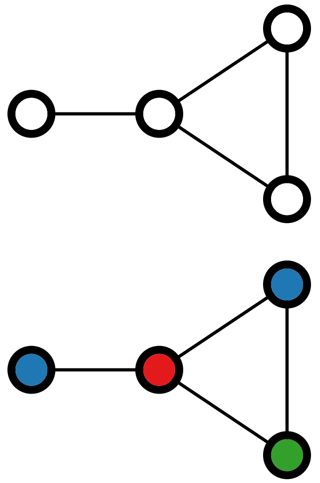
Minimum vertex coloring



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

Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

Minimum vertex coloring

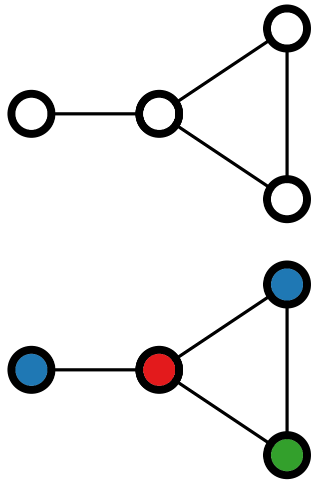


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

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

Minimum vertex coloring

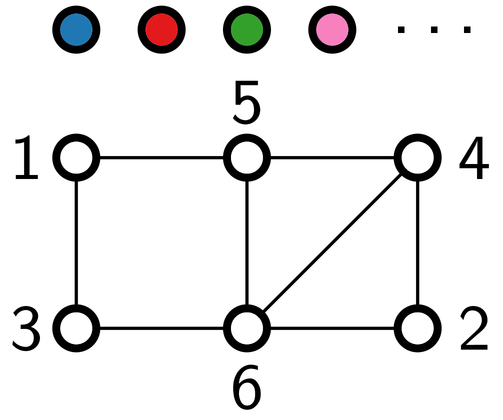


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

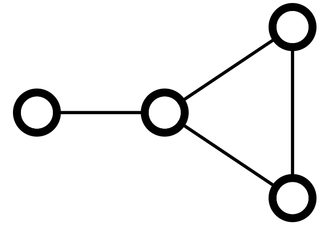
Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

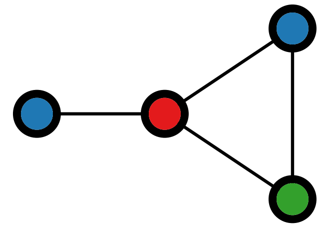
- Min Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

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



Minimum vertex coloring



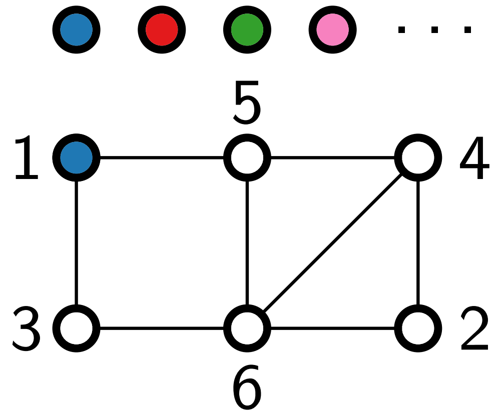


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

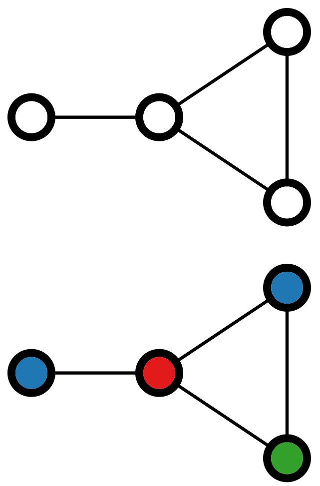
Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

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

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



Minimum vertex coloring

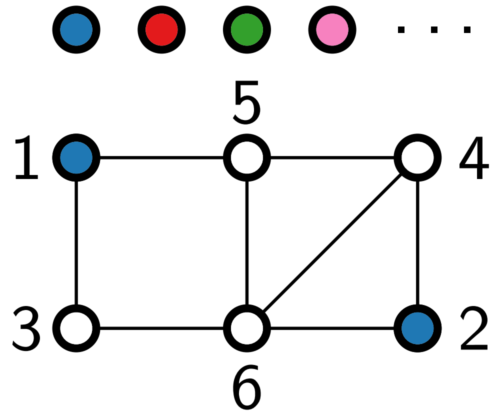


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

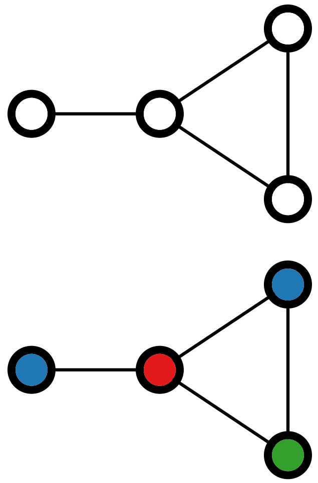
Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

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

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



Minimum vertex coloring



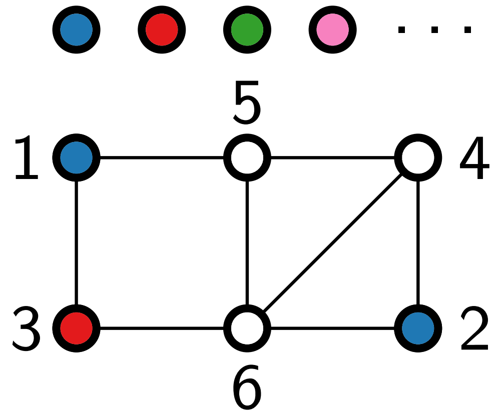
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

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

$\textsc{GreedyVertexColoring}(G)$

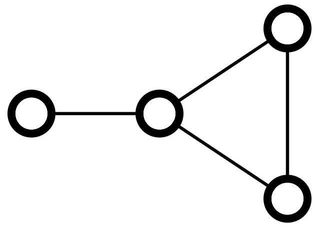
Color vertices in some order with lowest feasible color.

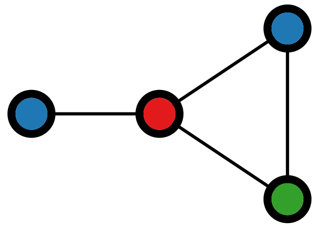


Minimum vertex coloring

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

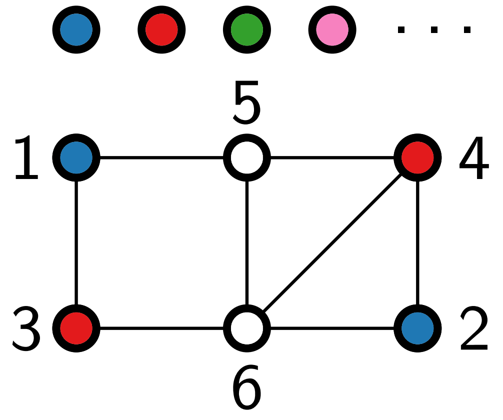
Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.



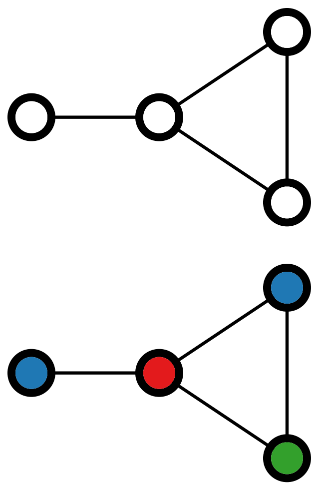


- Min Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

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



Minimum vertex coloring



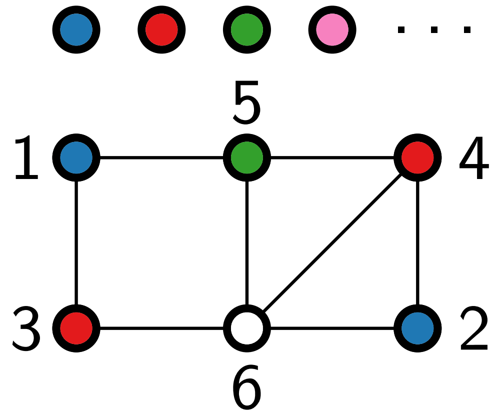
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

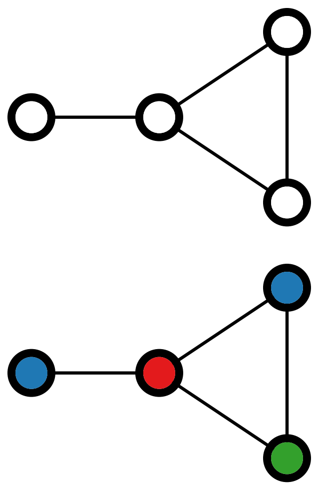
- Min Vertex Coloring is NP-hard.
- Even Vertex 3-Coloring is NP-complete.

$\textsc{GreedyVertexColoring}(G)$

Color vertices in some order with lowest feasible color.



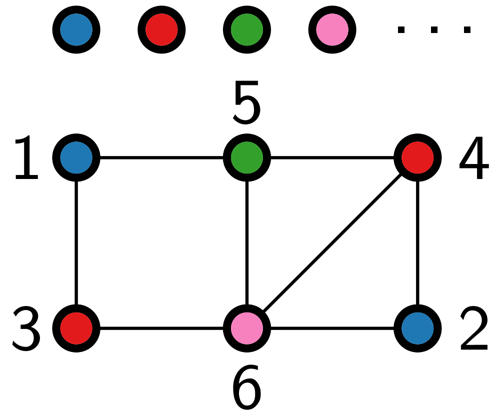
Minimum vertex coloring



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

Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.

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



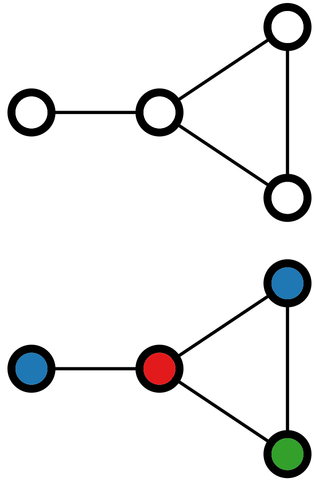
$\textsc{GreedyVertexColoring}(G)$

Color vertices in some order with lowest feasible color.

Minimum vertex coloring

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

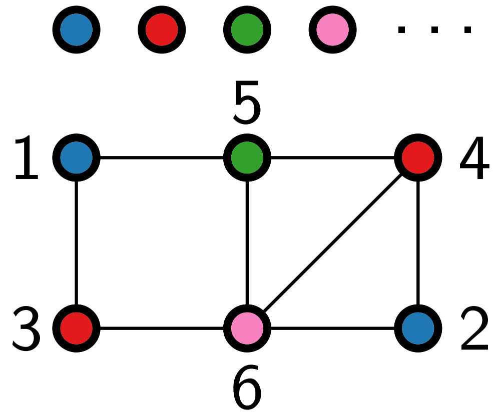
Output. A **vertex coloring**, that is, an assignment of colors to the vertices of G such that now two adjacent vertices get the same color, with minimum number of colors.



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

$\textsc{GreedyVertexColoring}(G)$

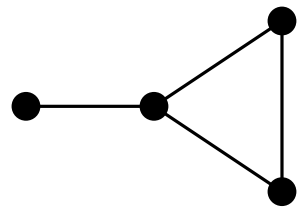
Color vertices in some order with lowest feasible color.



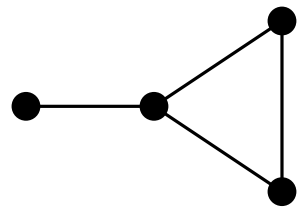
Theorem 1.
The algorithm `GreedyVertexColoring` computes a vertex coloring with at most $\Delta + 1$ colors in $\mathcal{O}(n + m)$ time.
Hence, it has an additive approximation gurantee of $\Delta - 1$.

Minimum edge coloring

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

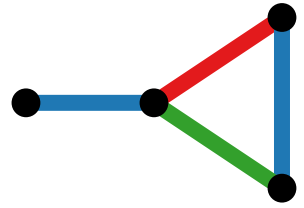


Minimum edge coloring

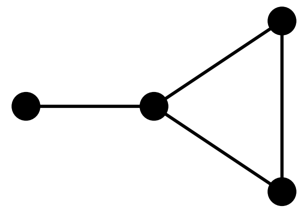


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Output. An **edge coloring**, that is, an assignment of colors to the edges of G such that now two incident edges get the same color, with minimum number of colors.

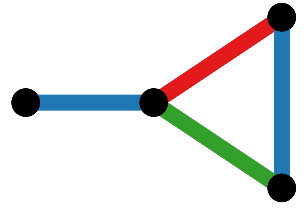


Minimum edge coloring



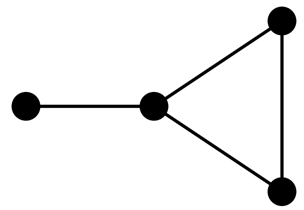
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.

Output. An **edge coloring**, that is, an assignment of colors to the edges of G such that now two incident edges get the same color, with minimum number of colors.

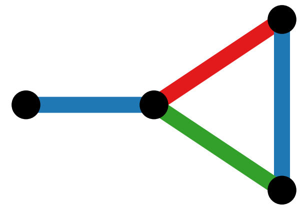


- Min Edge Coloring is NP-hard.
- Even Edge 3-Coloring is NP-complete.

Minimum edge coloring



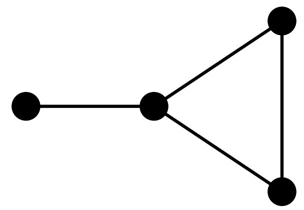
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.



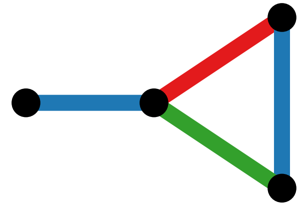
Output. An **edge coloring**, that is, an assignment of colors to the edges of G such that now two incident edges get the same color, with minimum number of colors.

- Min 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 lower bounded by

Minimum edge coloring

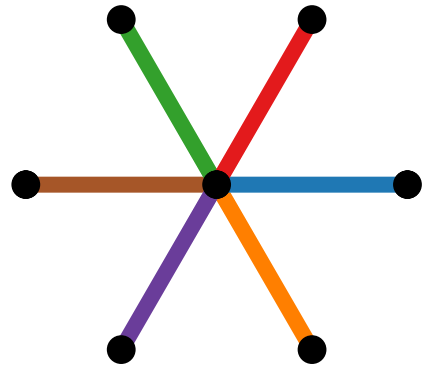


Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.



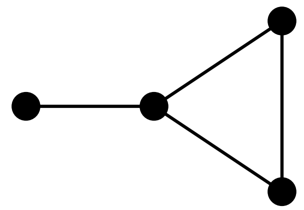
Output. An **edge coloring**, that is, an assignment of colors to the edges of G such that now two incident edges get the same color, with minimum number of colors.

- Min 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 lower bounded by Δ.

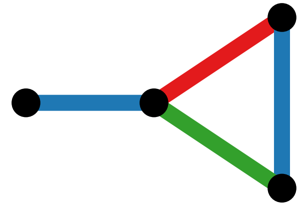


Minimum edge coloring

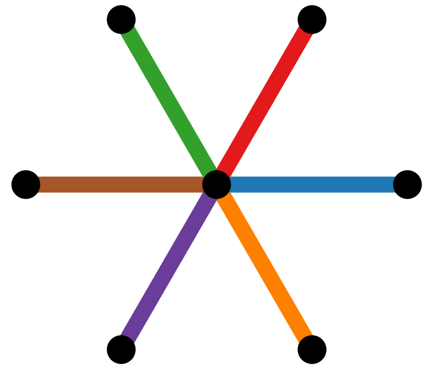
Input. A graph $G = (V, E)$. Let Δ be the maximum degree of G.



Output. An **edge coloring**, that is, an assignment of colors to the edges of G such that now two incident edges get the same color, with minimum number of colors.



- Min 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 lower bounded 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 holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

Minimum edge coloring – upper bound

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

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

- Base case $m = 1$ is trivial.

Minimum edge coloring – upper bound

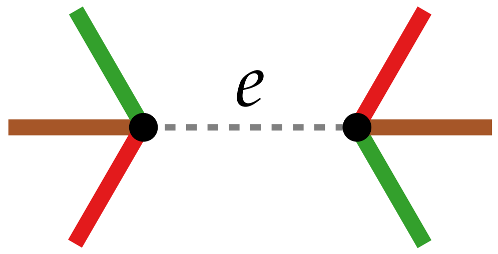
Vizing's Theorem.
For every graph $G = (V, E)$ with maximum degree Δ holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

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

- Base case $m = 1$ is trivial.

Let G be a graph on m edges and e an edge of G.

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



Minimum edge coloring – upper bound

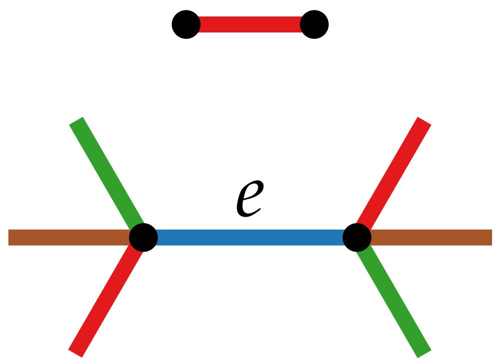
Vizing's Theorem.
For every graph $G = (V, E)$ with maximum degree Δ holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

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

- Base case $m = 1$ is trivial.

Let G be a graph on m edges and e 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 $\Delta(G) + 1$.



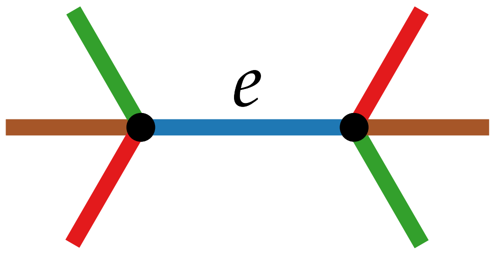
Minimum edge coloring – upper bound

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

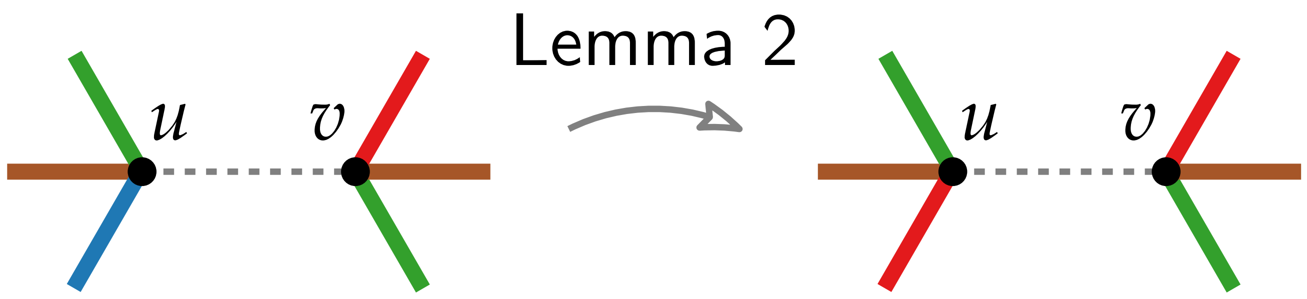
Proof by induction on $m = |E|$.

- Base case $m = 1$ is trivial.

Let G be a graph on m edges and e 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 coloring such that u and v (of $e = \{u, v\}$) miss the same color α.



Minimum edge coloring – upper bound

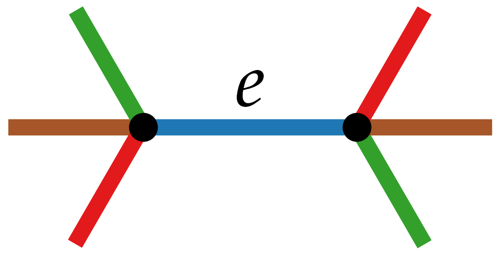
Vizing's Theorem.
For every graph $G = (V, E)$ with maximum degree Δ holds that $\Delta \leq \chi'(G) \leq \Delta + 1$.

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

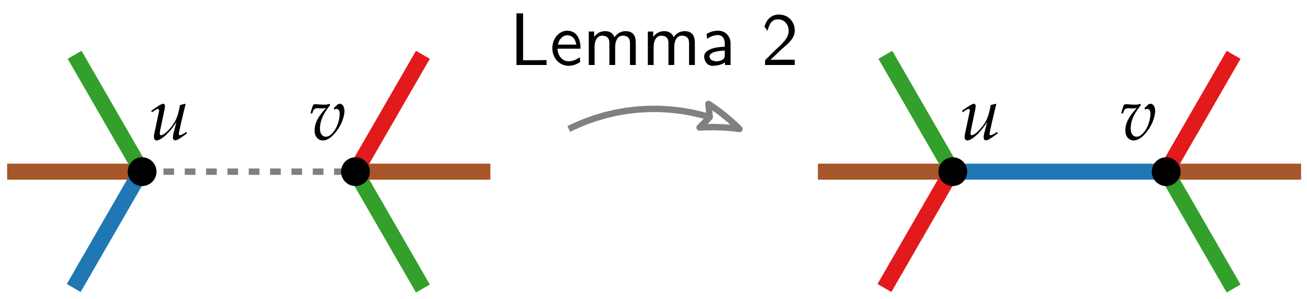
- Base case $m = 1$ is trivial.

Let G be a graph on m edges and e 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 coloring such that u and v (of $e = \{u, v\}$) miss the same color α.
- Then color e with with α.



Minimum edge coloring – recoloring

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

Minimum edge coloring – recoloring

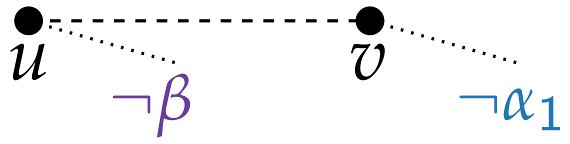
Lemma 2.
Let G have a $(\Delta + 1)$ edge coloring c, let u, v be non-adjacent, and $\deg(u), \deg(v) < \Delta$. Then c can be changed such that u and v miss the same color.

Proof. Note, each vertex is **missing** a color.

Minimum edge coloring – recoloring

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

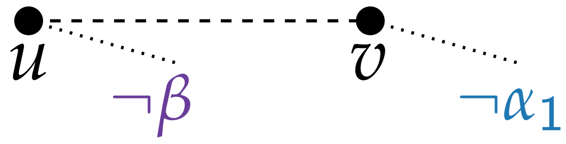
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:



Minimum edge coloring – recoloring

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

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



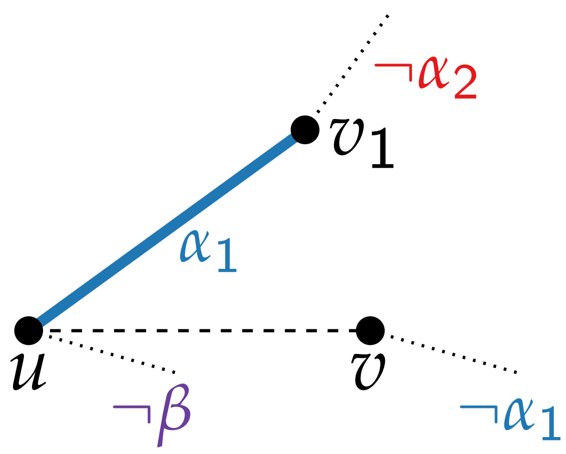
VizingRecoloring$(G = (V, E), u, c, \alpha_1)$

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

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

Minimum edge coloring – recoloring

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



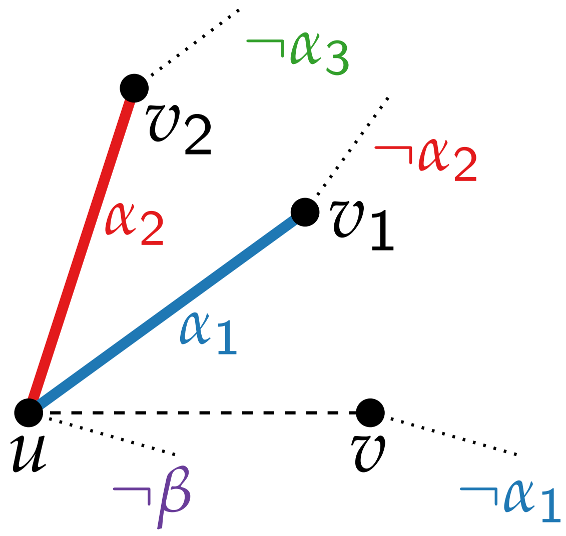
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G=(V,E), u, c, \alpha_1)$

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

Minimum edge coloring – recoloring

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



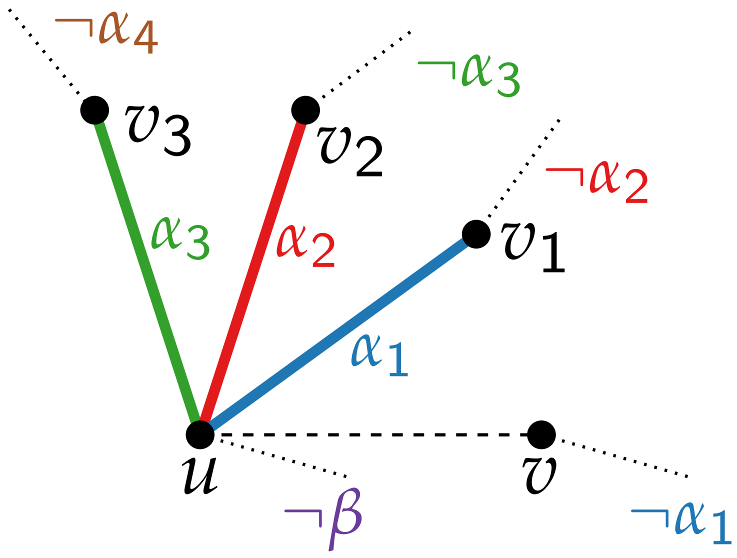
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

Minimum edge coloring – recoloring

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



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

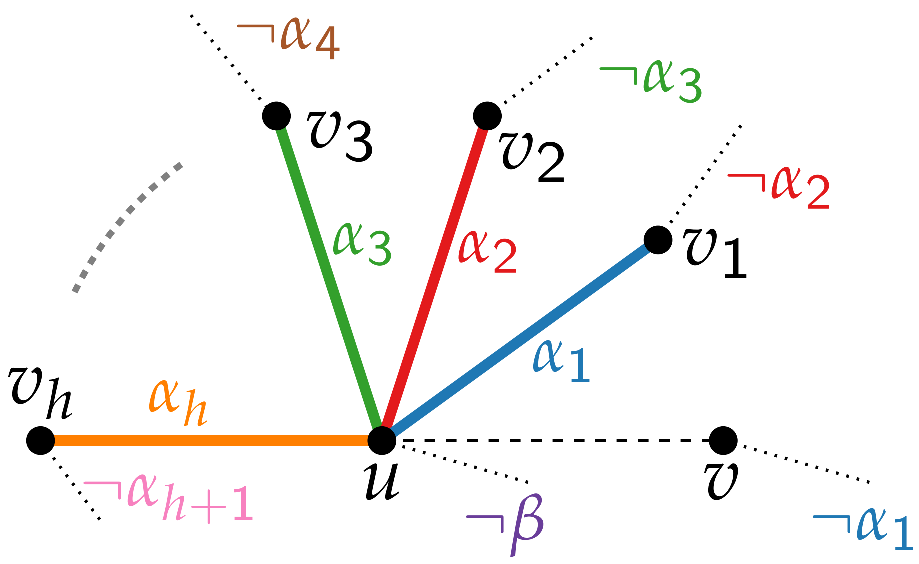
$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

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

Minimum edge coloring – recoloring

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



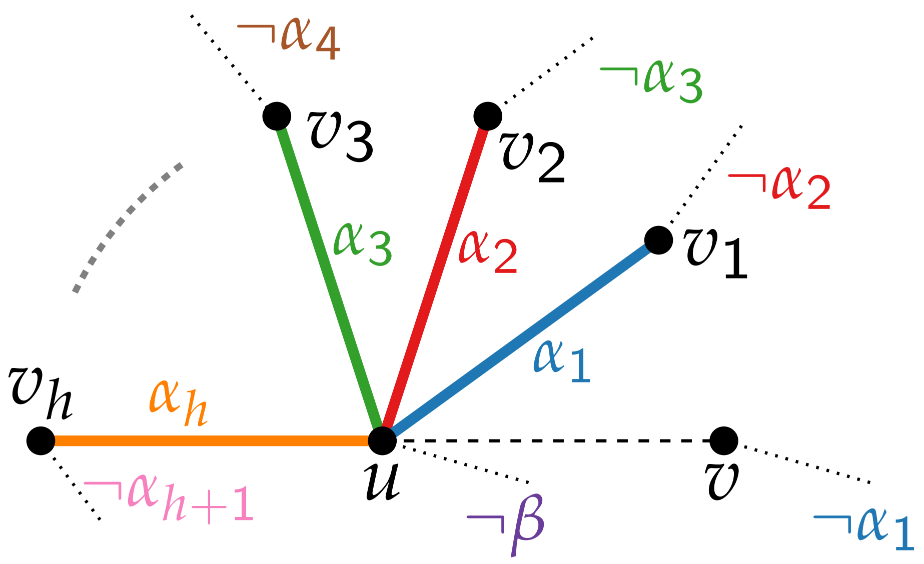
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

Minimum edge coloring – recoloring

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



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

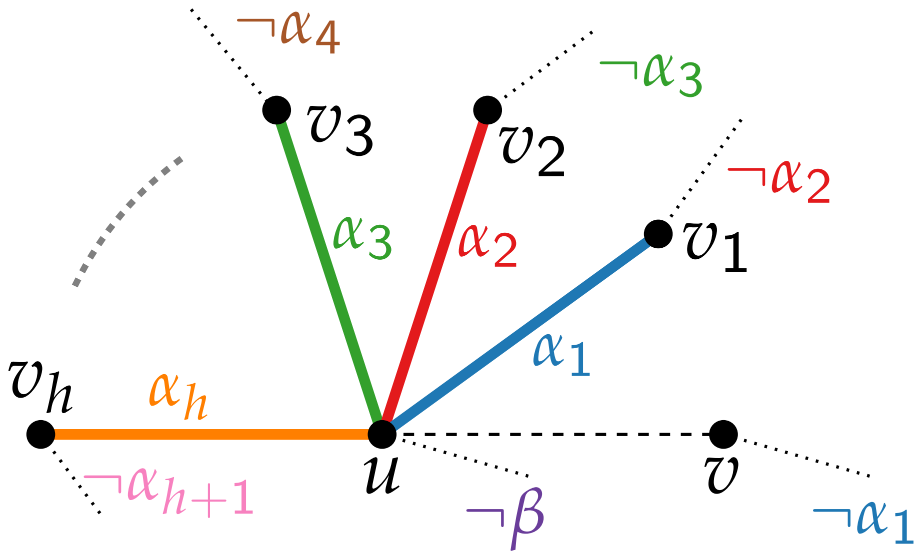
Case 1: u misses α_{h+1}.

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

Minimum edge coloring – recoloring

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

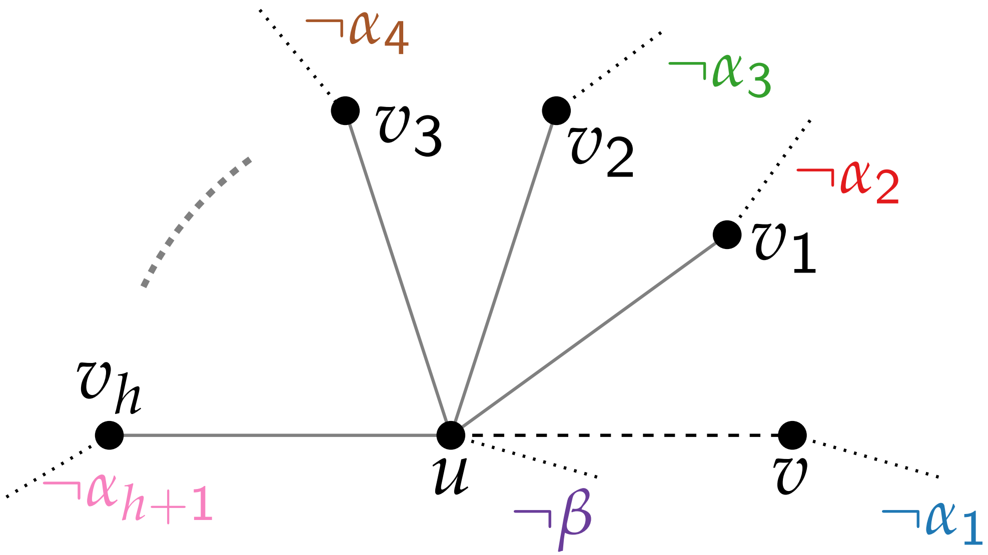


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

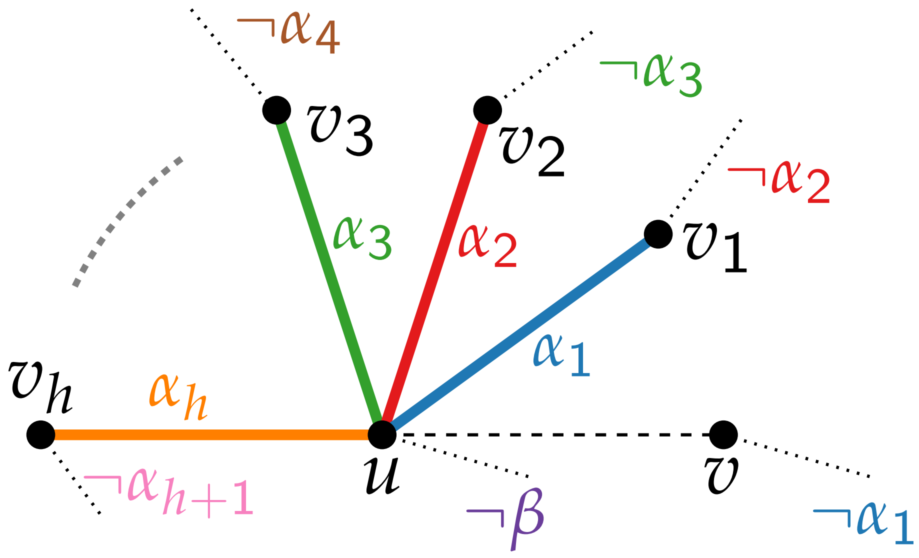
$i \leftarrow 1$
while $\exists w \in N(u)$: $c(\{u, w\}) = \alpha_i$ $\wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum edge coloring – recoloring

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



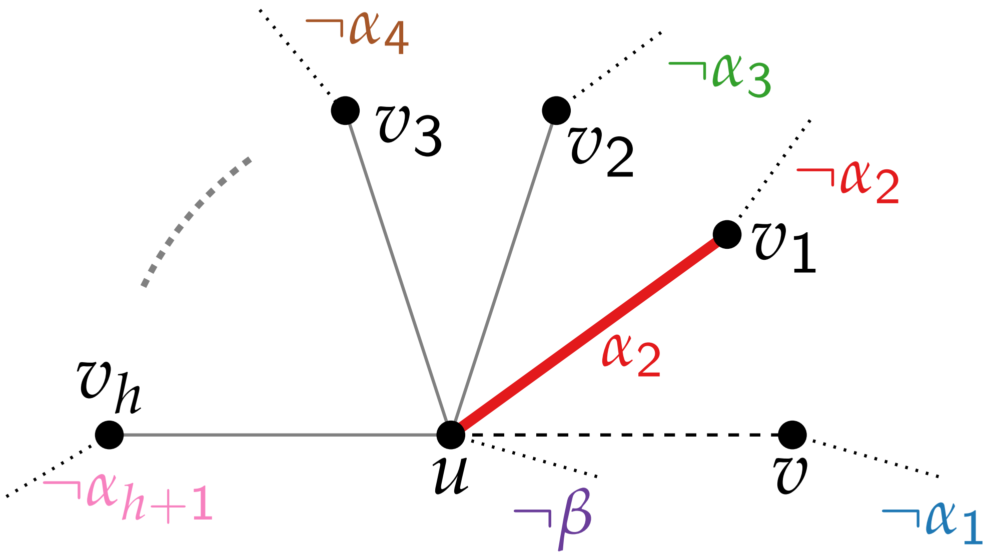
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

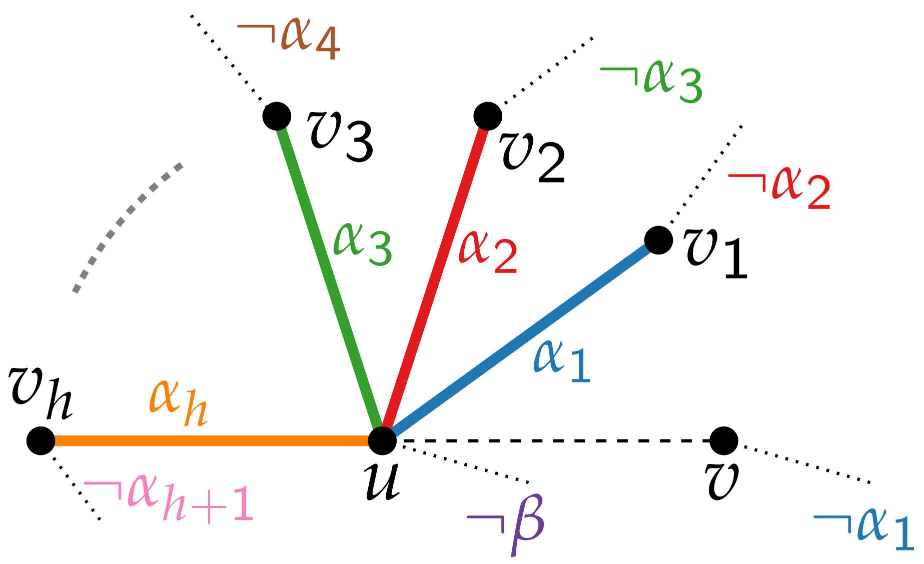
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum edge coloring – recoloring

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

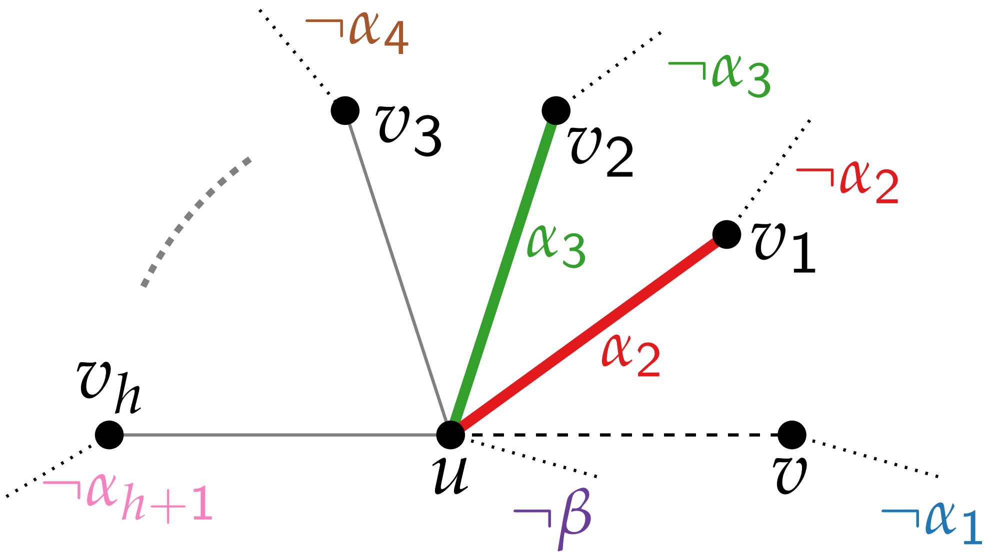


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G=(V,E), u, c, \alpha_1)$

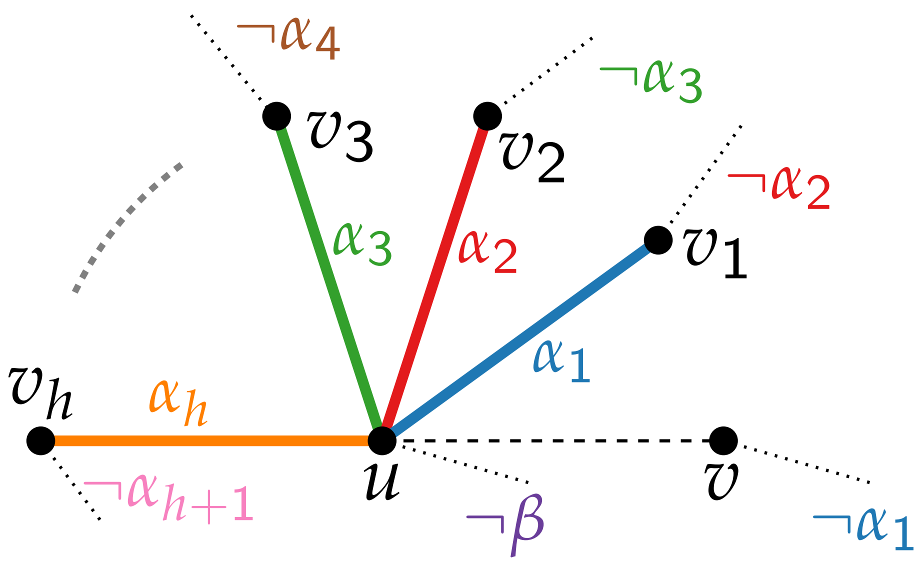
- $i \leftarrow 1$
- **while** $\exists w \in N(u)$: $c(\{u,w\}) = \alpha_i$ $\wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i{+}{+}$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

Case 1: u misses α_{h+1}.



Minimum edge coloring – recoloring

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

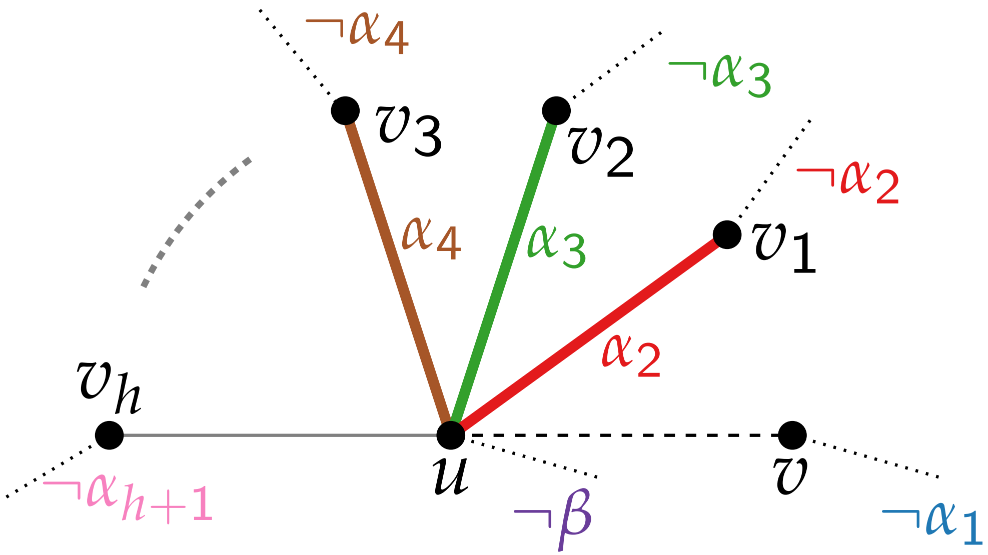


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V,E), u, c, \alpha_1)$

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

Case 1: u misses α_{h+1}.



Minimum edge coloring – recoloring

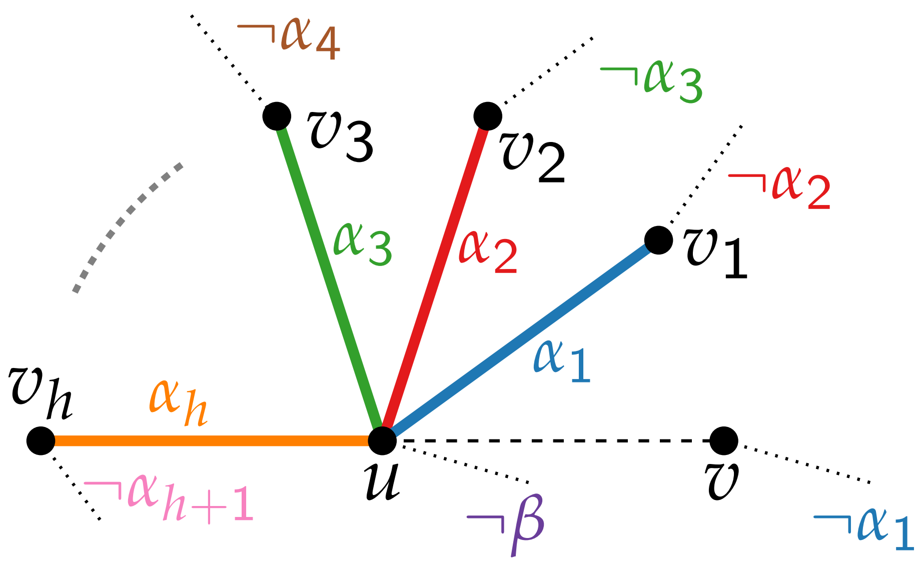
Lemma 2.
Let G have a $(\Delta+1)$ edge coloring c, let u, v be non-adjacent, and $\deg(u), \deg(v) < \Delta$. Then c can be changed such that u and v miss the same color.

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

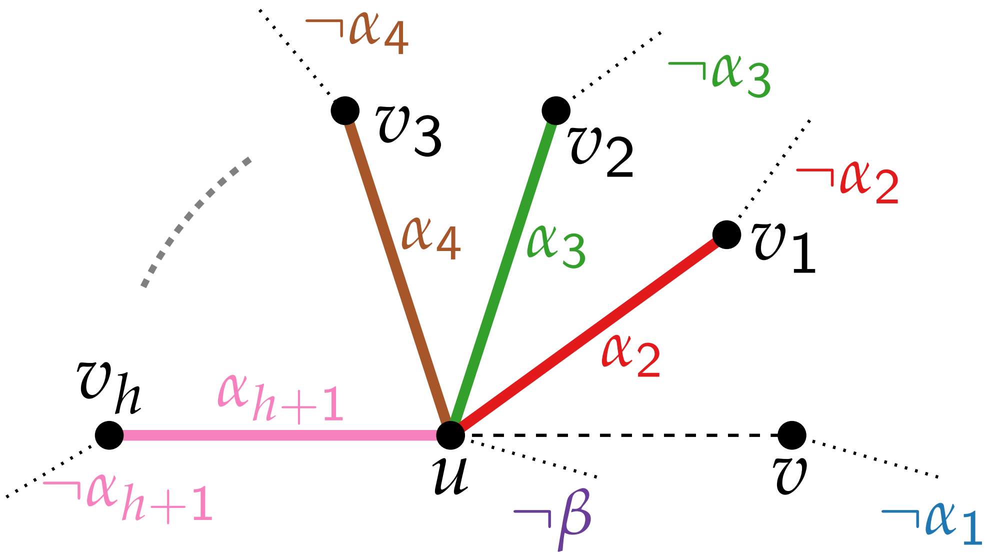
$\textsc{VizingRecoloring}(G=(V,E), u, c, \alpha_1)$

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

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



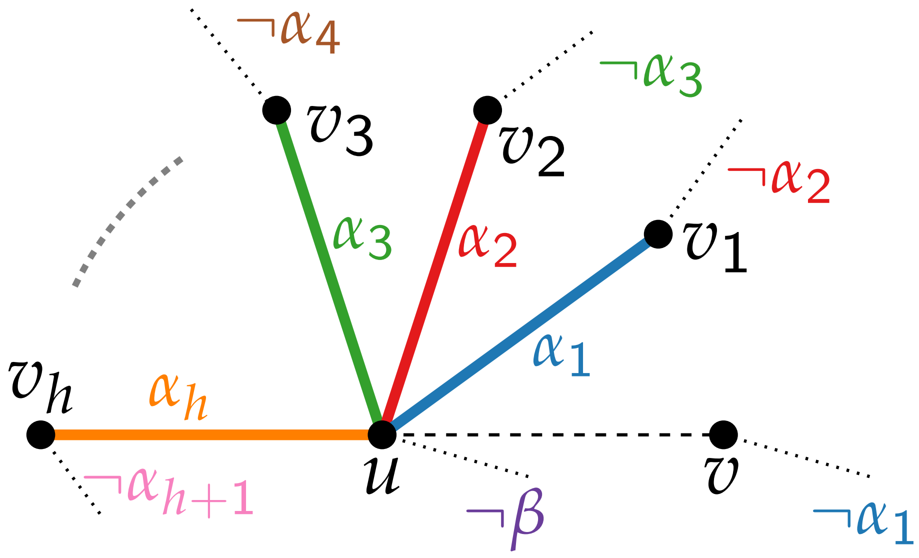
Case 1: u misses α_{h+1}.



Minimum edge coloring – recoloring

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

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

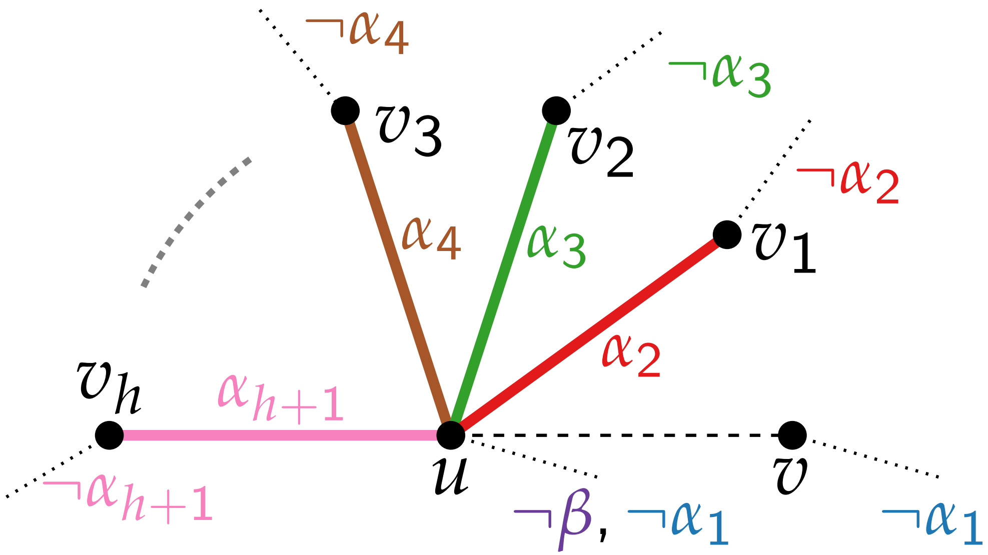


$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

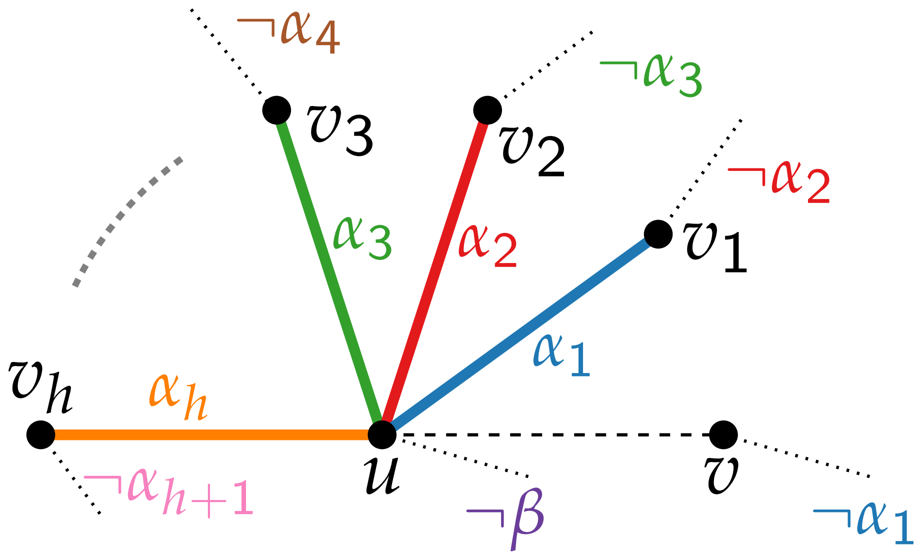
$i \leftarrow 1$
while $\exists w \in N(u)$: $c(\{u, w\}) = \alpha_i$ $\wedge$ $w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
- $v_i \leftarrow w$
- $\alpha_{i+1} \leftarrow$ min color missing at w
- $i{+}{+}$

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 have a $(\Delta+1)$ edge coloring c, let u, v be non-adjacent, and $\deg(u), \deg(v) < \Delta$. Then c can be changed such that u and v miss the same color.

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

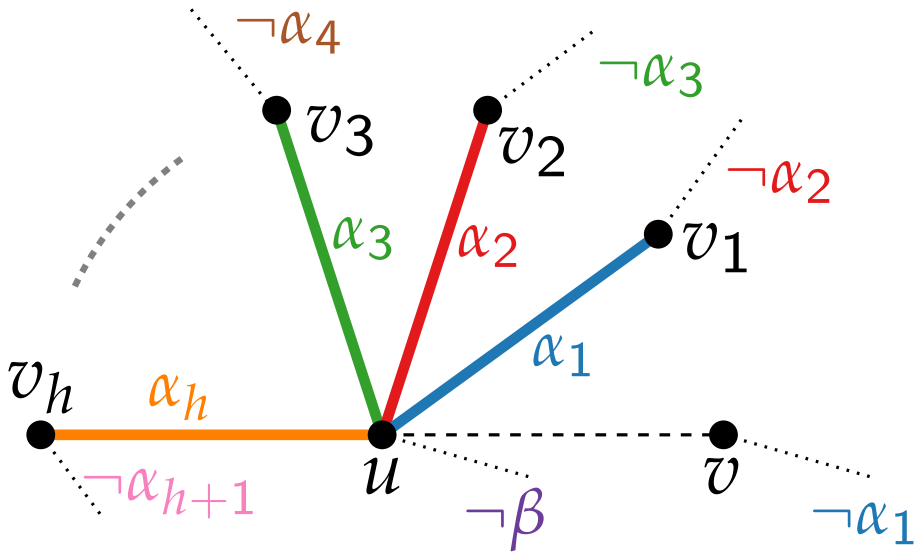
$\textsc{VizingRecoloring}(G=(V,E), u, c, \alpha_1)$

$i \leftarrow 1$
while $\exists w \in N(u)\colon\ c(\{u,w\}) = \alpha_i \ \wedge$
$w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
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 have a $(\Delta + 1)$ edge coloring c, let u, v be non-adjacent, and $\deg(u), \deg(v) < \Delta$. Then c can be changed such that u and v miss the same color.

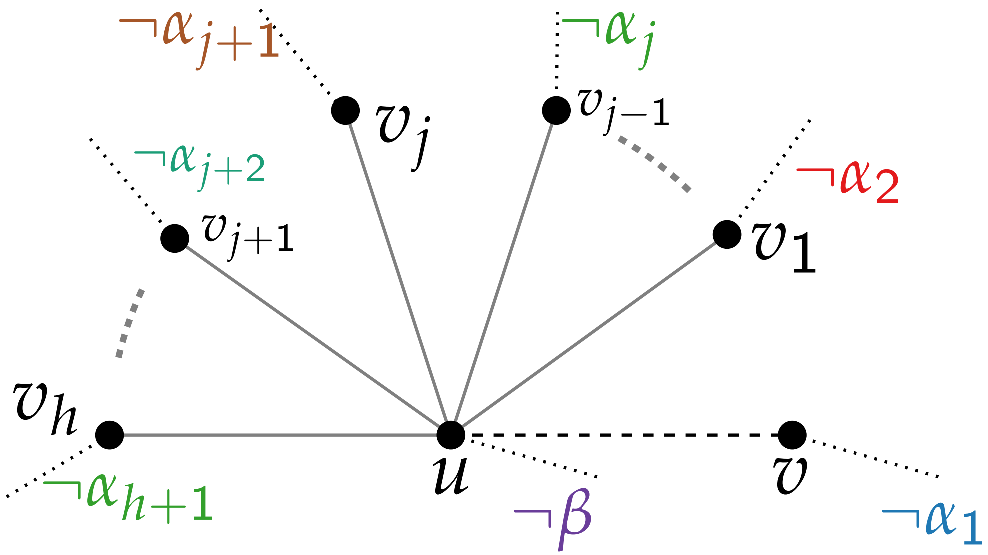


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

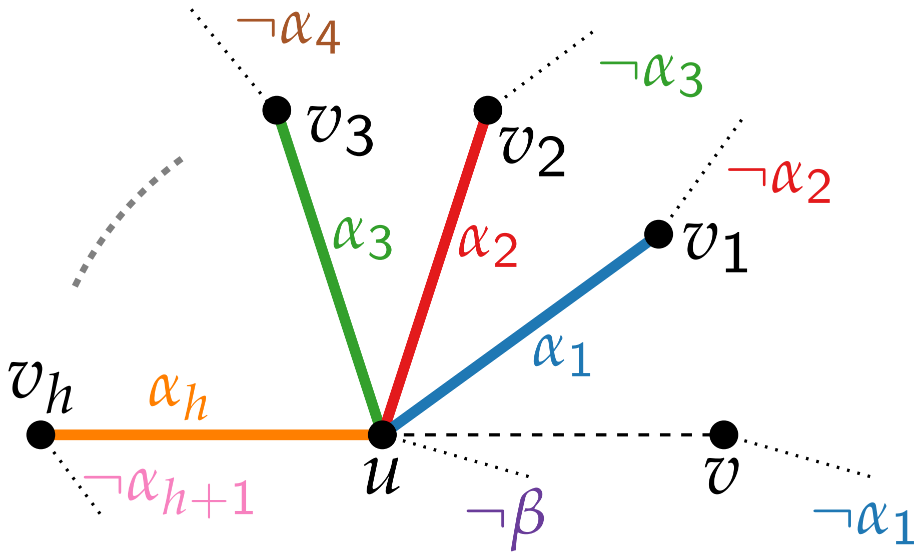
$i \leftarrow 1$
while $\exists w \in N(u)\colon\ c(\{u, w\}) = \alpha_i \ \wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

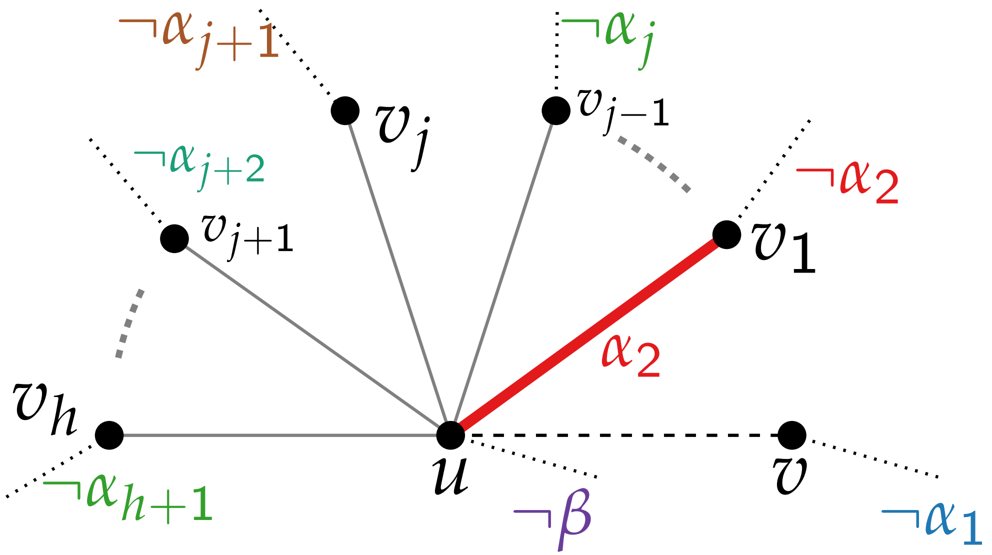


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

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



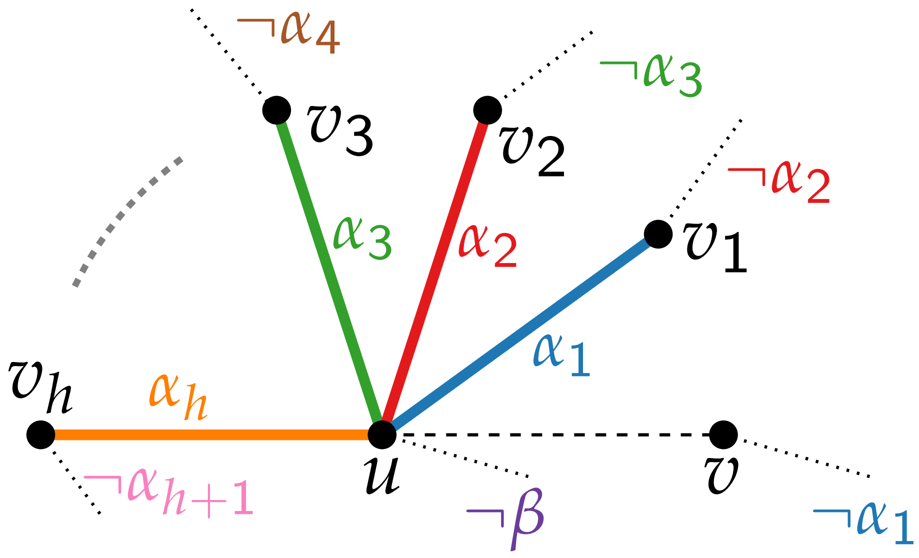
Minimum edge coloring – recoloring

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

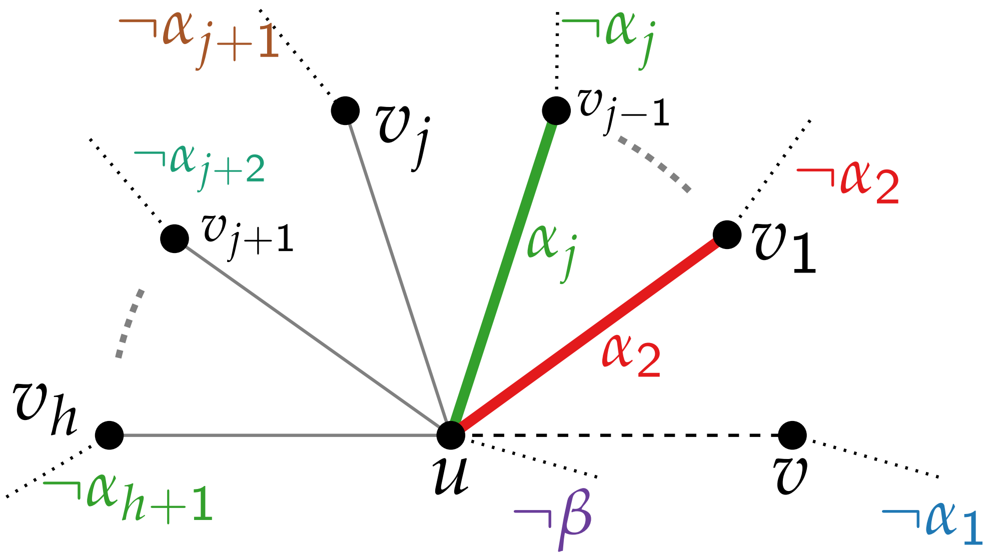
Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

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

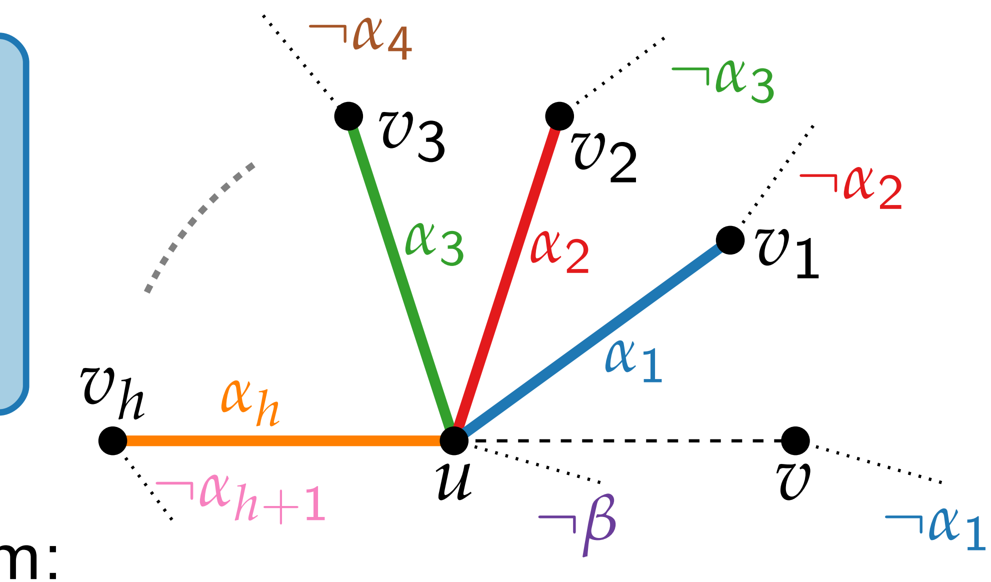


Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$.



Minimum edge coloring – recoloring

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

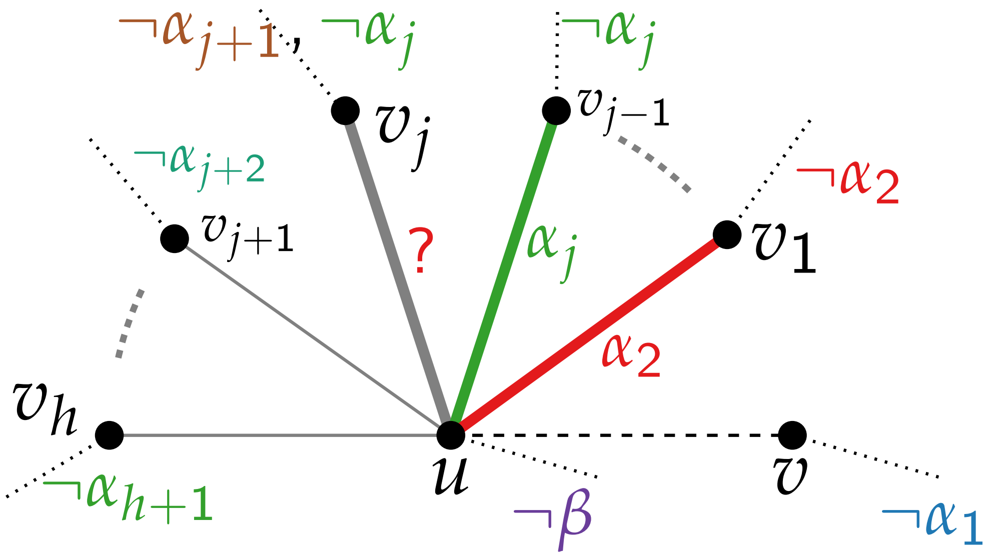


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G=(V,E), u, c, \alpha_1)$

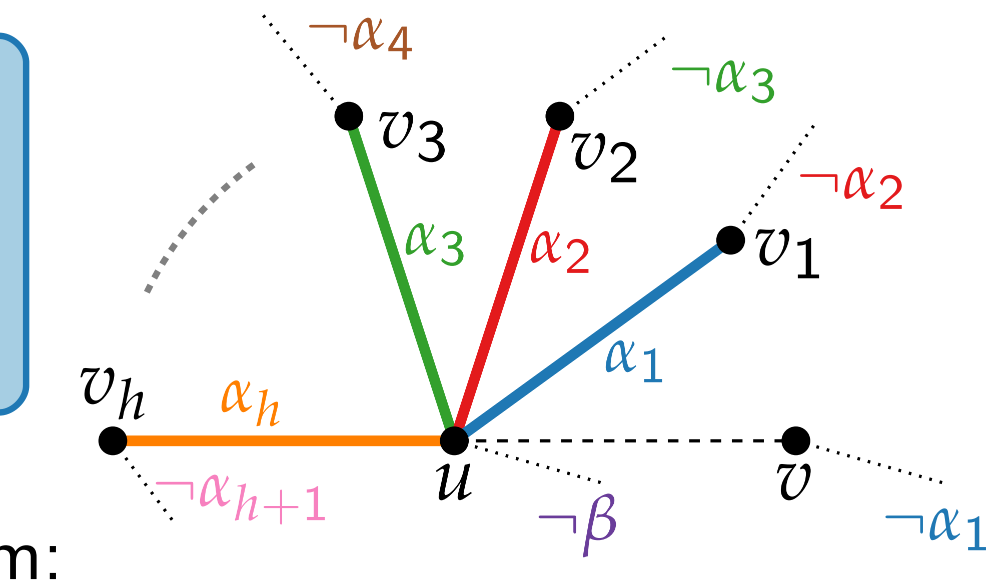
$i \leftarrow 1$
while $\exists w \in N(u)\colon\ c(\{u,w\}) = \alpha_i \ \wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

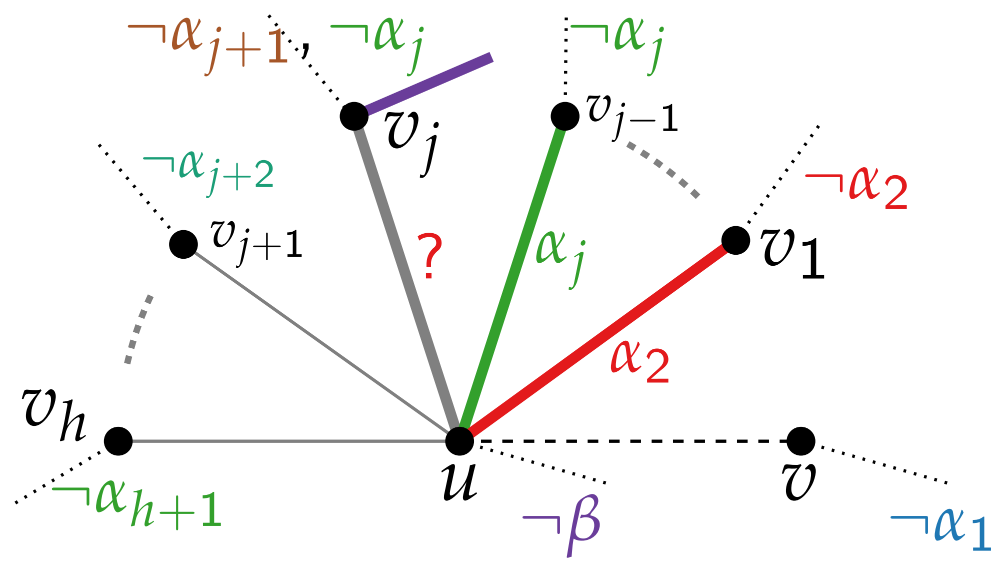


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

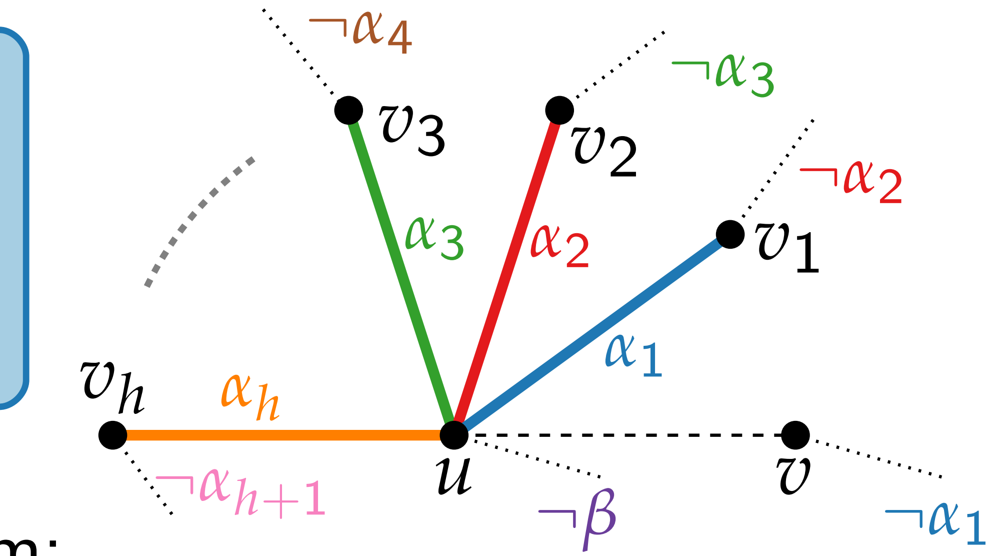
- $i \leftarrow 1$
- **while** $\exists w \in N(u)$: $c(\{u, w\}) = \alpha_i \wedge w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i{+}{+}$
- **return** $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

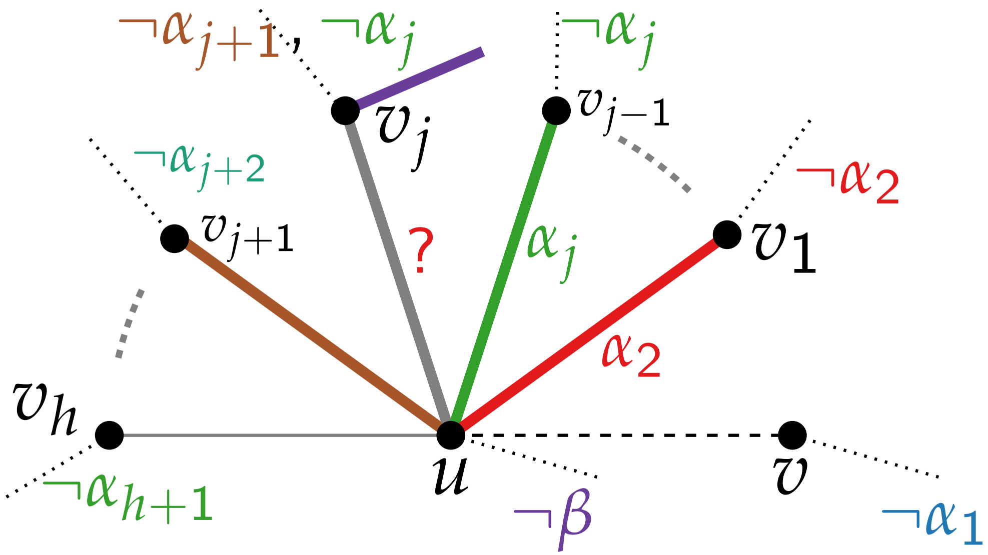


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

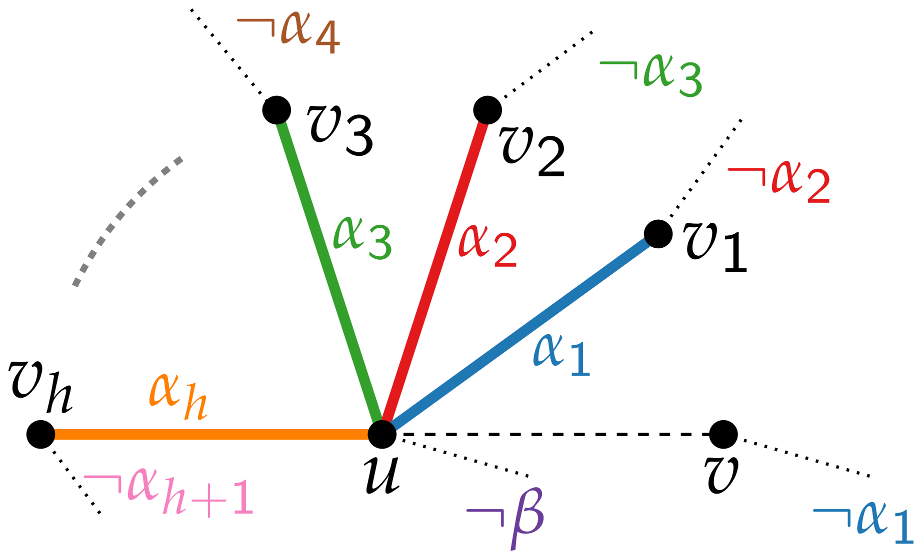
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon \; c(\{u, w\}) = \alpha_i \;\wedge\; w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i{+}{+}$
- **return** $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

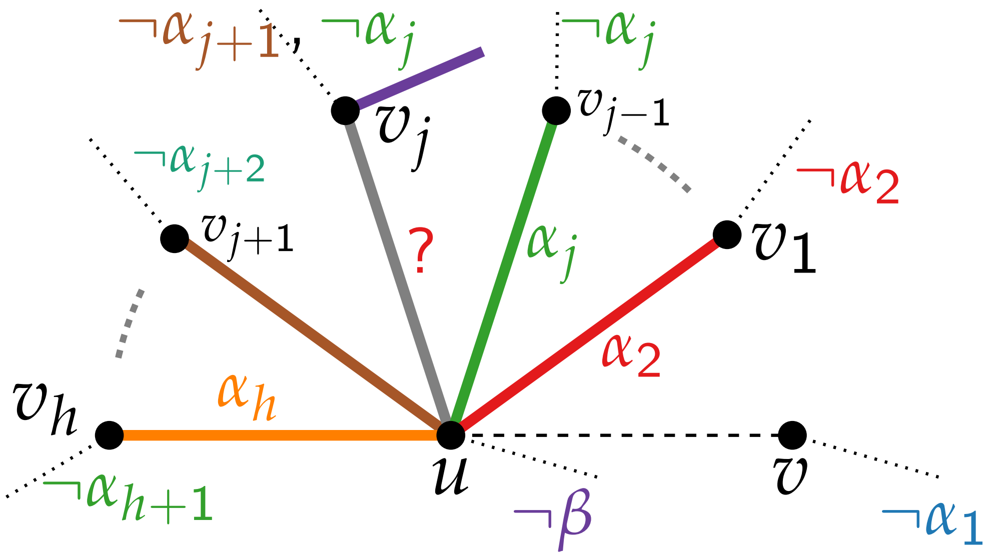


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

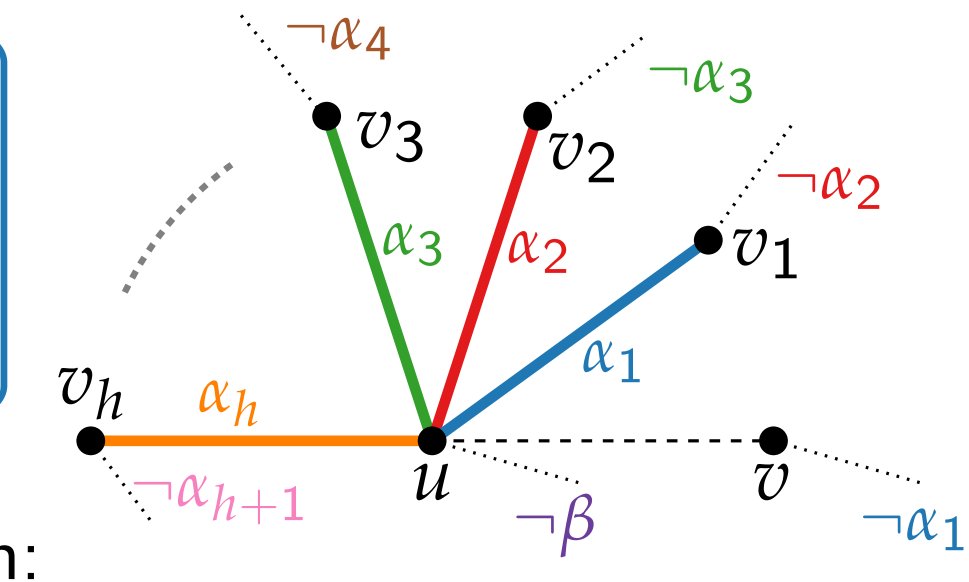
$i \leftarrow 1$
while $\exists w \in N(u)$: $c(\{u, w\}) = \alpha_i \;\wedge$ $w \notin \{v_1, \ldots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
return $v_1, \ldots, v_i; \alpha_1, \ldots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

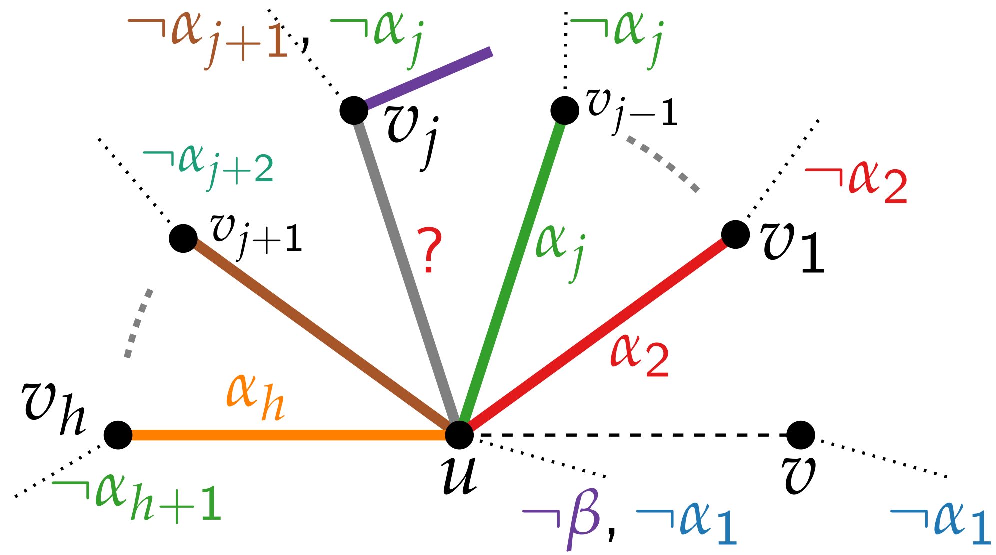


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

$\textsc{VizingRecoloring}(G = (V, E), u, c, \alpha_1)$

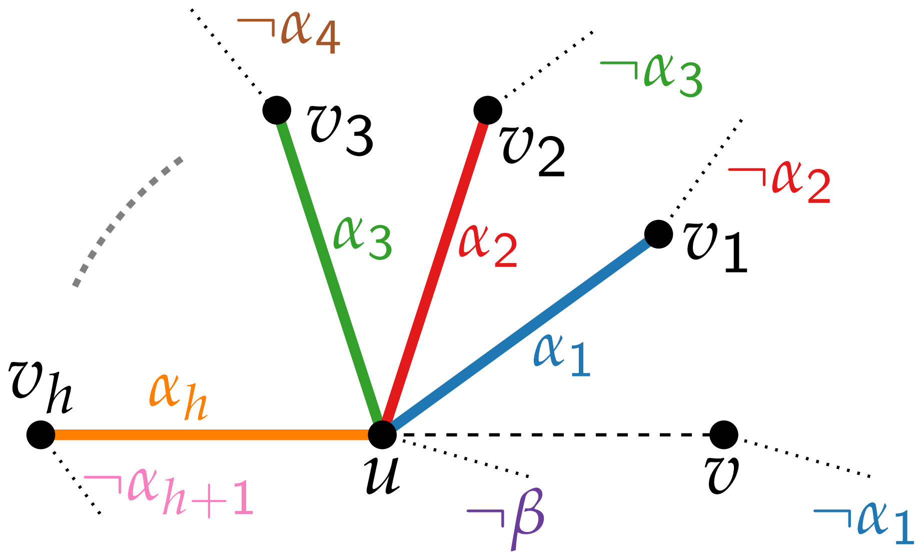
- $i \leftarrow 1$
- **while** $\exists w \in N(u)\colon\ c(\{u, w\}) = \alpha_i \ \wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  - $v_i \leftarrow w$
  - $\alpha_{i+1} \leftarrow$ min color missing at w
  - $i++$
- **return** $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

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

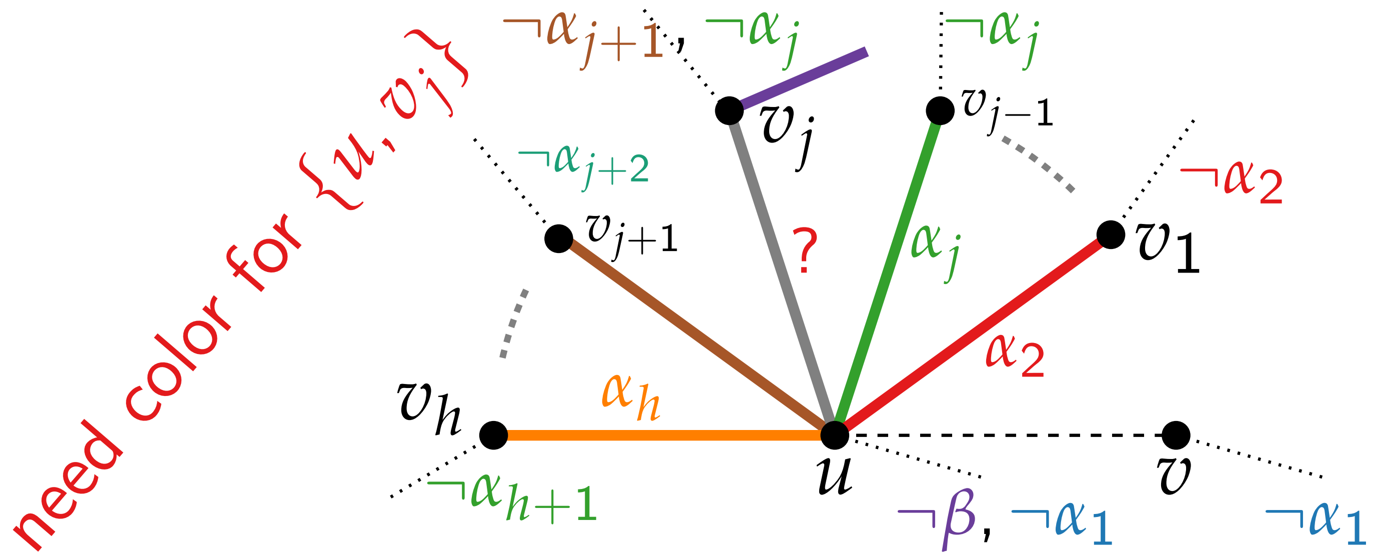


Proof. Note, each vertex is **missing** a color.
Let u miss β and v miss α_1; apply the following algorithm:

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

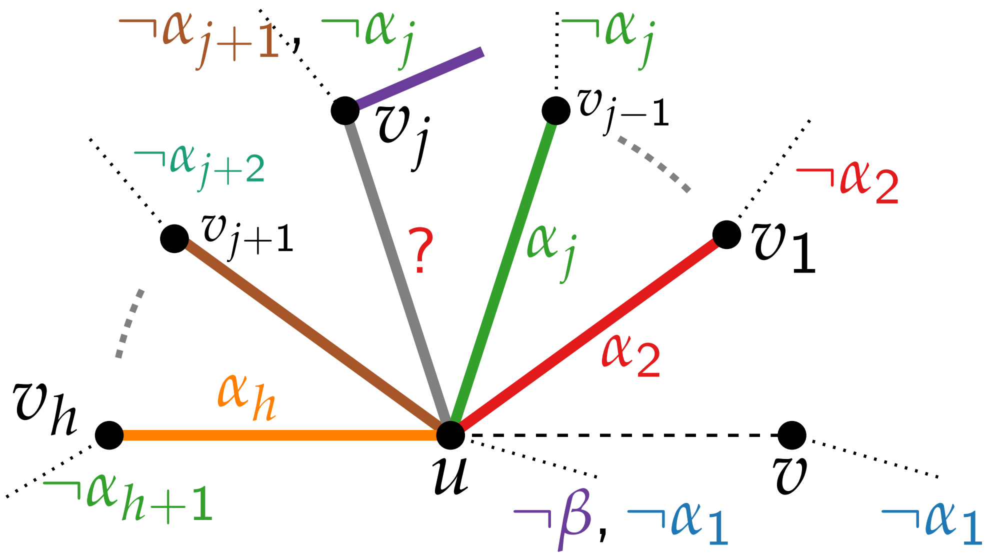
$i \leftarrow 1$
while $\exists w \in N(u)$: $c(\{u,w\}) = \alpha_i \ \wedge$ $w \notin \{v_1, \dots, v_{i-1}\}$ **do**
  $v_i \leftarrow w$
  $\alpha_{i+1} \leftarrow$ min color missing at w
  $i{+}{+}$
return $v_1, \dots, v_i; \alpha_1, \dots, \alpha_{i+1}$

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



Minimum edge coloring – recoloring

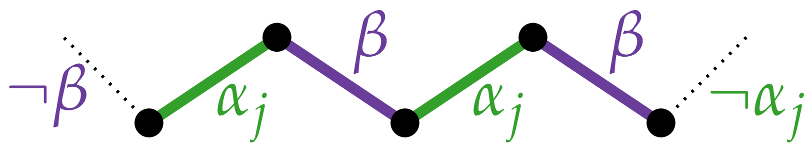
Proof continued for
Case 2: $\alpha_{h+1} = \alpha_j$, $j < h$ and
we need to find a color for $\{u, v_j\}$.

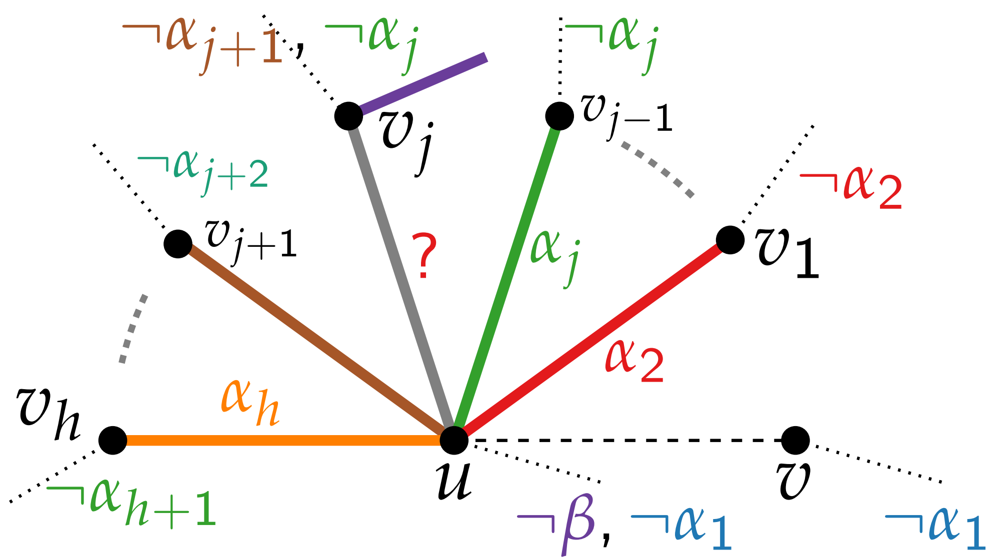


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.

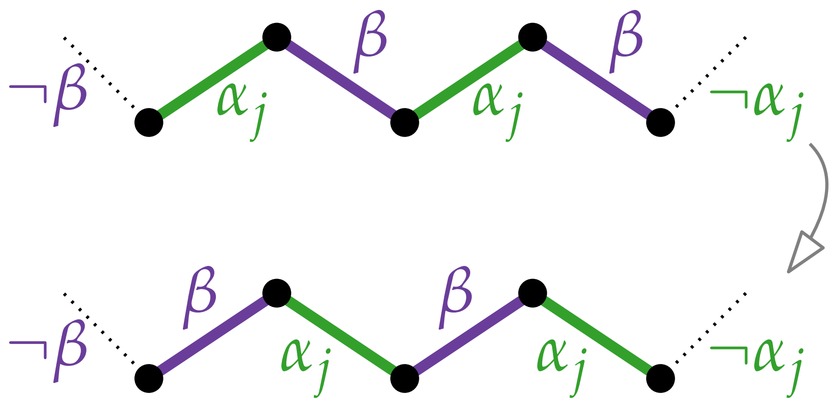


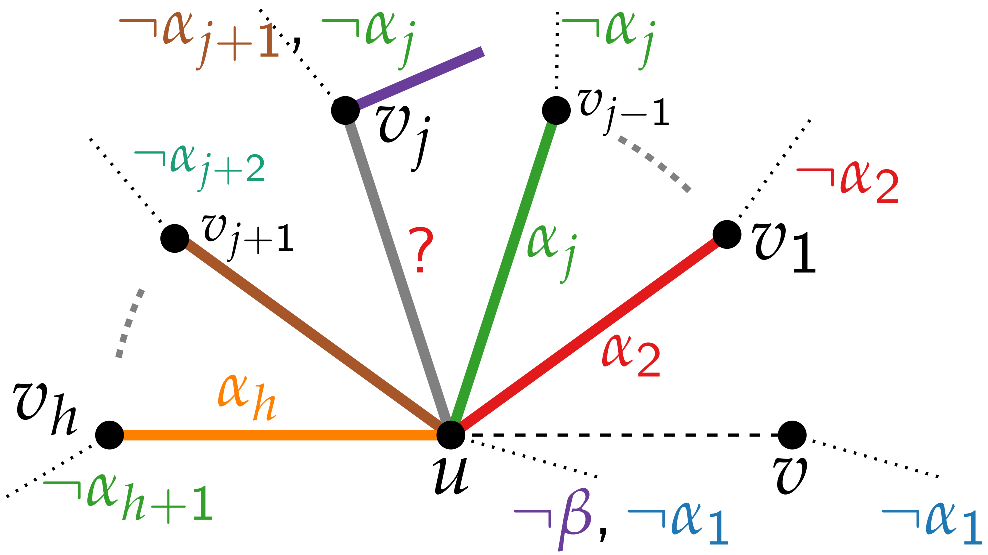


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.

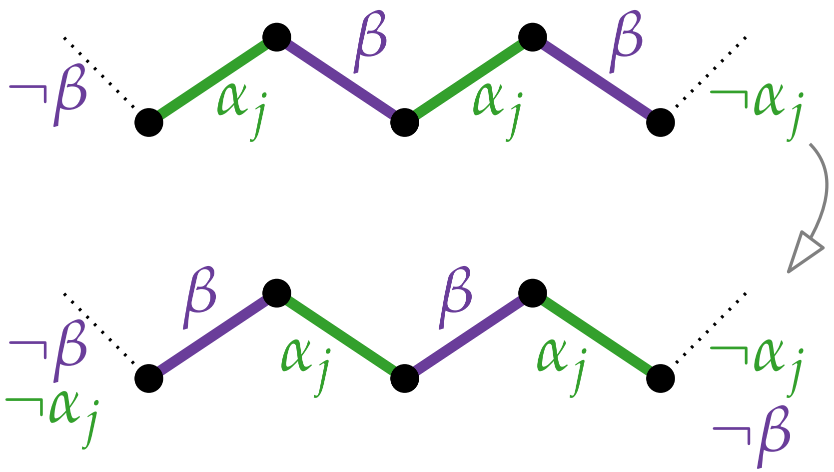


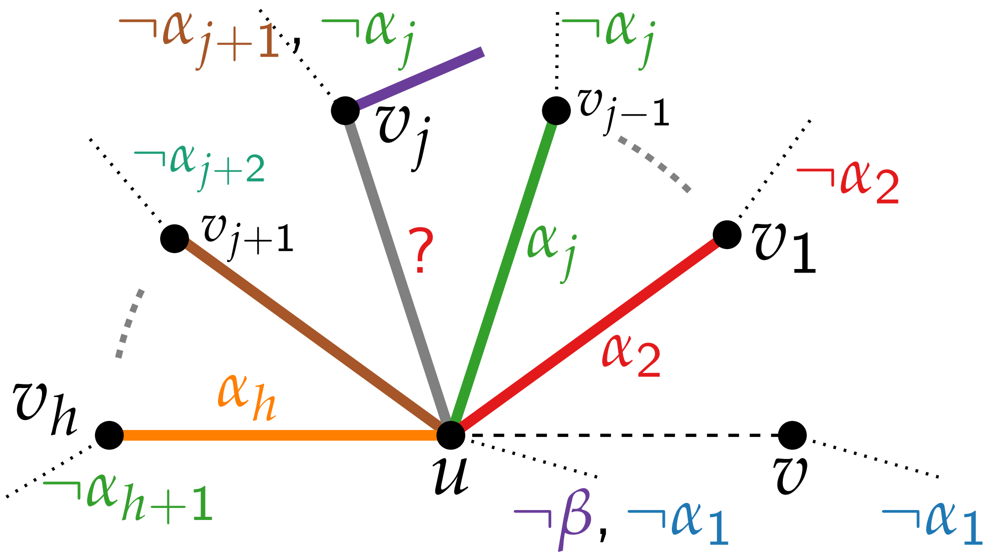


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.

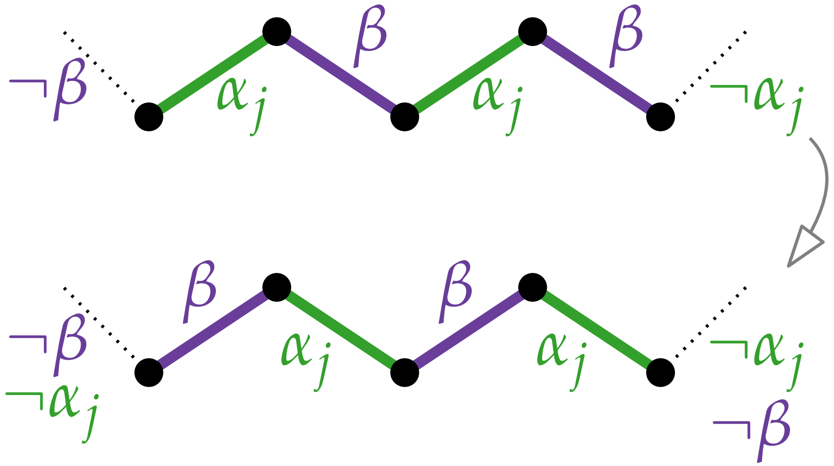


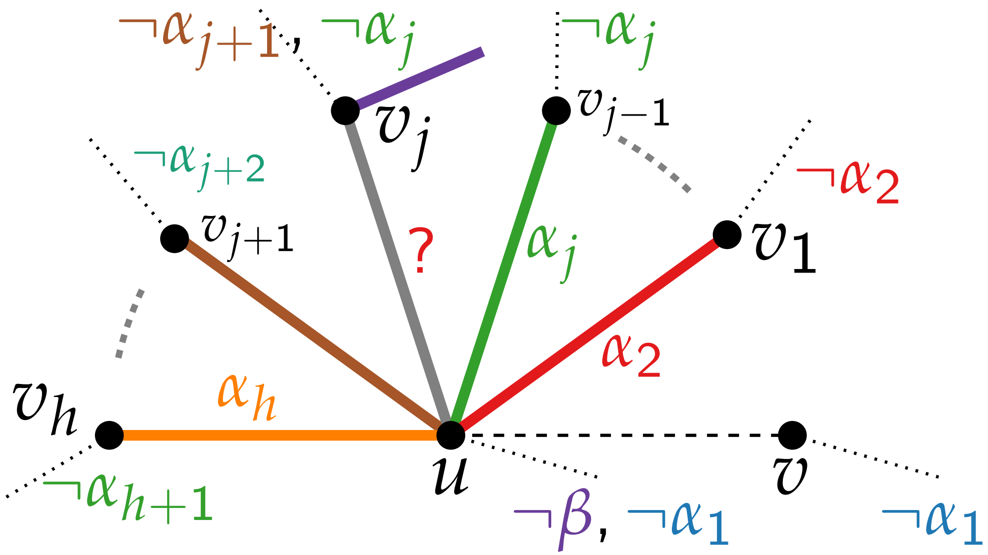


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- u, v_j, v_h have degree 1 in G'
  $\Rightarrow$ they are not all in same component

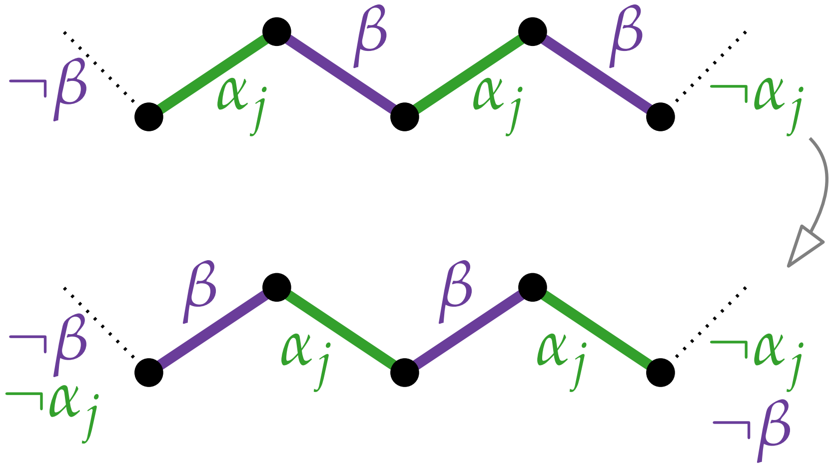


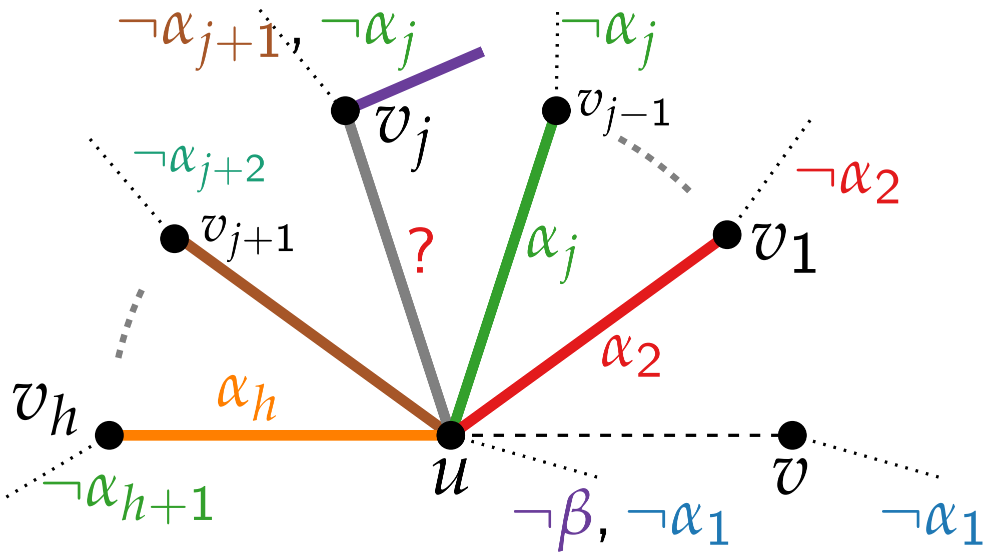


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- u, v_j, v_h have degree 1 in G'
  $\Rightarrow$ they are not all in same component
- If v_j and u are not in the same component:
  - Recolor component ending at v_j

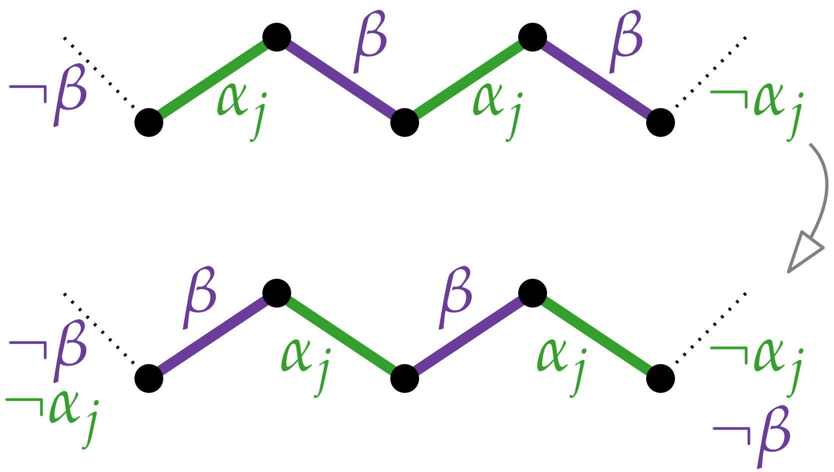


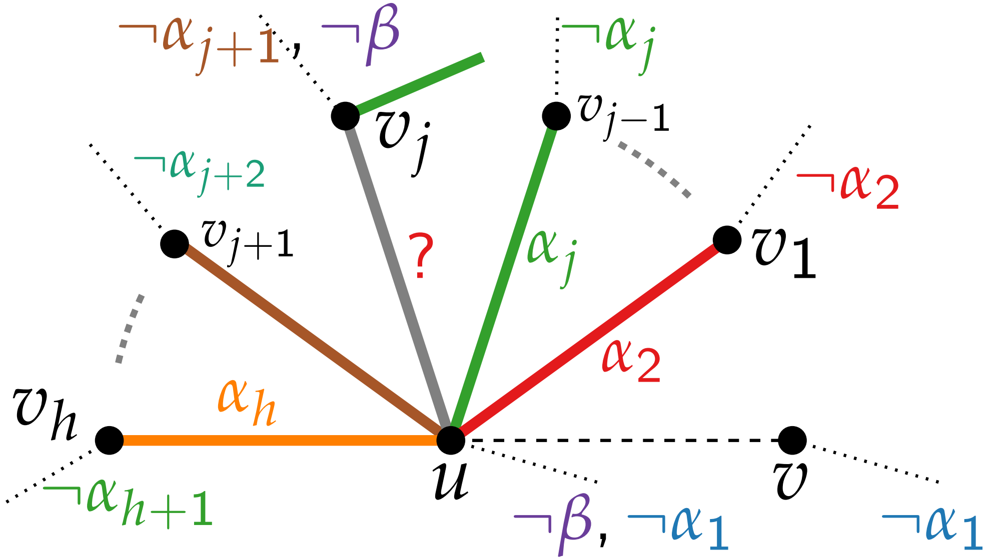


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- u, v_j, v_h have degree 1 in G'
  $\Rightarrow$ they are not all in same component
- If v_j and u are not in the same component:
  - Recolor component ending at v_j
  - v_j now misses β

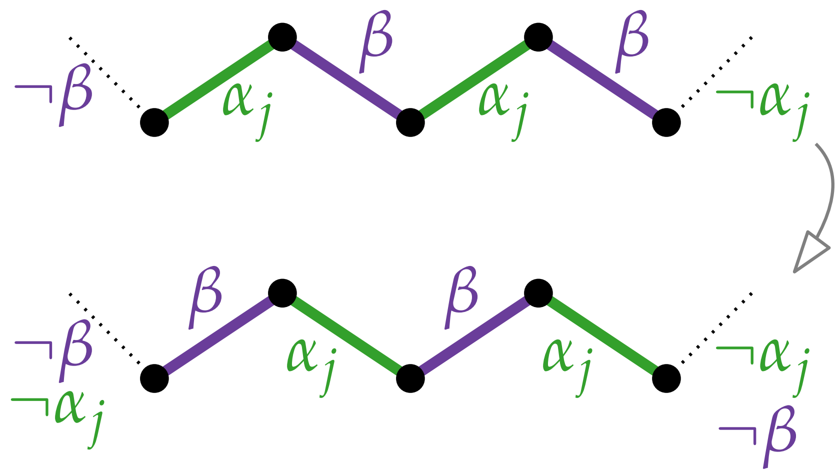


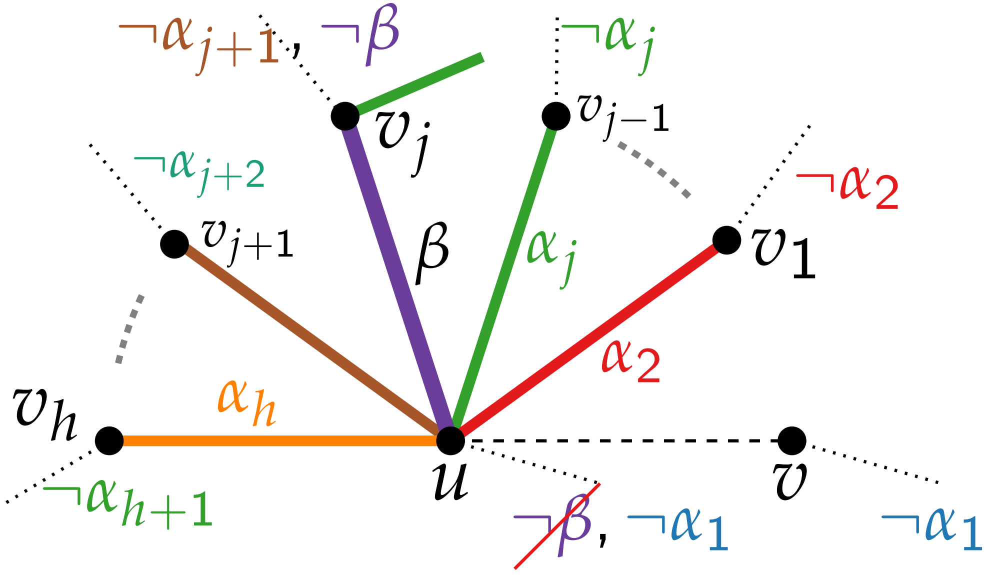


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- u, v_j, v_h have degree 1 in G'
  $\Rightarrow$ they are not all in same component
- If v_j and u are not in the same component:
  - Recolor component ending at v_j
  - v_j now misses β
  - Color $\{u, v_j\}$ in β

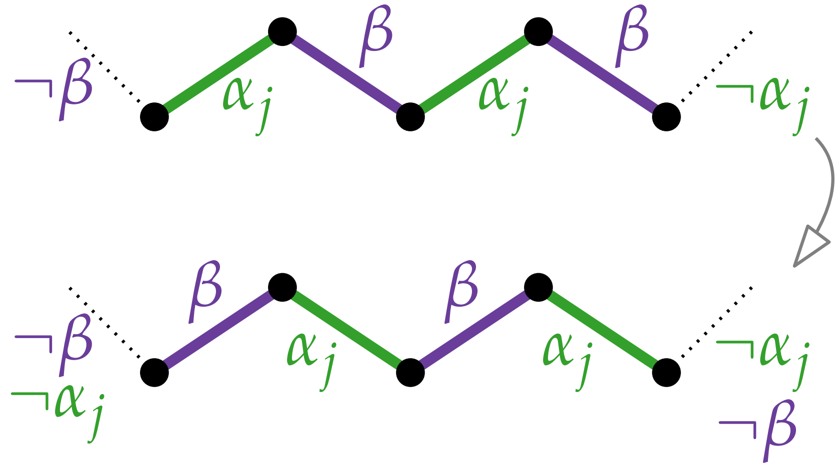


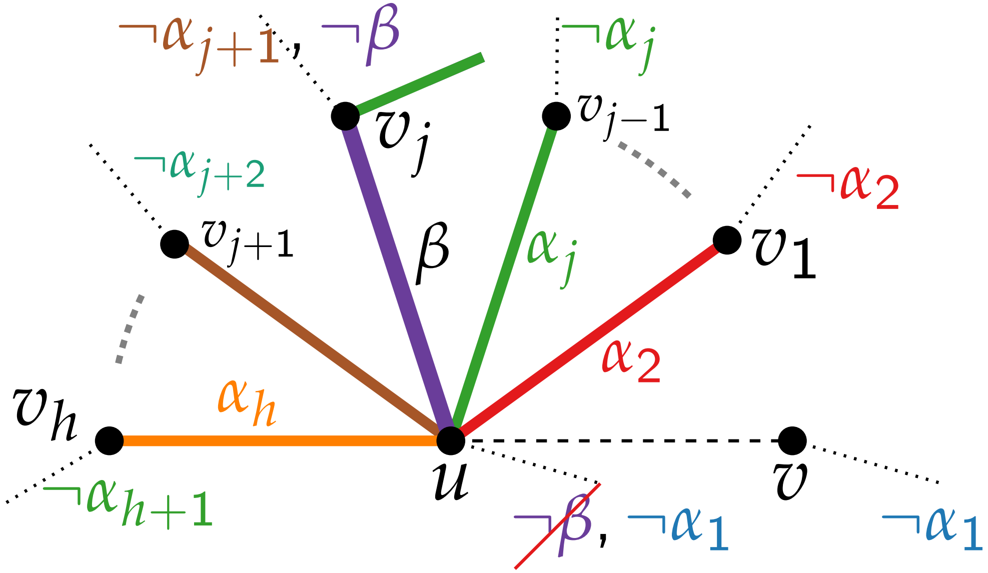


Minimum edge coloring – recoloring

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

- Consider subgraph G' of G induced by edges with color β and α_j.
- Since $\Delta(G') \leq 2$, we can recolor components.
- u, v_j, v_h have degree 1 in G'
  $\Rightarrow$ they are not all in same component
- If v_j and u are not in the same component:
  - Recolor component ending at v_j
  - v_j now misses β
  - Color $\{u, v_j\}$ in β
- What if v_j and u are in the same component?





Minimum edge coloring - algorithm

```
VizingEdgeColoring(G = (V, E))
  if E = ∅ then
    return 0
  else
    {u, v} ← random edge of G
    G' ← G − e
    VizingEdgeColoring(G')
    if Δ(G') < Δ(G) then
      Color {u, v} with lowest free color
    else
      Recolor E with Lemma 2
      Color {u, v} with color now missing at u and v
```

Minimum edge coloring - algorithm

VizingEdgeColoring($G = (V, E)$)

if $E = \emptyset$ **then**
- **return** 0

else
- $\{u, v\} \leftarrow$ random edge of G
- $G' \leftarrow G - e$
- VizingEdgeColoring(G')
- **if** $\Delta(G') < \Delta(G)$ **then**
  - Color $\{u, v\}$ with lowest free color
- **else**
  - Recolor E with Lemma 2
  - Color $\{u, v\}$ with color now missing at u and v

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

Approximation with relative factor

- An additive approximation guarantee can seldomly be achieved; but sometimes there is a multiplicative …

Approximation with relative factor

- An additive approximation guarantee can seldomly be achieved; but sometimes there is a multiplicative …

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

$$\frac{\mathcal{A}(I)}{\mathsf{OPT}(I)} \leq \alpha.$$

We call α the **approximation factor**.

Approximation with relative factor

- An additive approximation guarantee can seldomly be achieved; but sometimes there is a multiplicative …

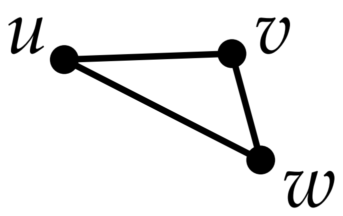
Definition.
Let Π be an minimisation (maximisation) problem and $\alpha \in \mathbb{Q}^+$.
A **(factor) α-approximation algorithm** for Π is a polynomial-time algorithm $\mathcal{A}$, which computes for every instance I of Π a value $\mathcal{A}(I)$ such that

$$\frac{\mathcal{A}(I)}{\mathrm{OPT}(I)} \leq \alpha. \quad (\geq)$$

We call α the **approximation factor**.

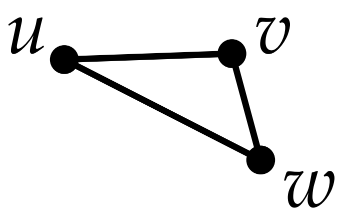
2-approximation for Metric TSP (from AGT)

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



2-approximation for Metric TSP (from AGT)

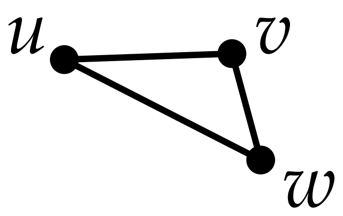
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

2-approximation for Metric TSP (from AGT)

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

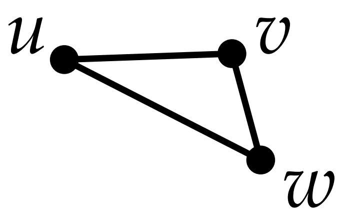


Output. Shortest Hamilton cycle.

Algorithm.

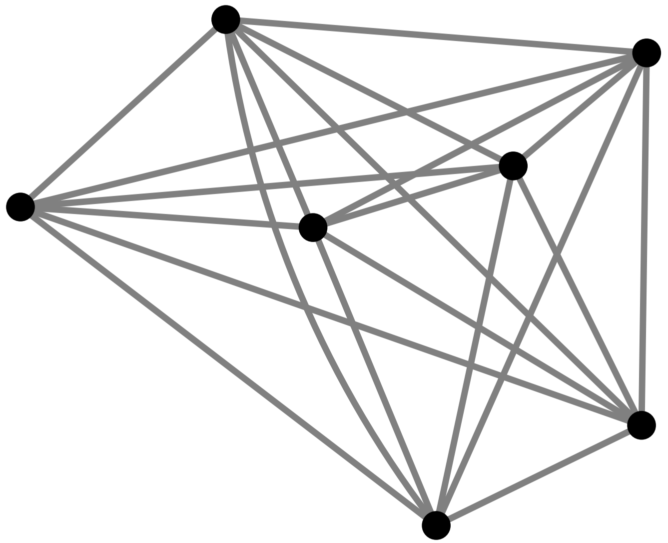
2-approximation for Metric TSP (from AGT)

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



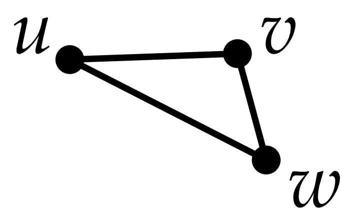
Output. Shortest Hamilton cycle.

Algorithm.



2-approximation for Metric TSP (from AGT)

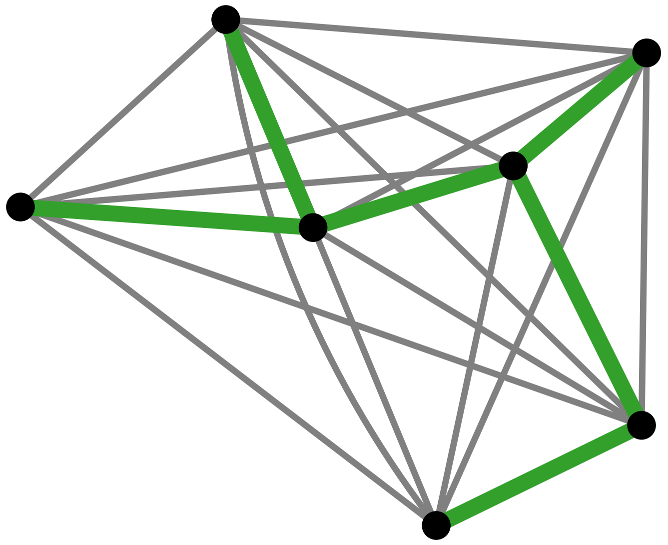
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

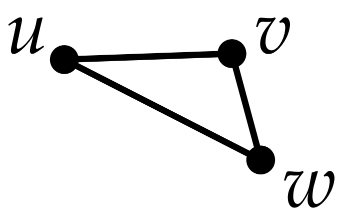
Algorithm.

- Compute MST.



2-approximation for Metric TSP (from AGT)

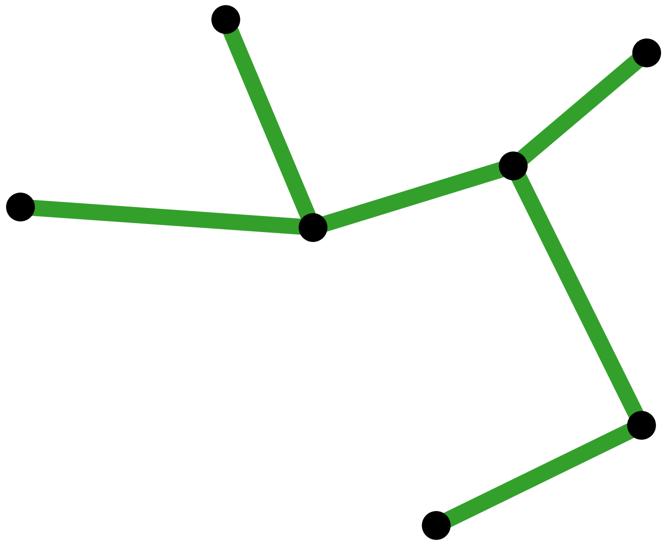
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

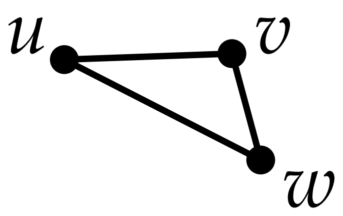
Algorithm.

- Compute MST.



2-approximation for Metric TSP (from AGT)

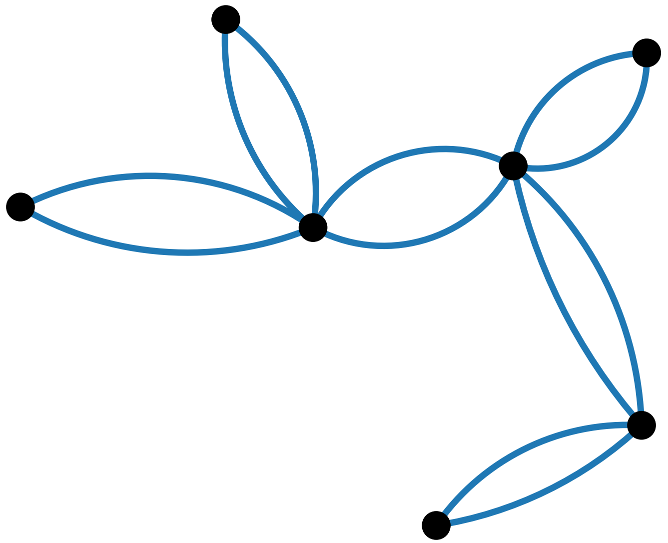
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

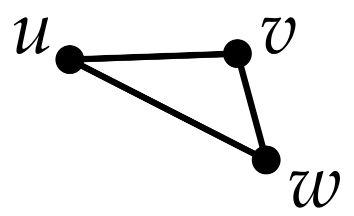
Algorithm.

- Compute MST.
- Double edges.



2-approximation for Metric TSP (from AGT)

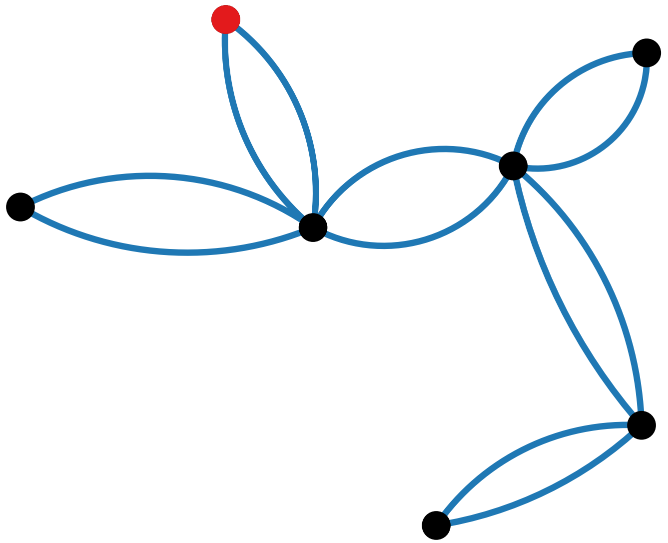
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

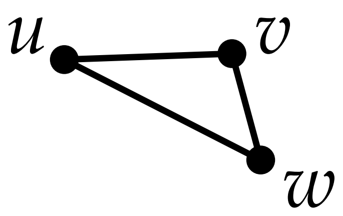
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,



2-approximation for Metric TSP (from AGT)

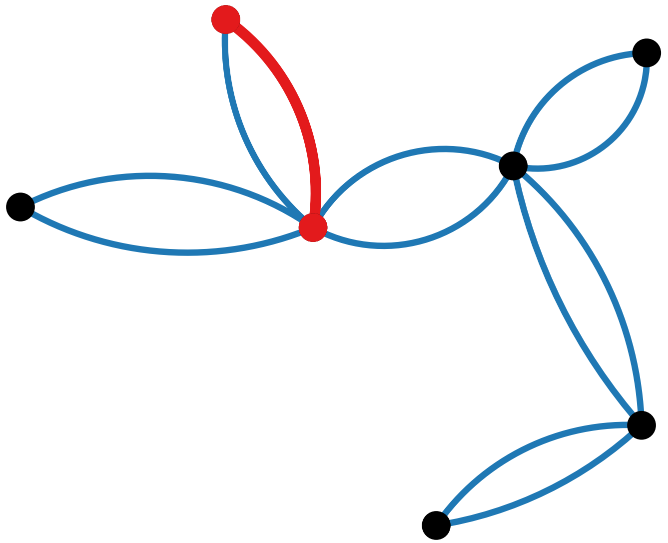
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

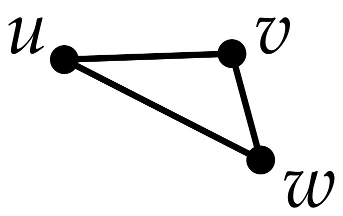
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,



2-approximation for Metric TSP (from AGT)

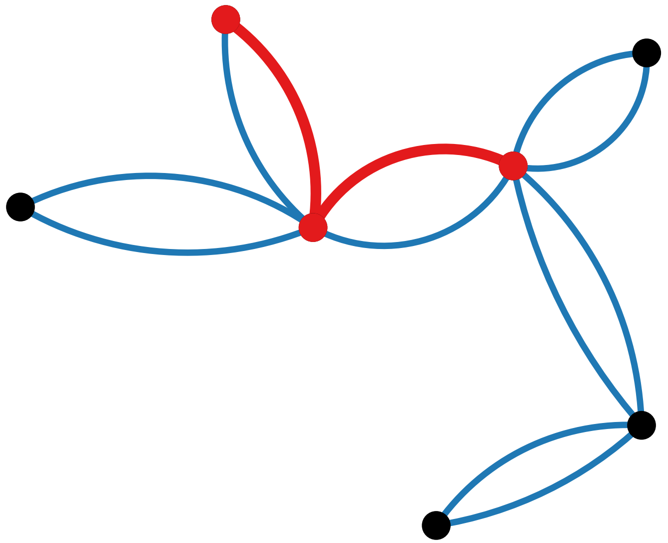
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

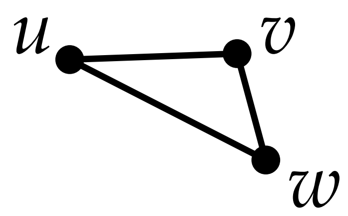
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,



2-approximation for Metric TSP (from AGT)

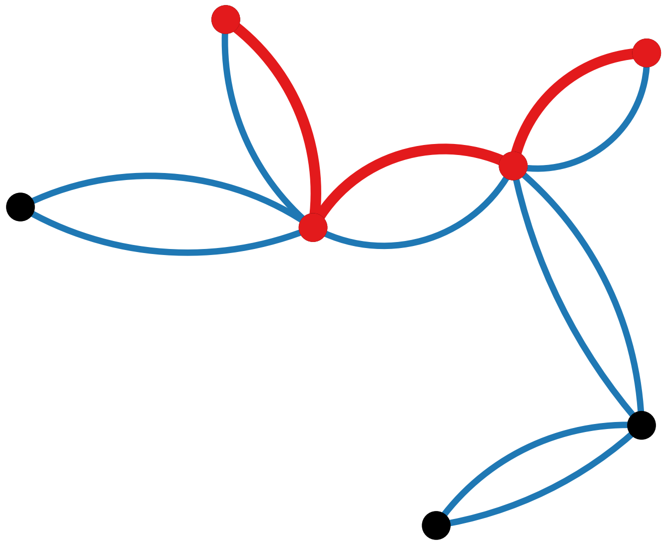
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

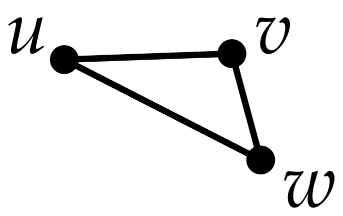
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,



2-approximation for Metric TSP (from AGT)

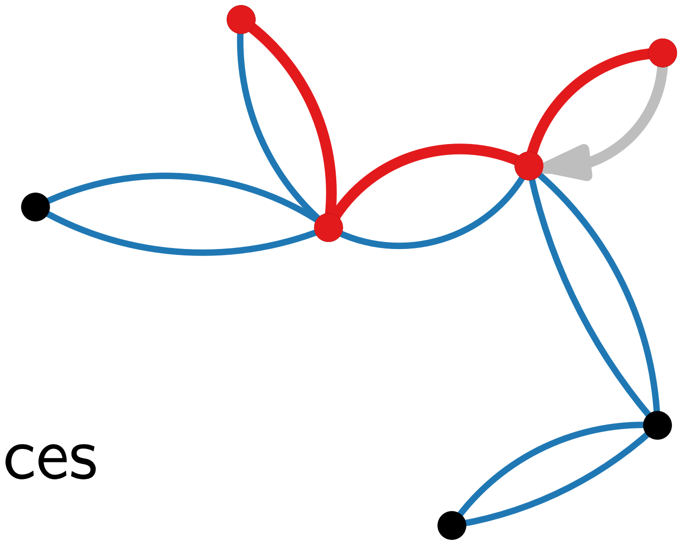
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

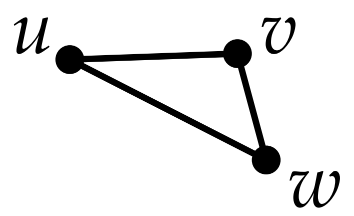
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,
- skipping visited vertices



2-approximation for Metric TSP (from AGT)

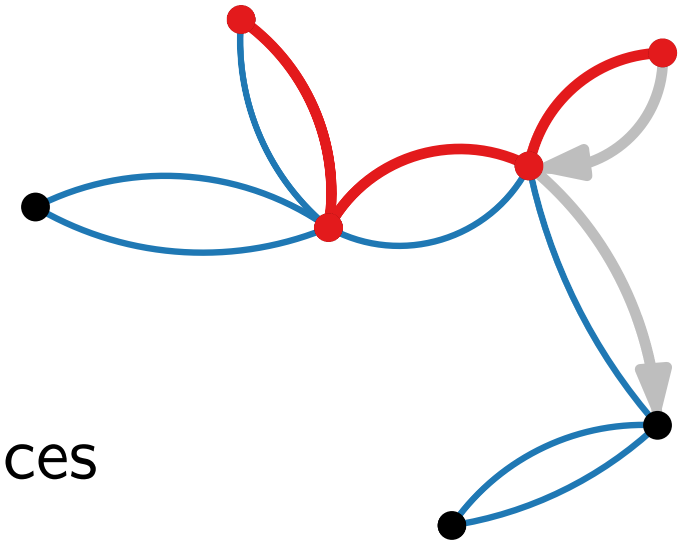
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u,w) \leq d(u,v) + d(v,w)$.



Output. Shortest Hamilton cycle.

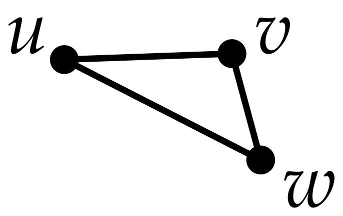
Algorithm.

- Compute MST.
- Double edges.
- Walk along tree,
- skipping visited vertices



2-approximation for Metric TSP (from AGT)

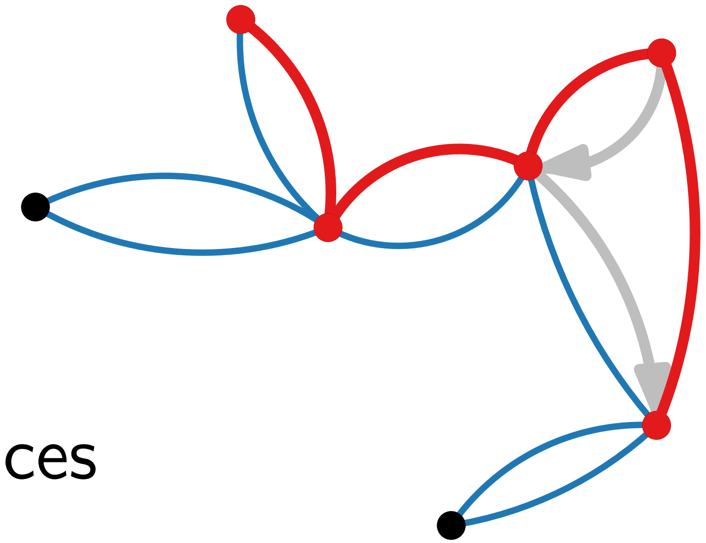
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

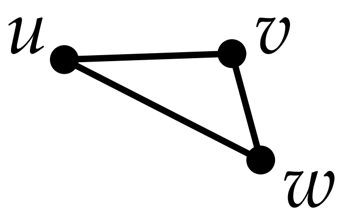
Algorithm.

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



2-approximation for Metric TSP (from AGT)

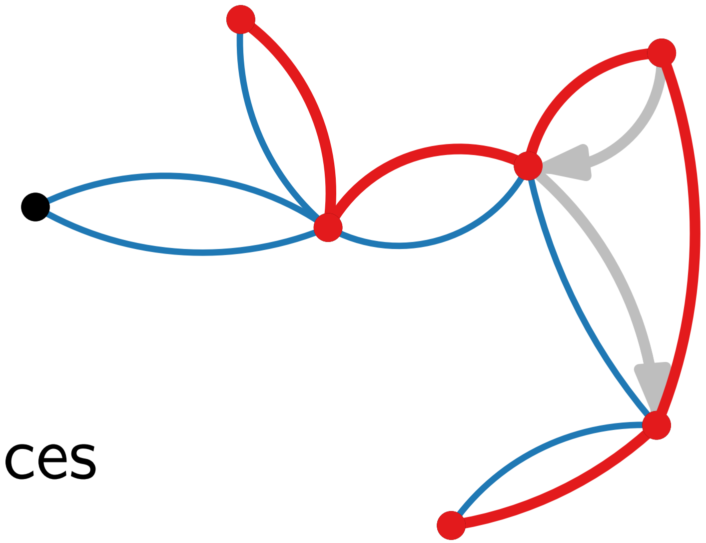
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

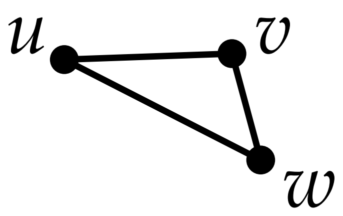
Algorithm.

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



2-approximation for Metric TSP (from AGT)

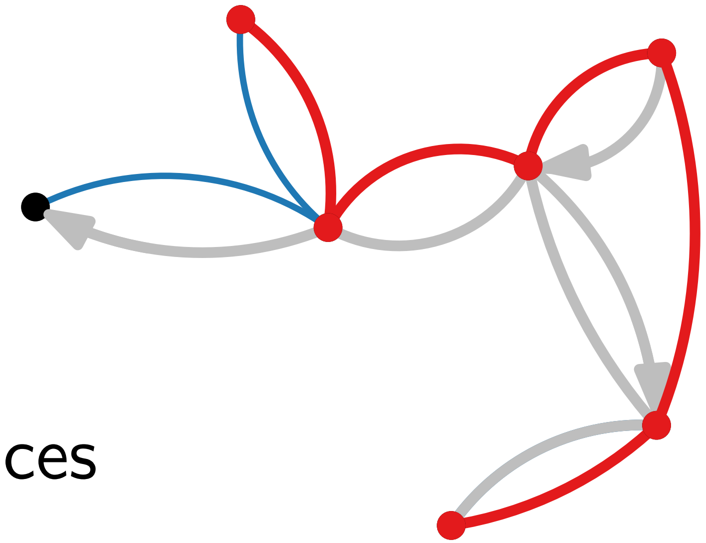
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

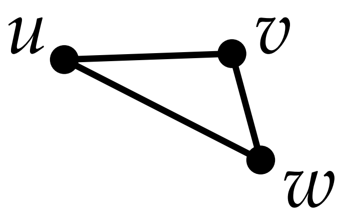
Algorithm.

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



2-approximation for Metric TSP (from AGT)

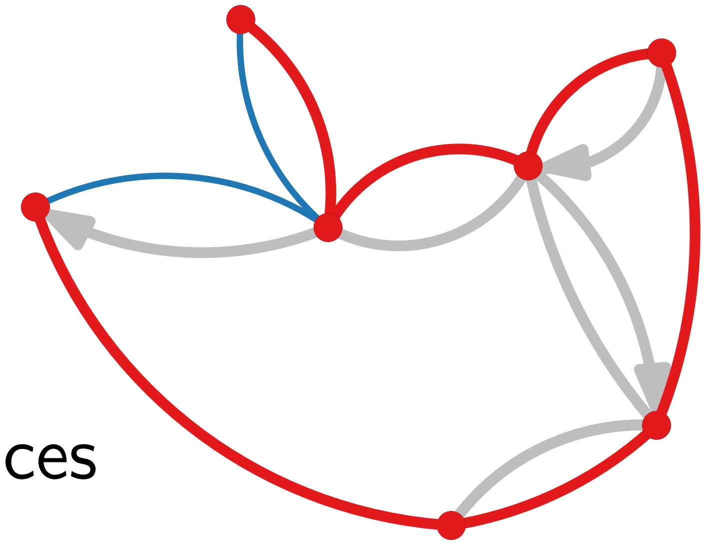
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

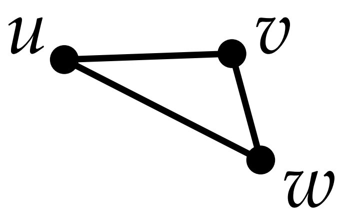
Algorithm.

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



2-approximation for Metric TSP (from AGT)

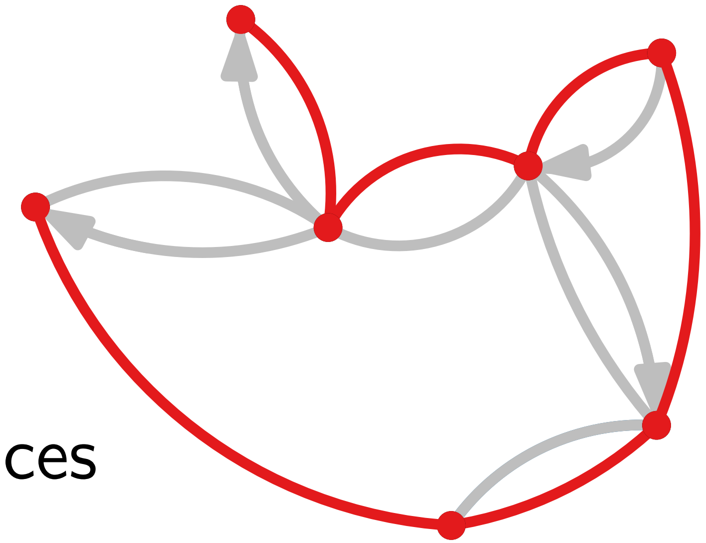
Input. Complete graph $G = (V, E)$ and distance function $d : E \to \mathbb{R}_{\geq 0}$, which satisfies the triangle inequality, i.e. $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$.



Output. Shortest Hamilton cycle.

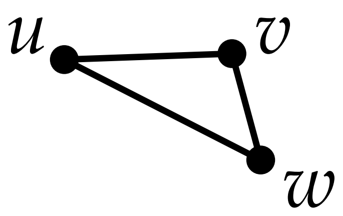
Algorithm.

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



2-approximation for Metric TSP (from AGT)

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



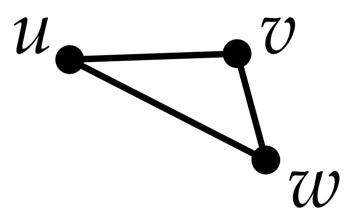
Output. Shortest Hamilton cycle.

Algorithm.

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

2-approximation for Metric TSP (from AGT)

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



Output. Shortest Hamilton cycle.

Algorithm.

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

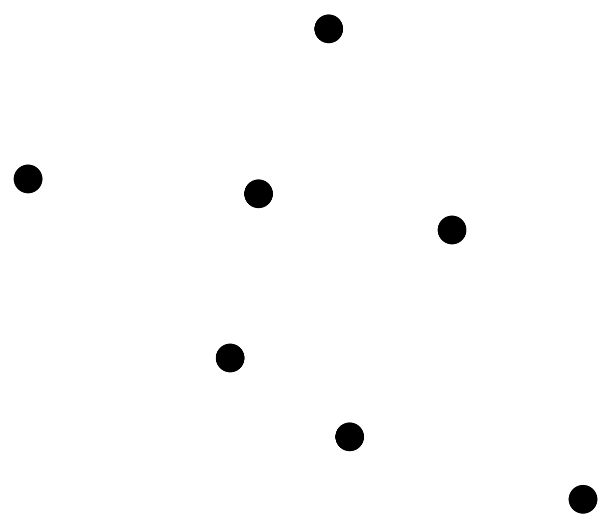
Theorem 5.
The MST edge doubling algorithm is a 2-approximation algorithm for metric TSP.

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

Nearest addition algorithm for Metric TSP

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

- Find closest pair, say i and j
- Set tour T to go from i to j to i
- **for** $n - 2$ iterations **do**
  - Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
  - Let k be vertex after i in T
  - Add j between i and k



Nearest addition algorithm for Metric TSP

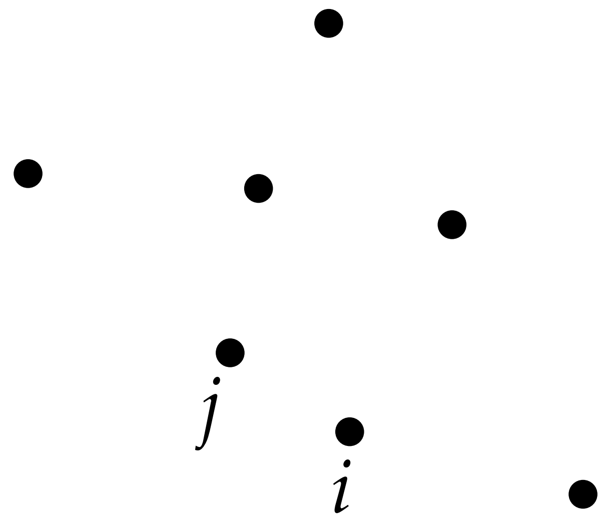
NEARESTADDITIONALGORITHM$(G = (V, E), d)$

Find closest pair, say i and j

Set tour T to go from i to j to i

for $n - 2$ iterations **do**

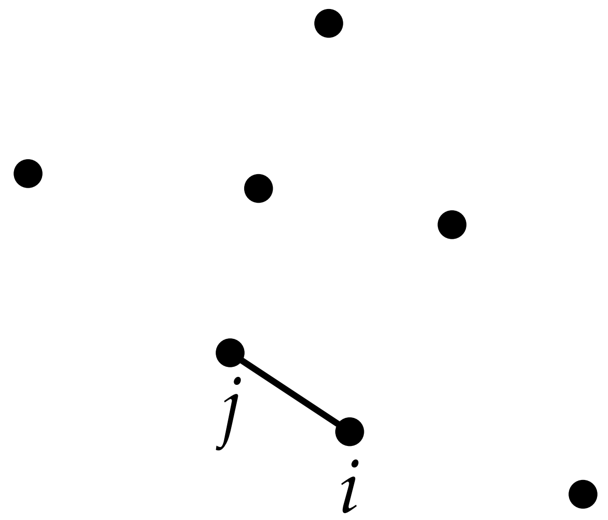
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

$\textsc{NearestAdditionAlgorithm}(G = (V, E),\, d)$

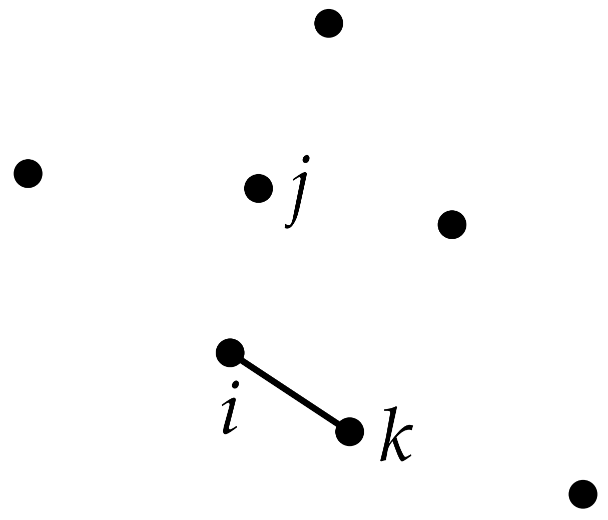
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

$\textsc{NearestAdditionAlgorithm}(G = (V, E), d)$

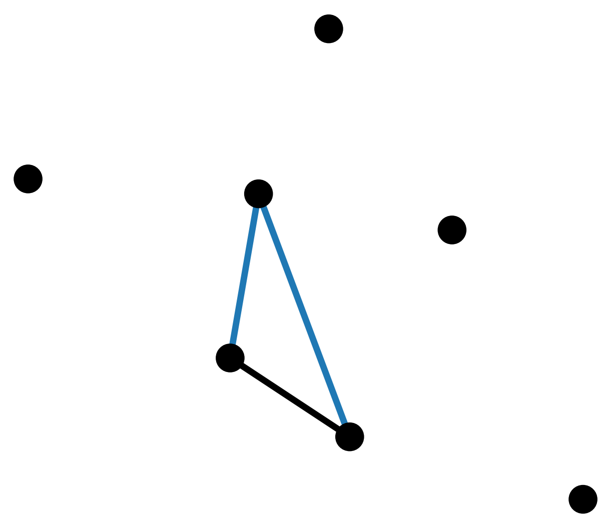
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

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

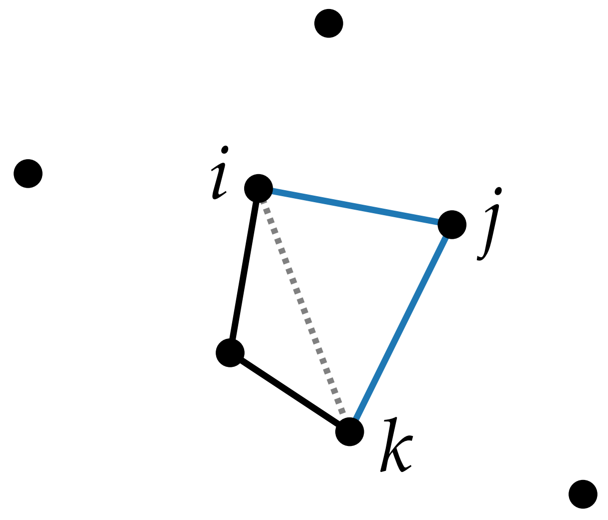
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
 Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
 Let k be vertex after i in T
 Add j between i and k



Nearest addition algorithm for Metric TSP

$\textsc{NearestAdditionAlgorithm}(G = (V, E), d)$

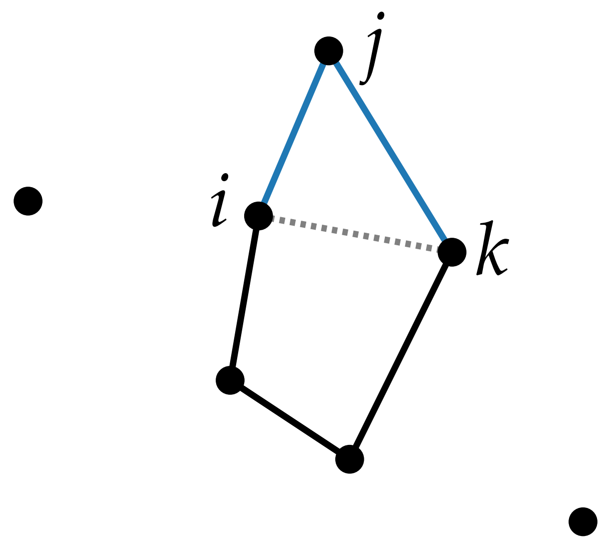
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

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

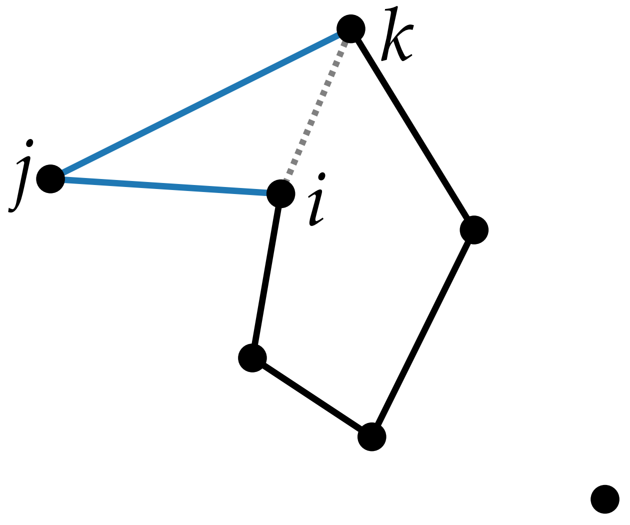
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

$\textsc{NearestAdditionAlgorithm}(G = (V, E), d)$

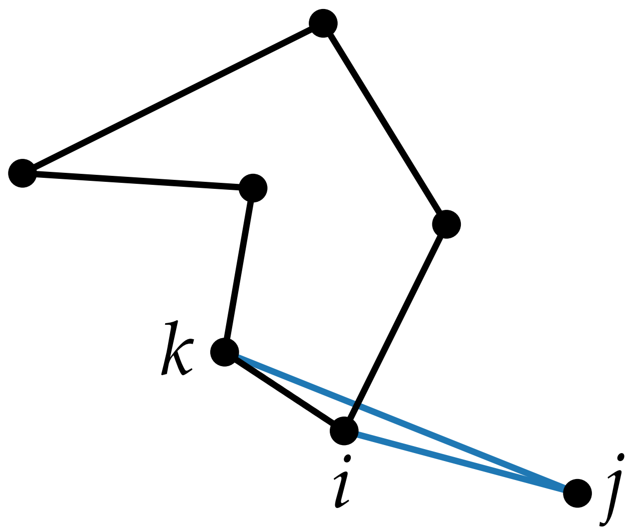
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

$\textsc{NearestAdditionAlgorithm}(G = (V, E), d)$

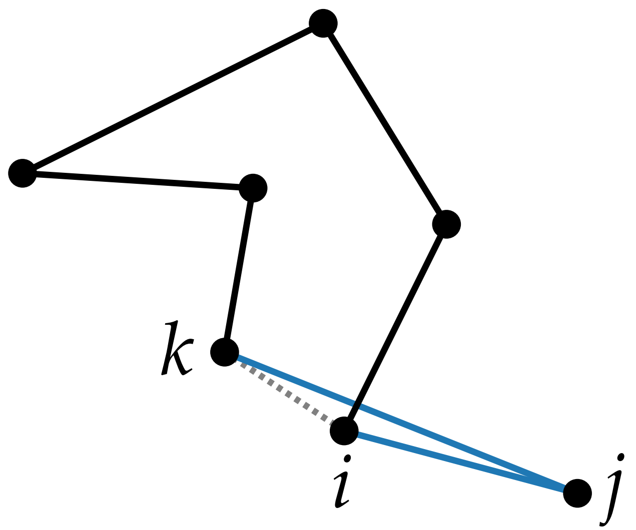
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Nearest addition algorithm for Metric TSP

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

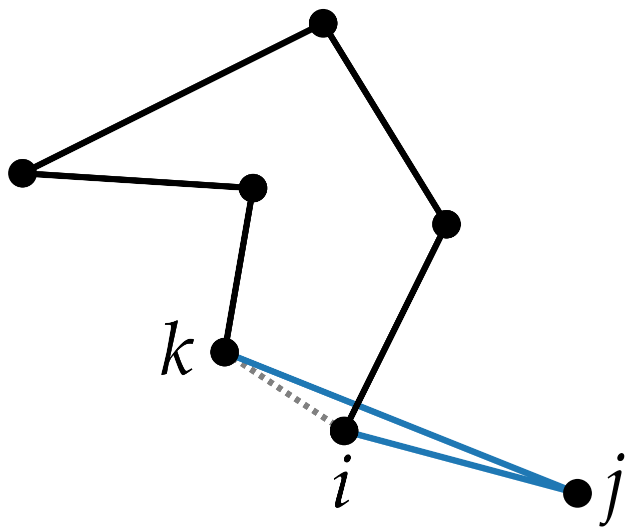
Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
    Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
    Let k be vertex after i in T
    Add j between i and k



Nearest addition algorithm for Metric TSP

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

Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
  Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
  Let k be vertex after i in T
  Add j between i and k

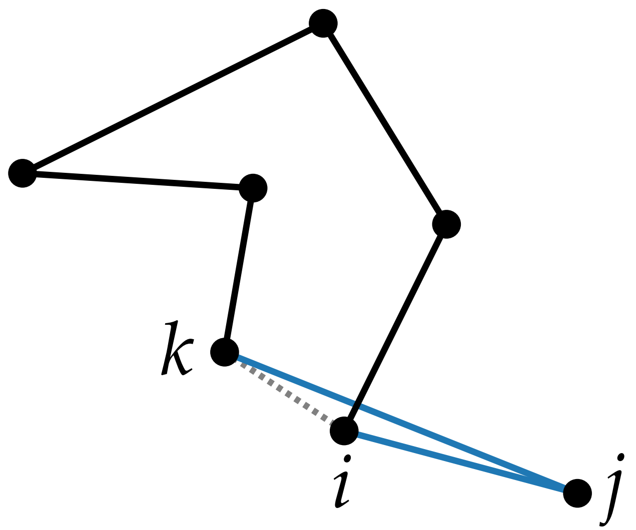


Theorem 6.
The NearestAdditionAlgorithm is a 2-approximation algorithm for metric TSP.

Nearest addition algorithm for Metric TSP

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

Find closest pair, say i and j
Set tour T to go from i to j to i
for $n - 2$ iterations **do**
- Find pair $i \in T$ and $j \notin T$ with $\min d(i, j)$
- Let k be vertex after i in T
- Add j between i and k



Theorem 6.
The 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 minimisation problem. An algorithm $\mathcal{A}$ is called an **polynomial-time approximation scheme (PTAS)**, if $\mathcal{A}$ computes for every input (I, ε) consisting of an instance I of Π and $\varepsilon > 0$ a value $\mathcal{A}(I)$, such that:

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

Approximation schemes

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

Definition.

maximisation

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

$\geq (1 - \varepsilon)$

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

Approximation schemes

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

Definition.

maximisation

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

$\geq (1-\varepsilon)$

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

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

Approximation schemes

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

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

$\geq (1 - \varepsilon)$

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

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

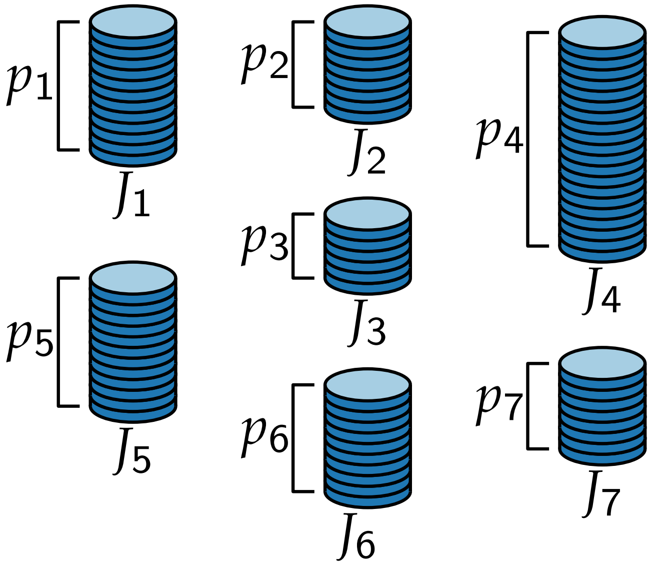
Examples.

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

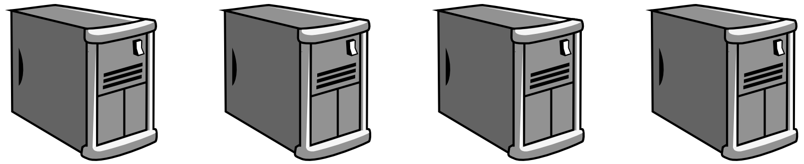
Multiprocessor Scheduling

Input.

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



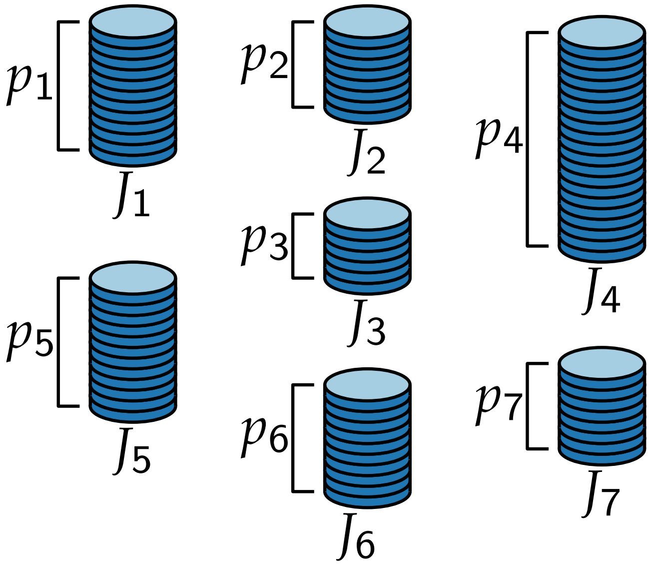
- m identical machines ($m < n$)



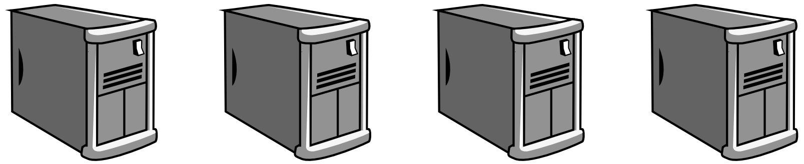
Multiprocessor Scheduling

Input.

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



- m identical machines ($m < n$)

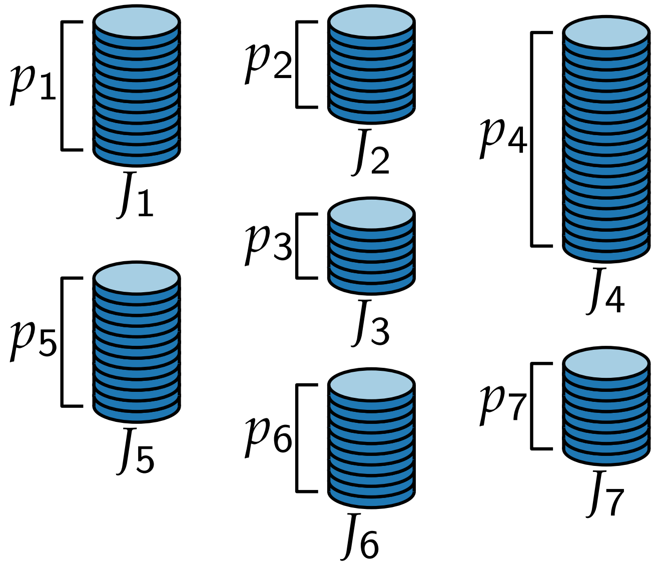


Output. Distribution of jobs to machines such that the time when all jobs have been processed is minimal.
This is called the **makespan** of the distribution.

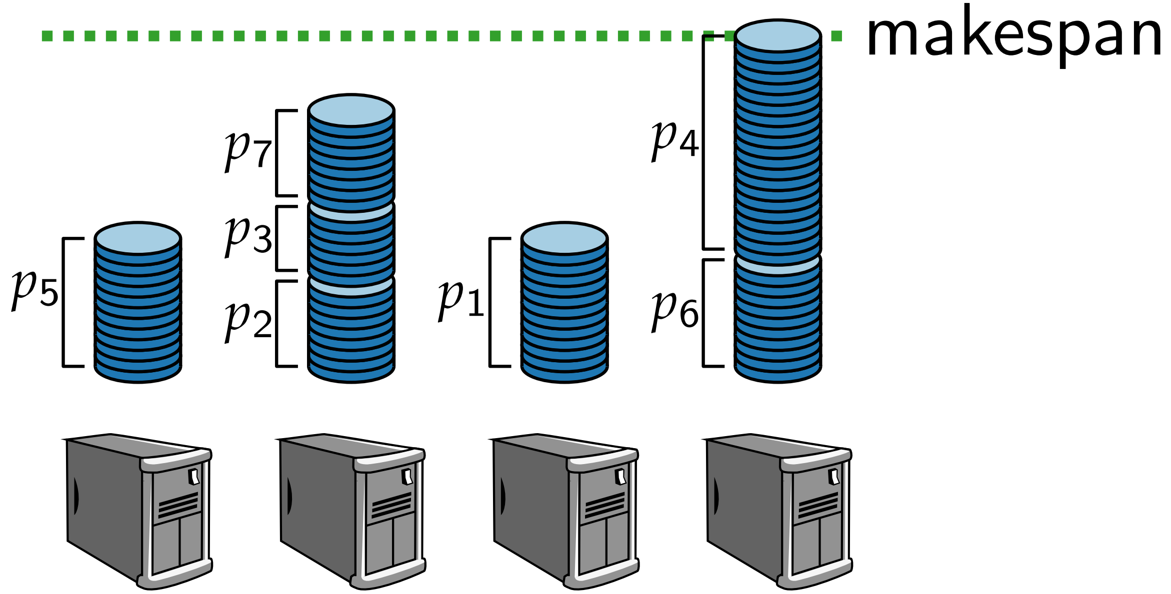
Multiprocessor Scheduling

Input.

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



- m identical machines ($m < n$)

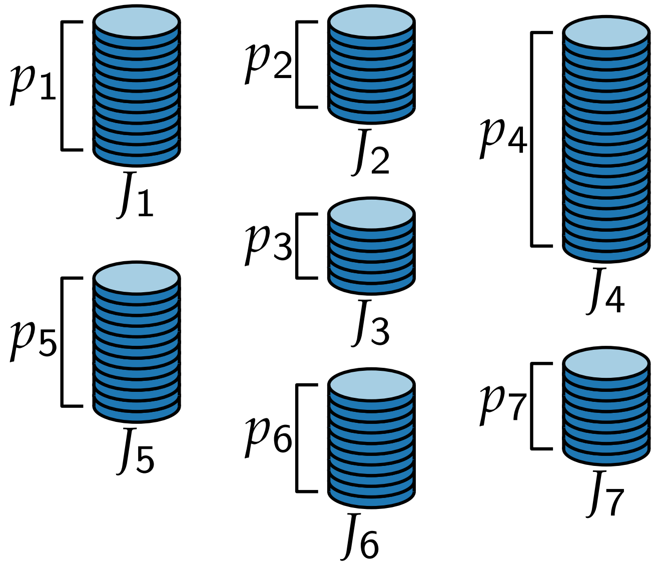


Output. Distribution of jobs to machines such that the time when all jobs have been processed is minimal.
This is called the **makespan** of the distribution.

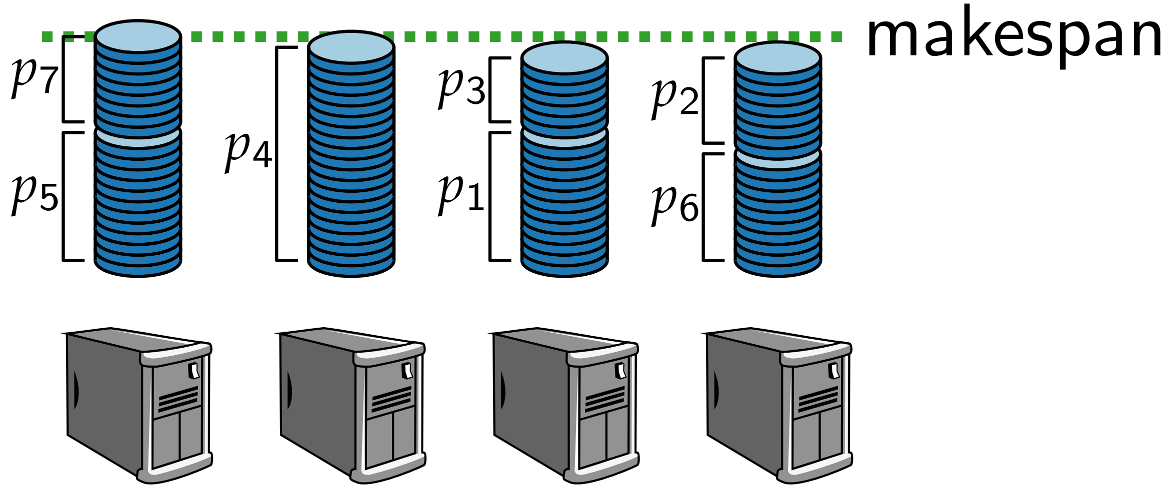
Multiprocessor Scheduling

Input.

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



- m identical machines ($m < n$)

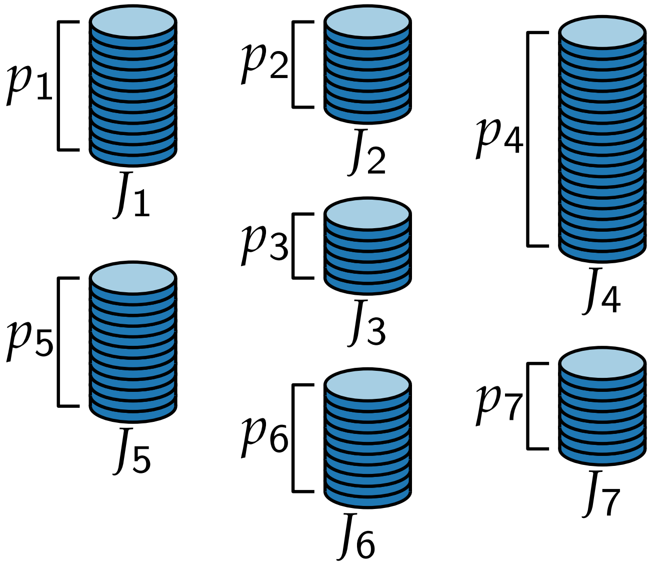


Output. Distribution of jobs to machines such that the time when all jobs have been processed is minimal.
This is called the **makespan** of the distribution.

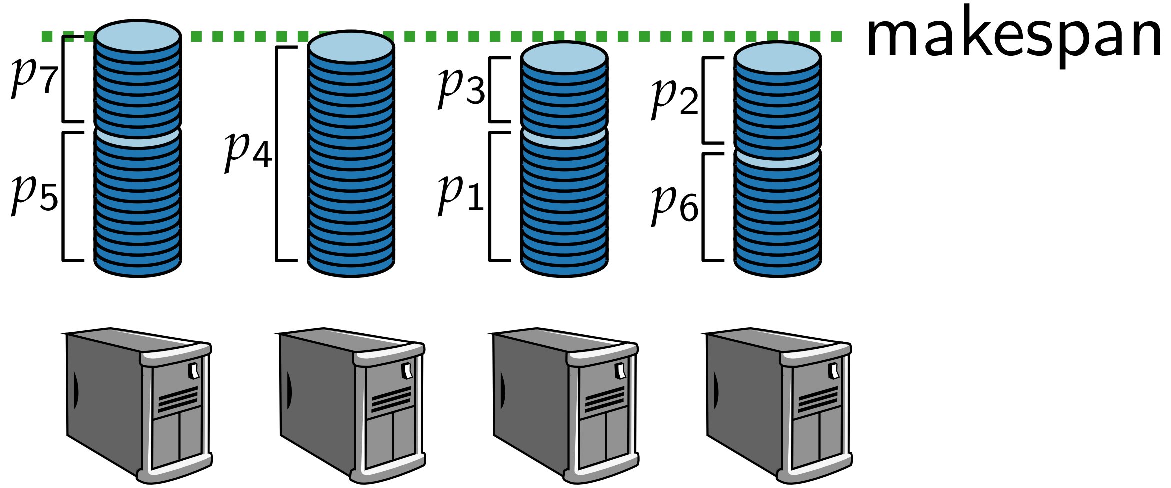
Multiprocessor Scheduling

Input.

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



- m identical machines ($m < n$)



Output. Distribution of jobs to machines such that the time when all jobs have been processed is minimal.
This is called the **makespan** of the distribution.

- Multiprocess scheduling is NP-hard.

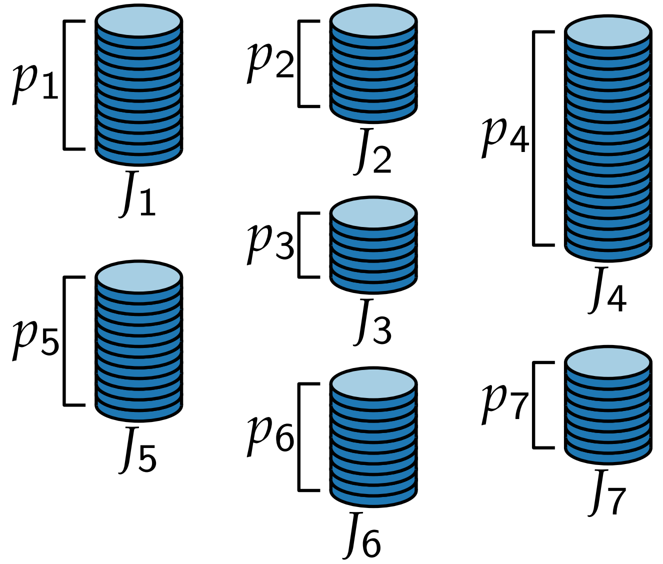
Multiprocessor Scheduling – List scheduling

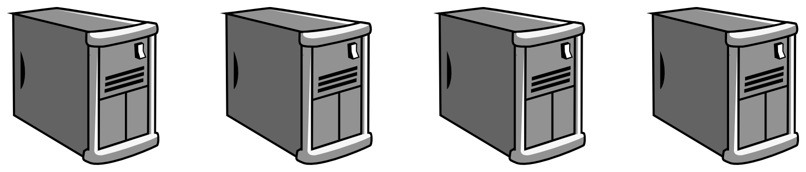
$\text{ListScheduling}(J_1, \ldots, J_n, m)$

Put the first m jobs on the m machines

Put next job on first free machine

Example.





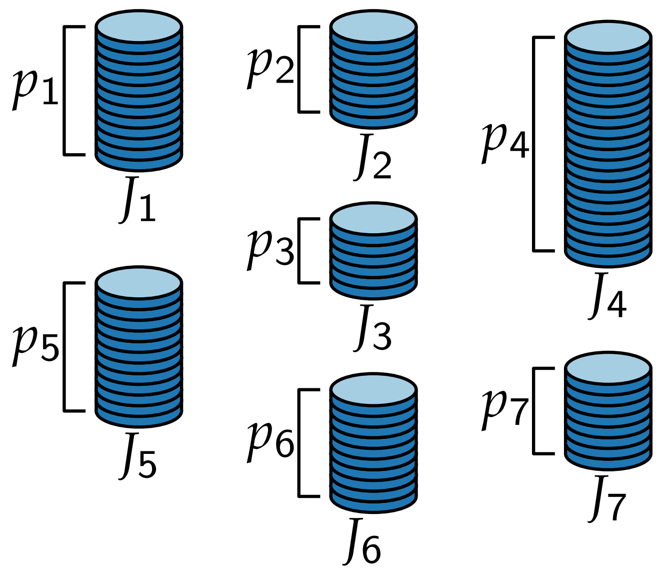
Multiprocessor Scheduling – List scheduling

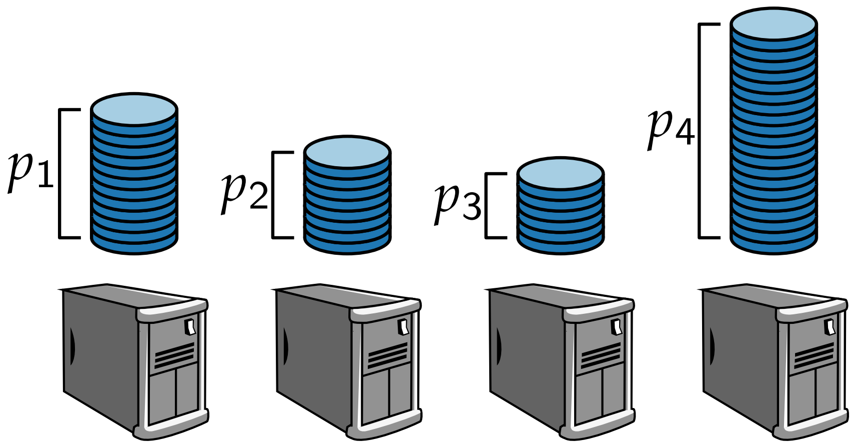
$\textsc{ListScheduling}(J_1, \ldots, J_n, m)$

Put the first m jobs on the m machines

Put next job on first free machine

Example.





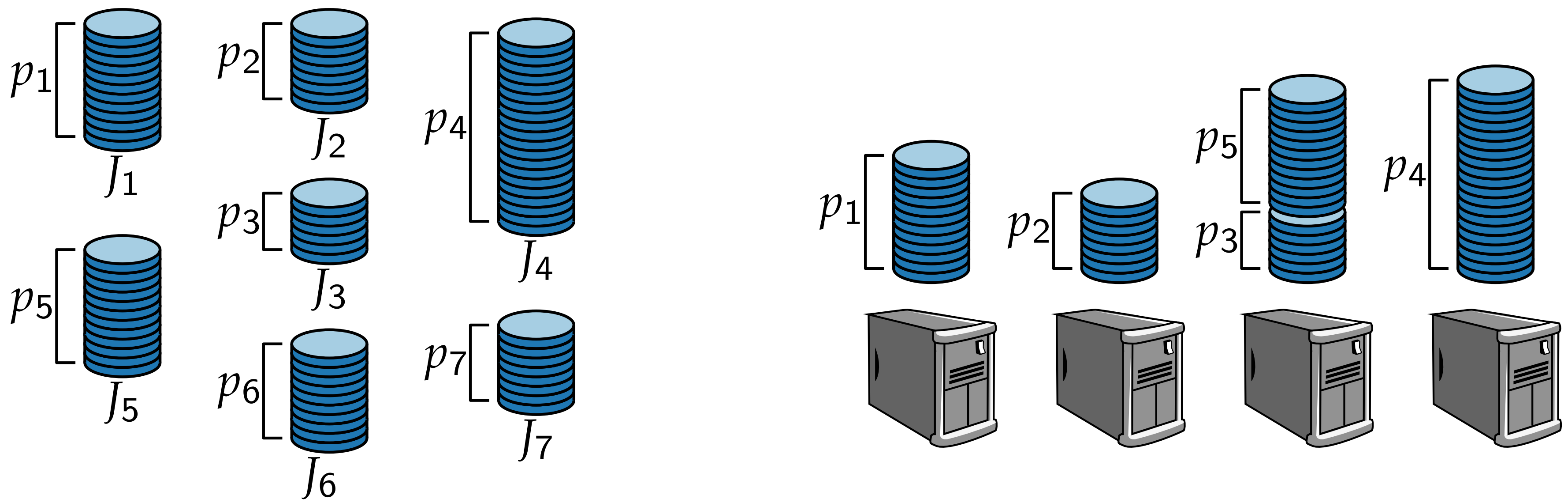
Multiprocessor Scheduling – List scheduling

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

Put the first m jobs on the m machines

Put next job on first free machine

Example.



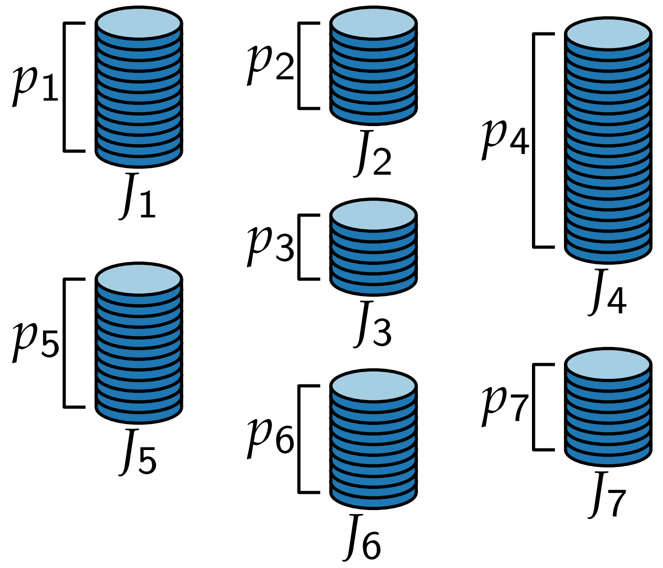
Multiprocessor Scheduling – List scheduling

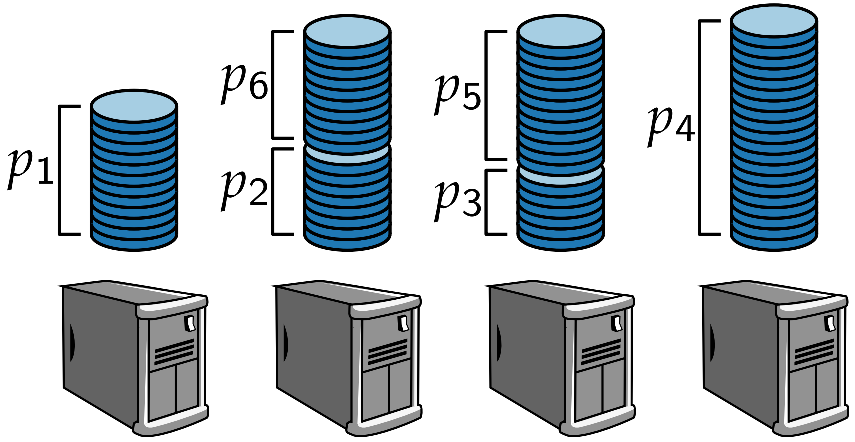
$\textsc{ListScheduling}(J_1, \ldots, J_n, m)$

Put the first m jobs on the m machines

Put next job on first free machine

Example.





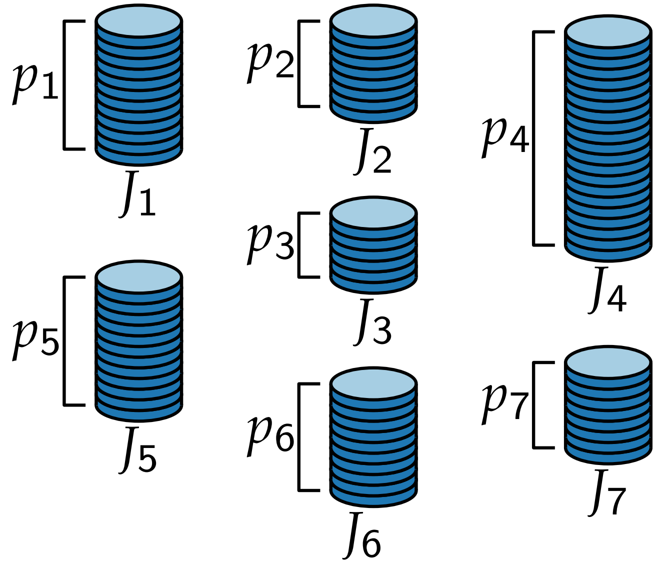
Multiprocessor Scheduling – List scheduling

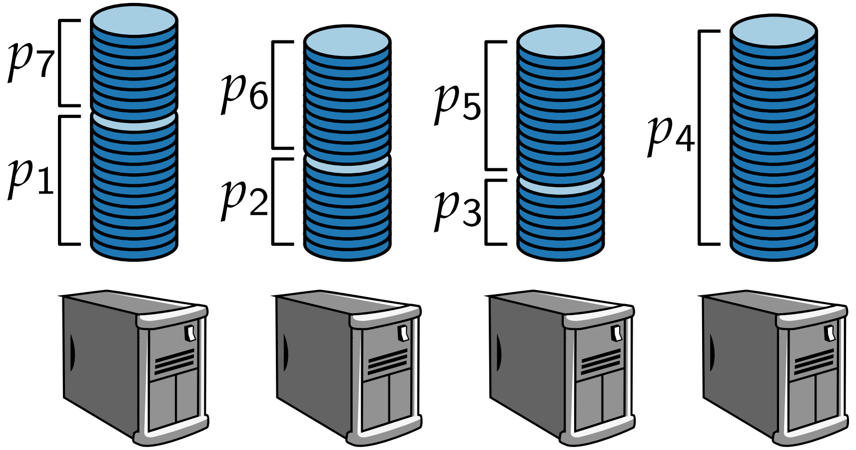
$\textsc{ListScheduling}(J_1, \ldots, J_n, m)$

Put the first m jobs on the m machines

Put next job on first free machine

Example.





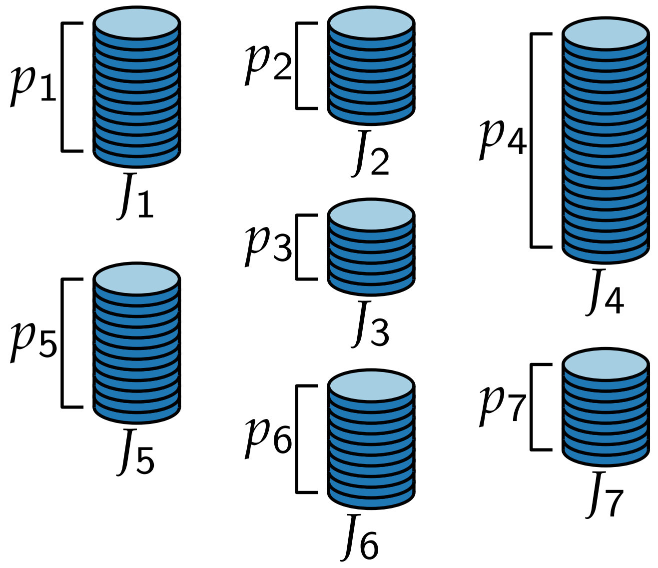
Multiprocessor Scheduling – List scheduling

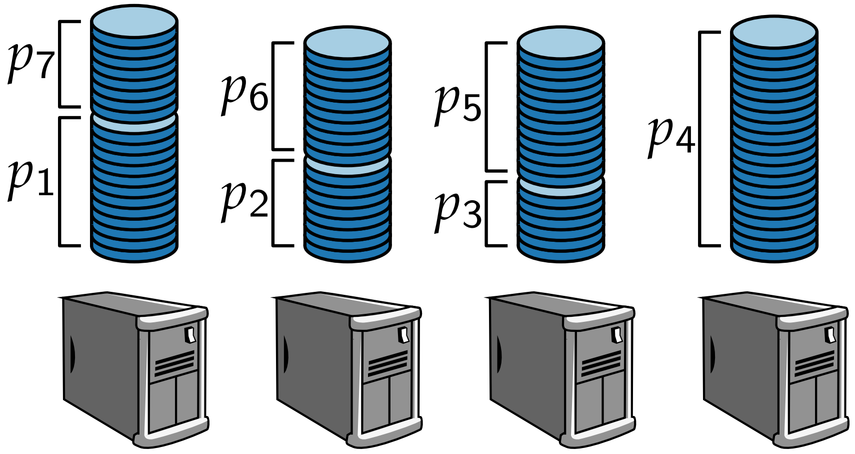
ListScheduling$(J_1, \ldots, J_n, m)$

Put the first m jobs on the m machines

Put next job on first free machine

Example.





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

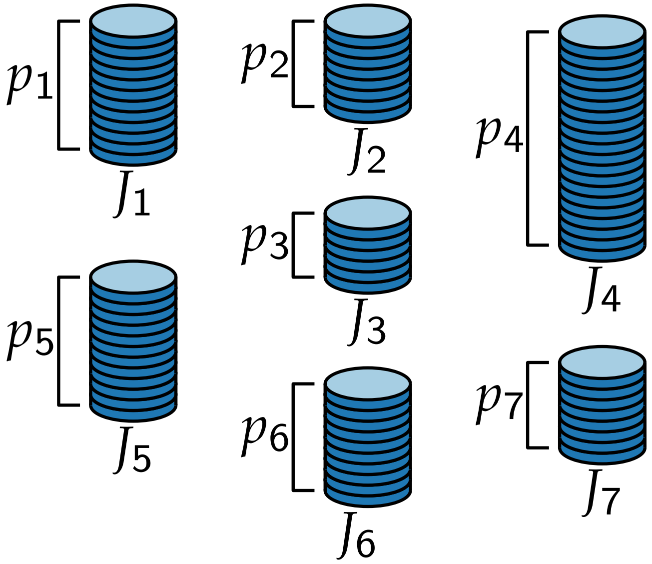
Multiprocessor Scheduling – List scheduling

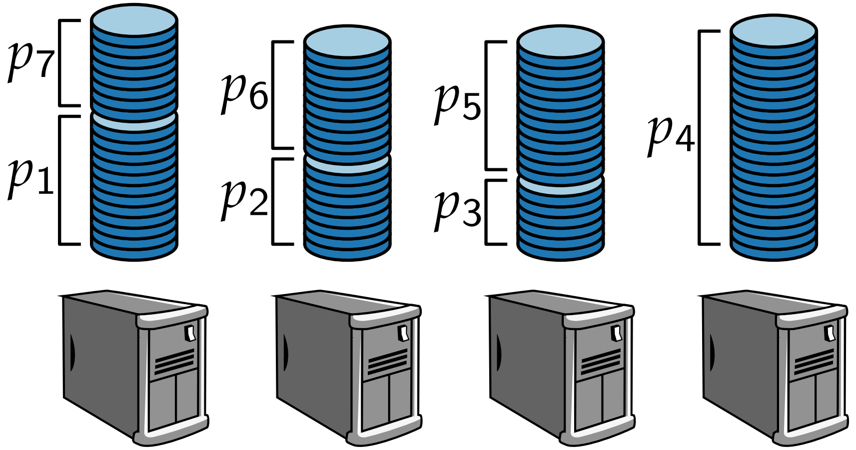
ListScheduling$(J_1, \ldots, J_n, m)$

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

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

Example.





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

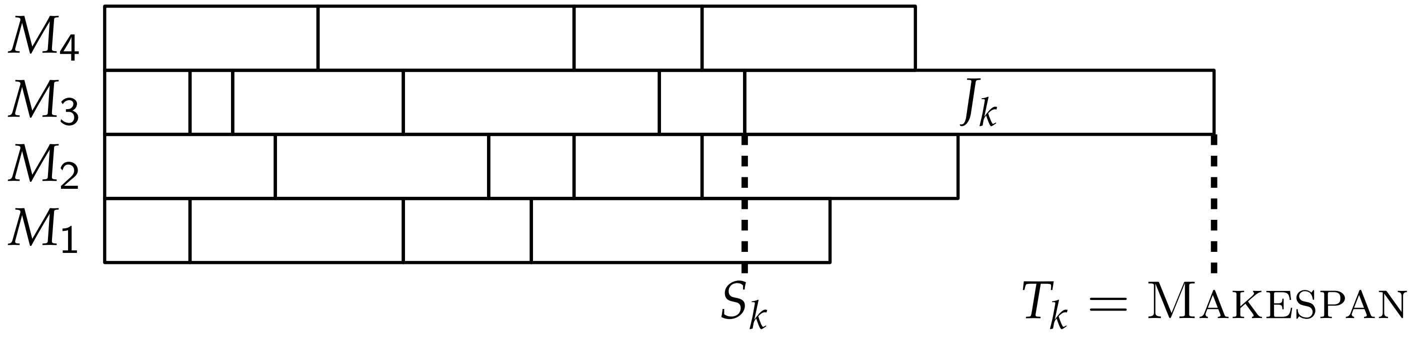
Multiprocessor Scheduling – List scheduling (proof)

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

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

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$



Multiprocessor Scheduling – List scheduling (proof)

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

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

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

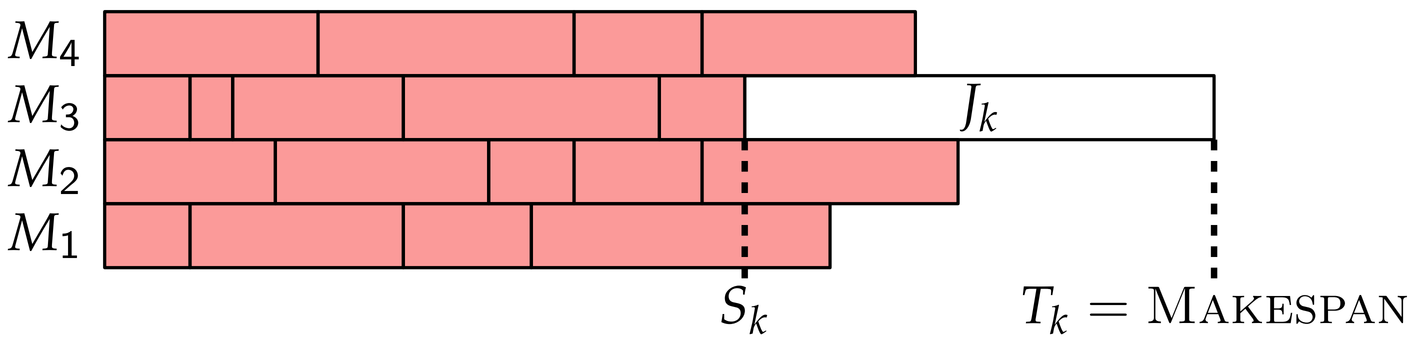
Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

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

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

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

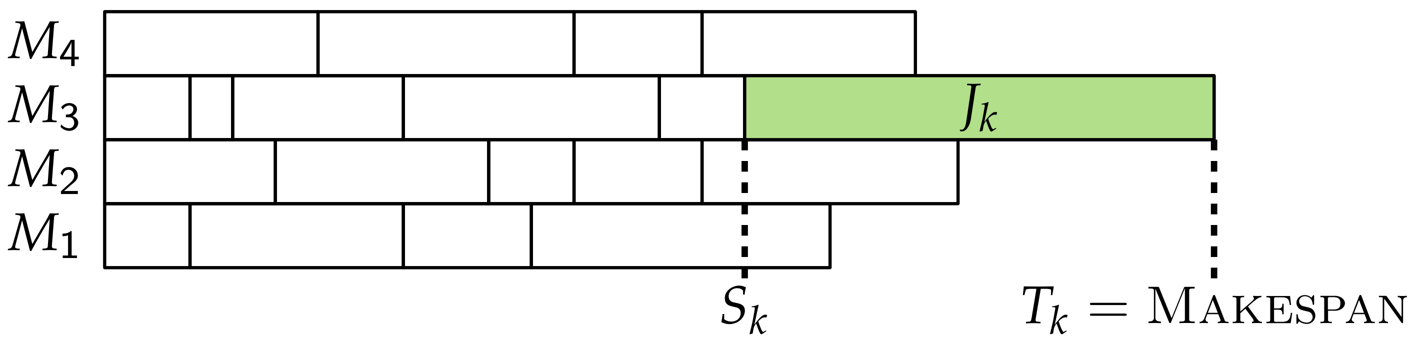
- No machine idles at time S_k.

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

weight of all jobs but J_k

evenly distributed on m machines

- For an optimal Makespan T_{OPT}, we have:
- $T_{\text{OPT}} \geq p_k$



Multiprocessor Scheduling – List scheduling (proof)

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

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

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \textsc{Makespan}$

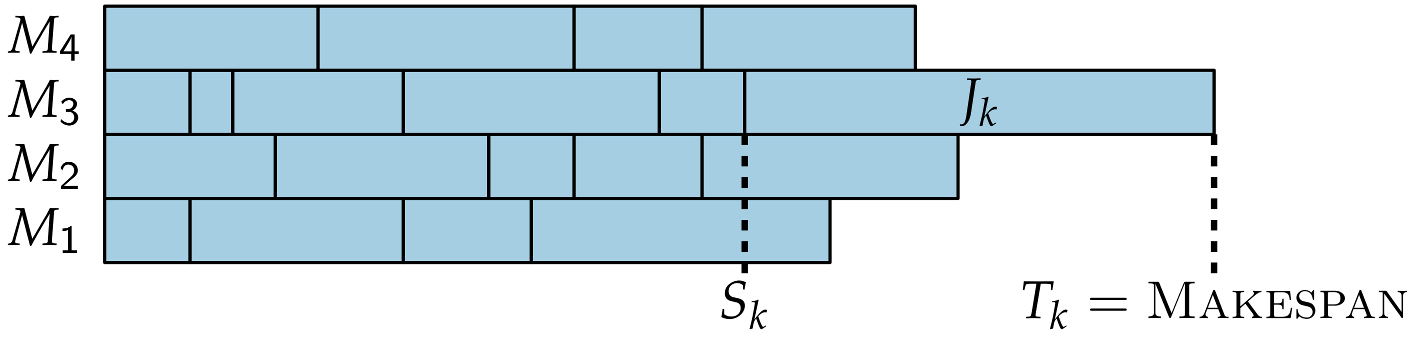
- No machine idles at time S_k.

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

weight of all jobs but J_k
evenly distributed on m machines

- For an optimal $\textsc{Makespan}$ 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 next job on first free machine

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

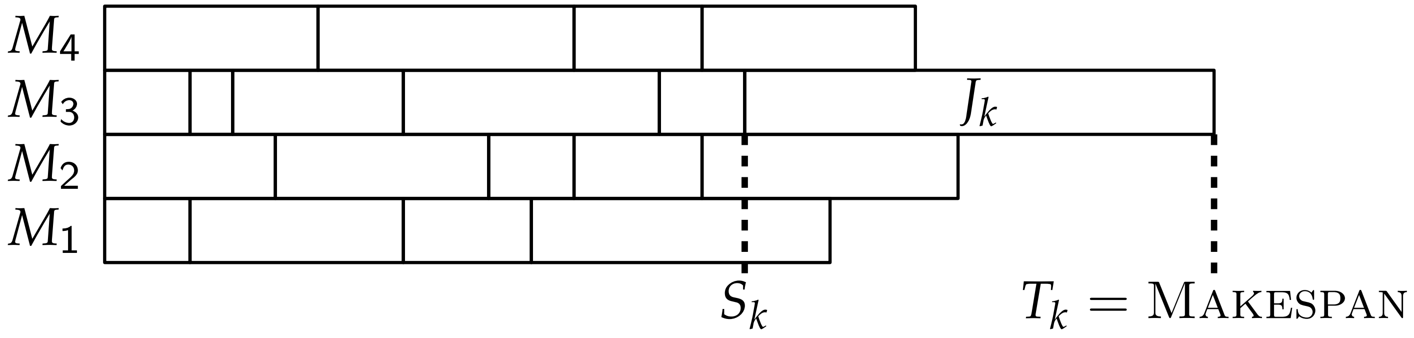
Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

- 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 an 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)

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

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

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \textsc{Makespan}$

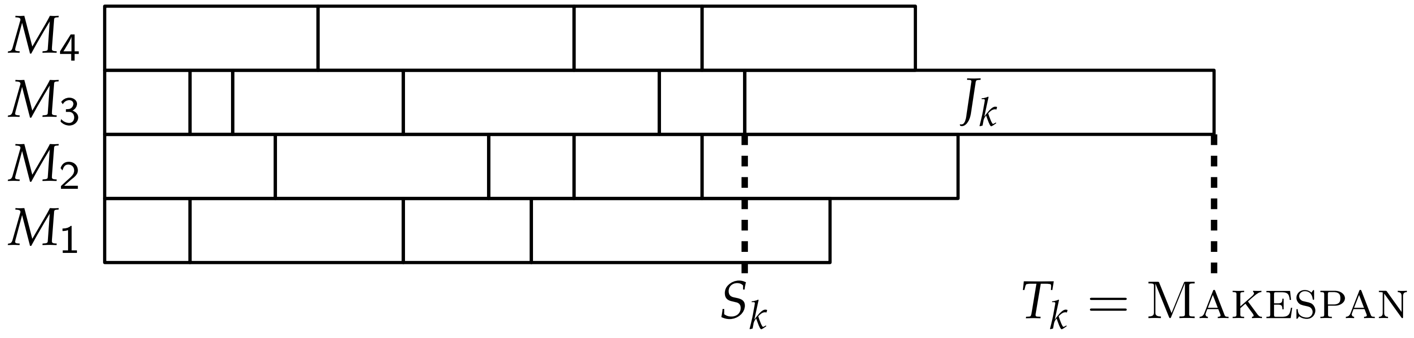
- No machine idles at time S_k.

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

- For an optimal $\textsc{Makespan}$ 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:

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



Multiprocessor Scheduling – List scheduling (proof)

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

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

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

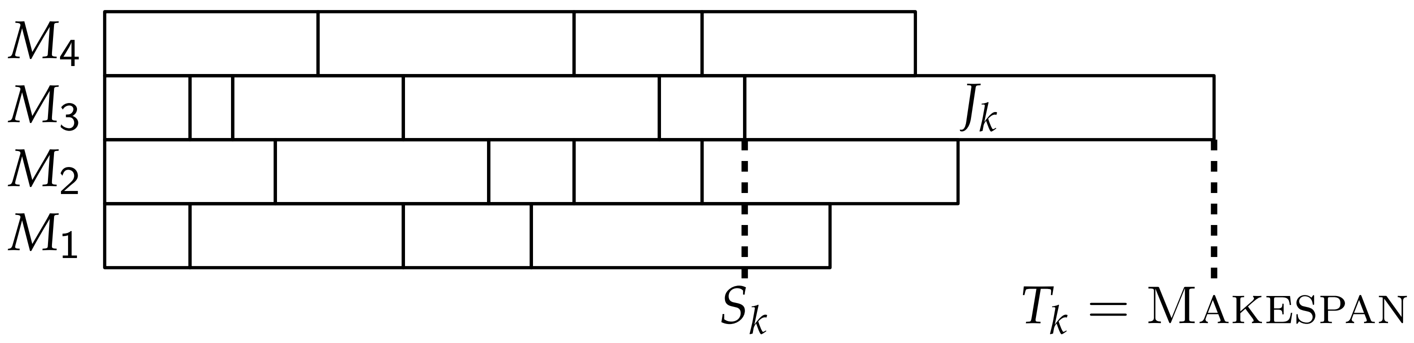
- No machine idles at time S_k.

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

- For an 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)

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

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

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

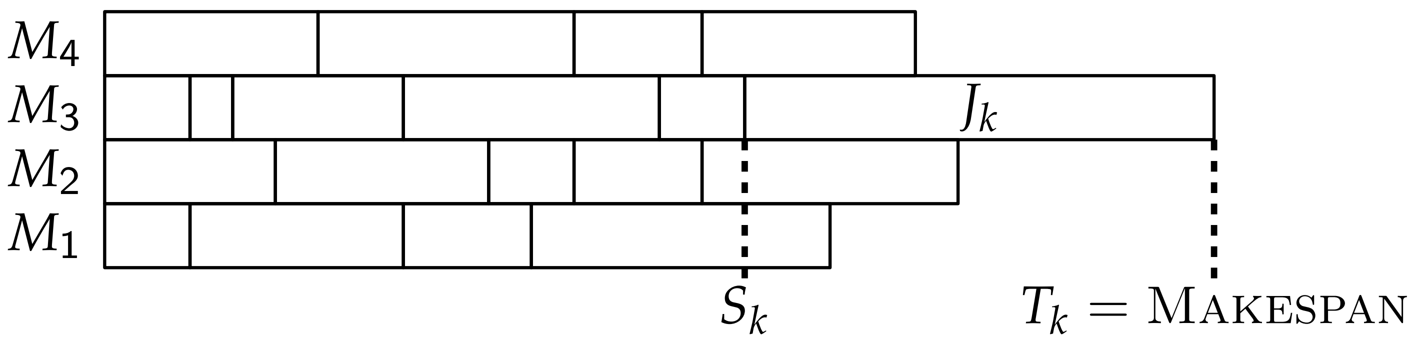
Proof. Let J_k be the last job with start time S_k and finish time $T_k = \textsc{Makespan}$

- 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 an optimal $\textsc{Makespan}$ 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:

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

Multiprocessor Scheduling – List scheduling (proof)

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

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

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

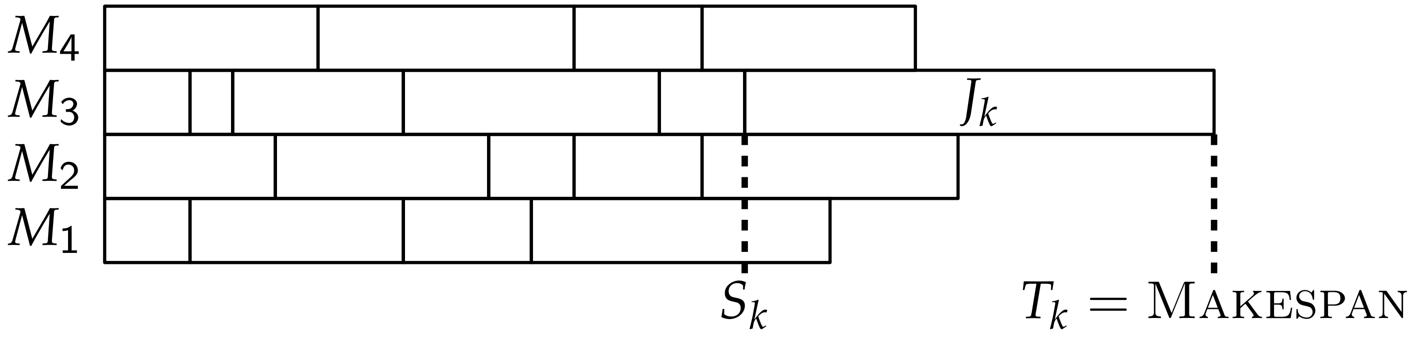
Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

- 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 an optimal Makespan T_{OPT}, we have:
  - $T_{\text{OPT}} \ge p_k$
  - $T_{\text{OPT}} \ge \frac{1}{m} \sum_{i=1}^{n} p_i$ — weight of all jobs evenly distributed



- Hence:

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

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 reamining 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.

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

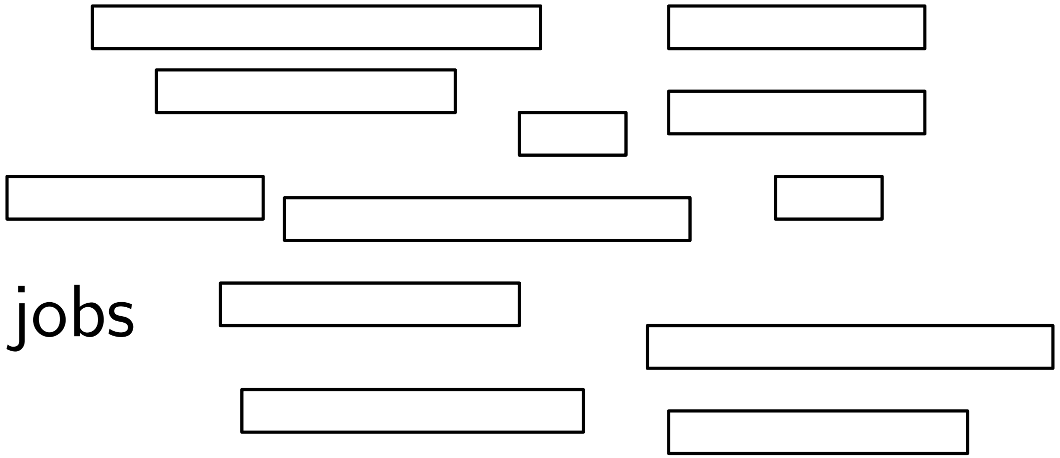
Sort jobs in descending order of runtime

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

Use ListScheduling for the reamining 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.

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

Sort jobs in descending order of runtime

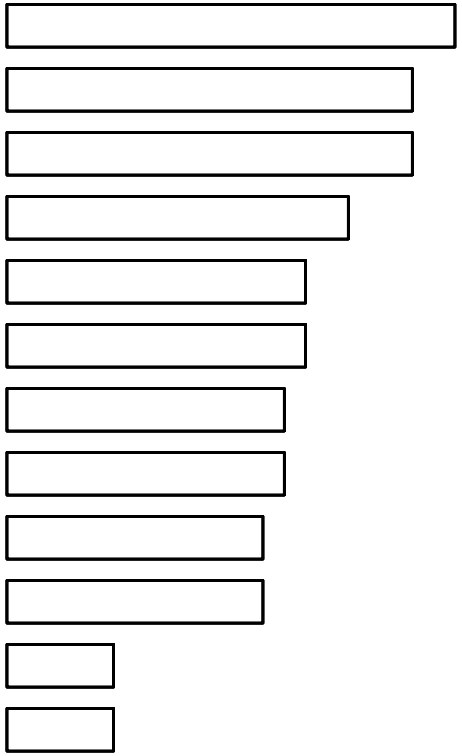
Schedule the ℓ longest jobs $J_1, \dots, J_\ell$ optimally

Use ListScheduling for the reamining jobs $J_{\ell+1}, \dots, J_n$

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

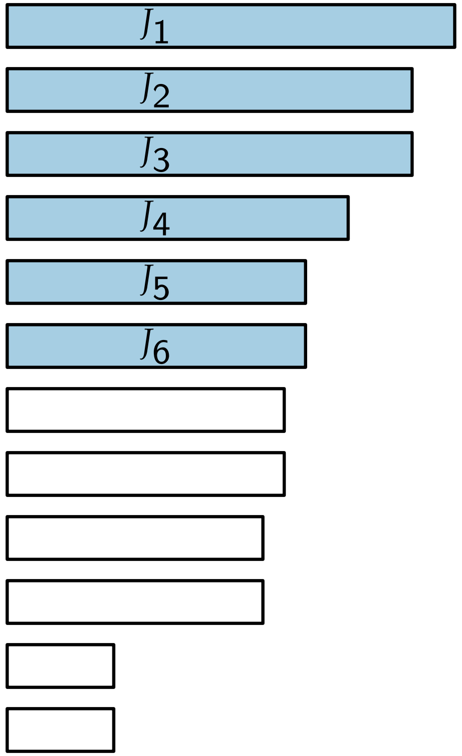
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

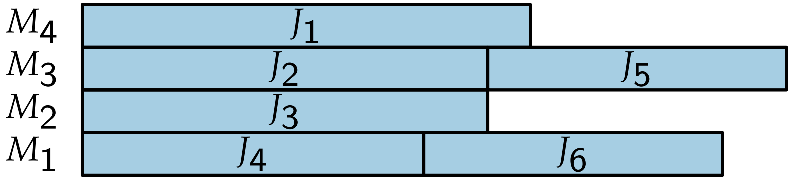
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

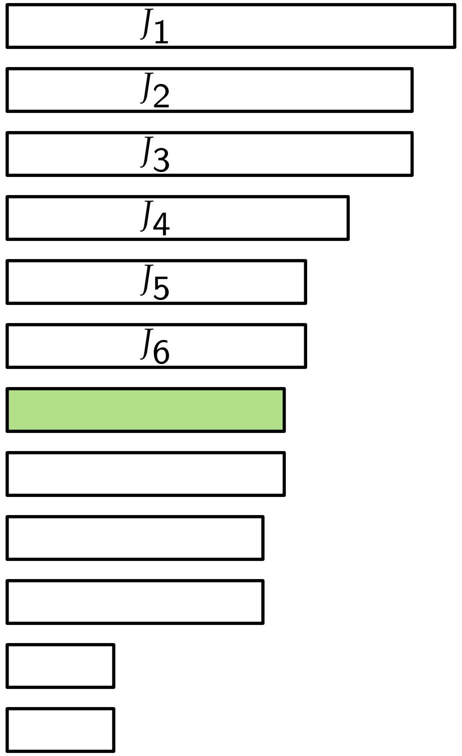
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

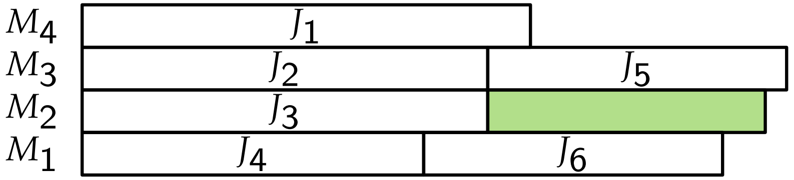
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

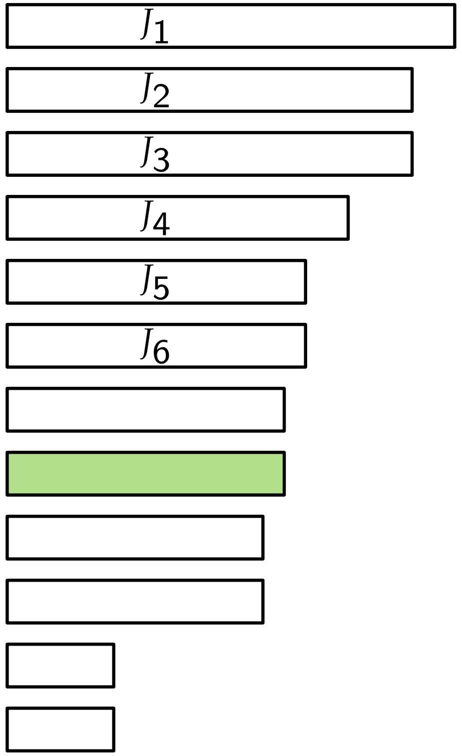
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

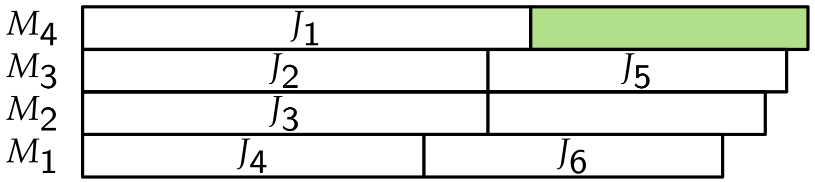
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

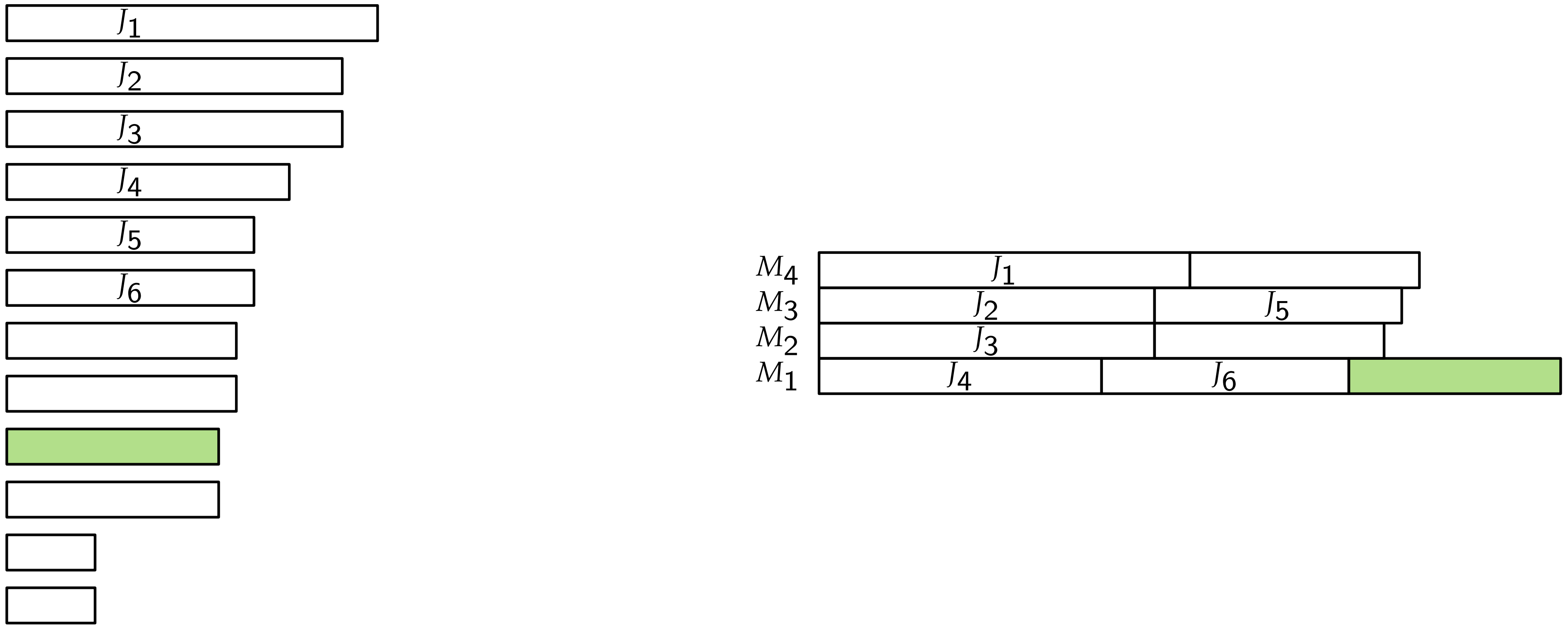
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

Use ListScheduling for the reamining 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.

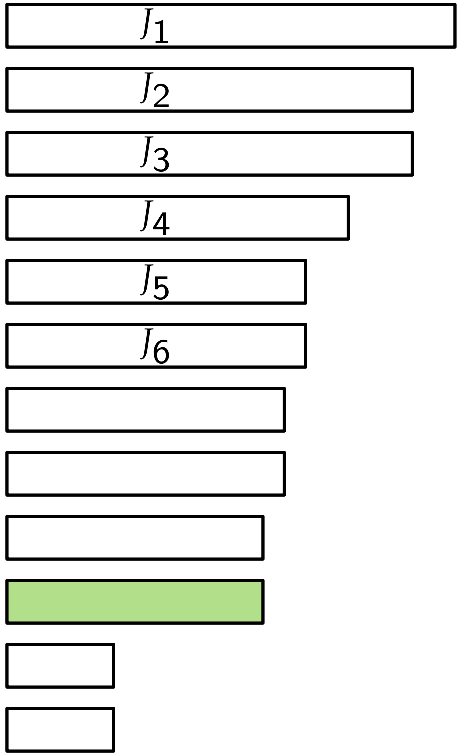
$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$

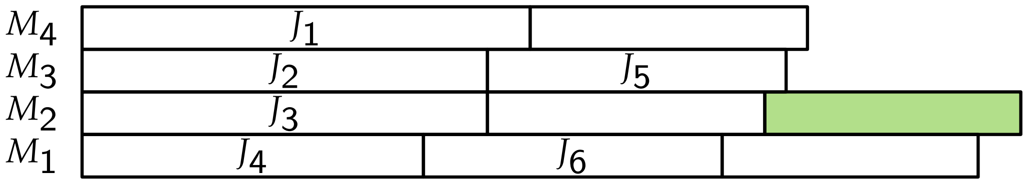
- Sort jobs in descending order of runtime
- Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally
- Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

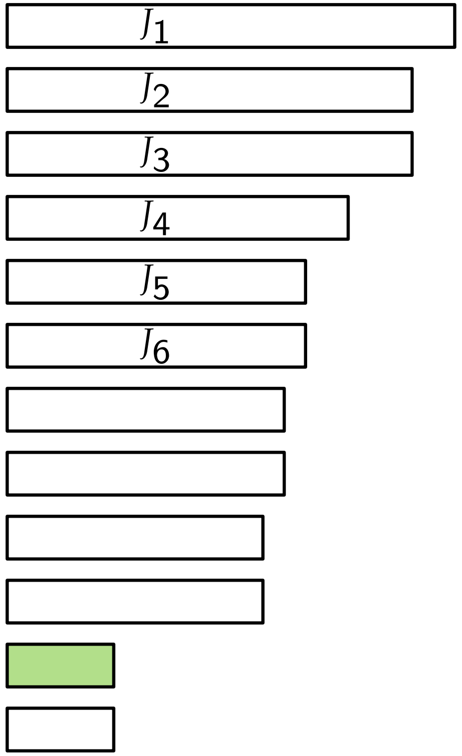
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

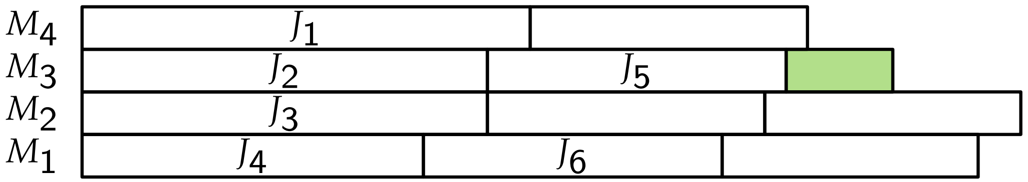
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

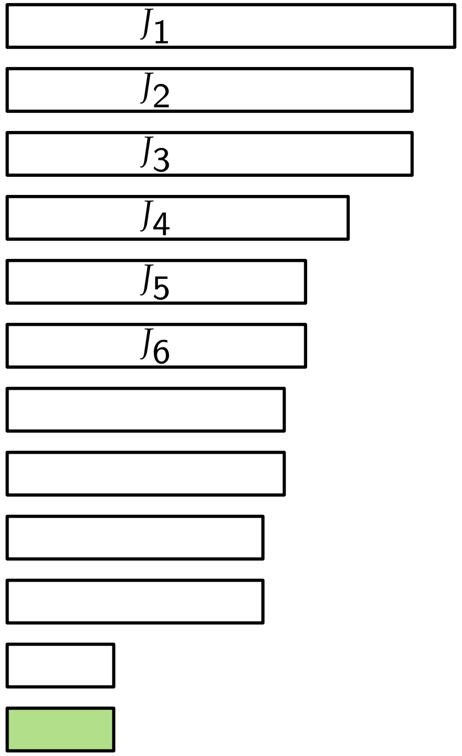
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

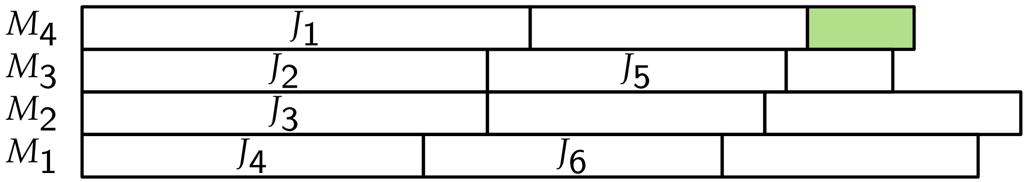
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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

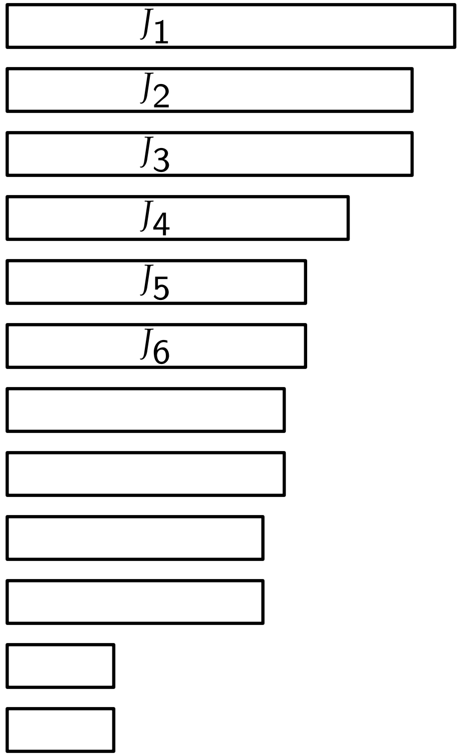
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally

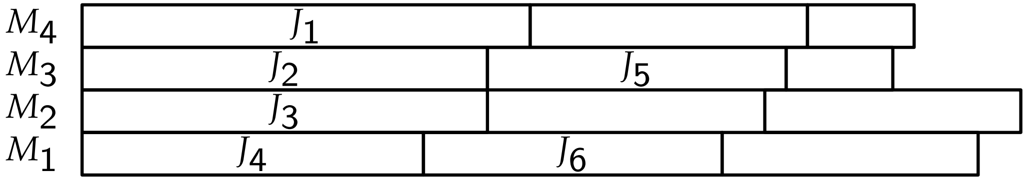
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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.

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

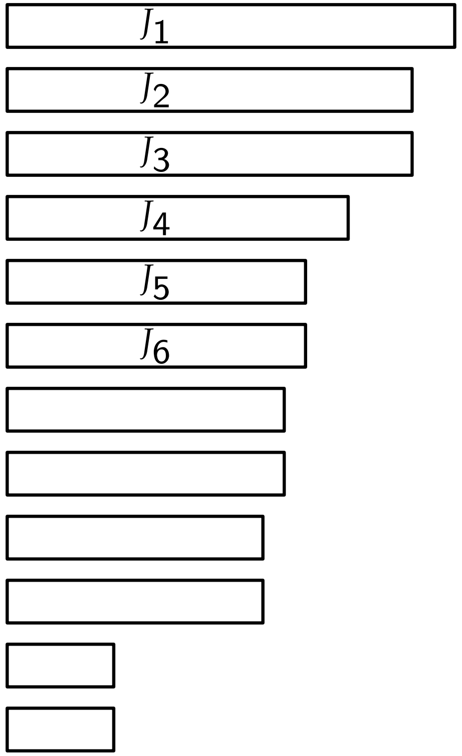
- Sort jobs in descending order of runtime — $\mathcal{O}(n \log n)$
- Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally — $\mathcal{O}(m^\ell)$
- Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$ — $\mathcal{O}(n)$

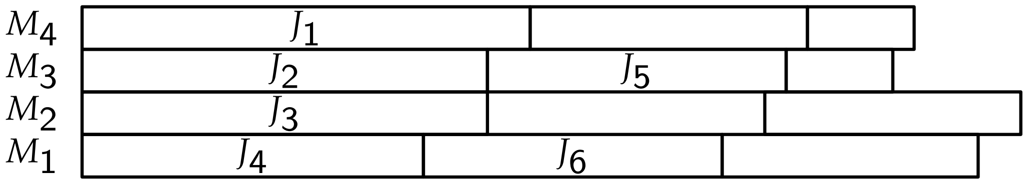
- Polynomial time for constant ℓ: $\mathcal{O}(m^\ell + n \log n)$

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 reamining jobs $J_{\ell+1}, \ldots, J_n$	$\mathcal{O}(n)$

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

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.

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 reamining jobs $J_{\ell+1}, \ldots, J_n$ $\mathcal{O}(n)$

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

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.

- 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 reamining jobs $J_{\ell+1}, \dots, J_n$	$\mathcal{O}(n)$

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

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.

- For $\varepsilon > 0$, choose ℓ such that $\mathcal{A}_\varepsilon = \mathcal{A}_{\ell(\varepsilon)}$ is a $(1 + \varepsilon)$-approximation algorithm.
- $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ isn't a 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 reamining jobs $J_{\ell+1}, \ldots, J_n$

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \textsc{Makespan}$

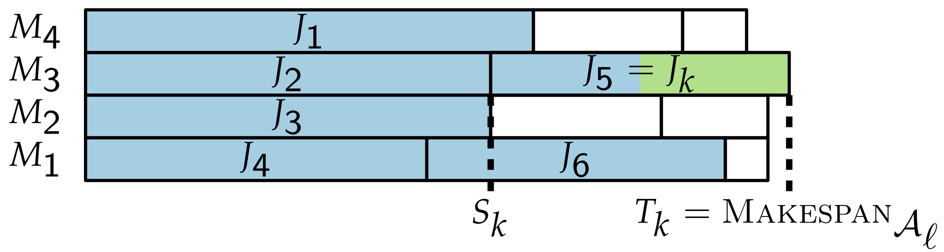
Multiprocessor Scheduling – PTAS (proof)

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 reamining jobs $J_{\ell+1}, \dots, J_n$

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

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



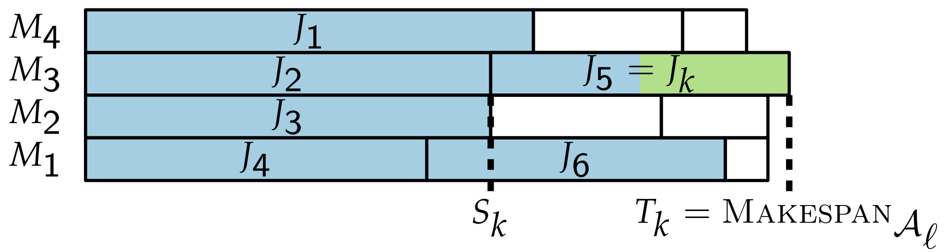
Multiprocessor Scheduling – PTAS (proof)

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 reamining jobs $J_{\ell+1}, \ldots, J_n$

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \textsc{Makespan}$

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)

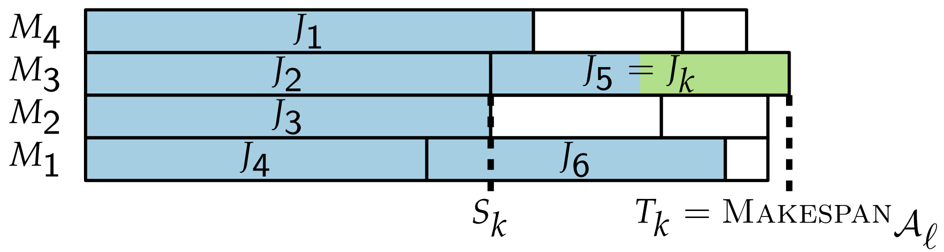
Theorem 8.
For constant $1 \le \ell \le n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

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

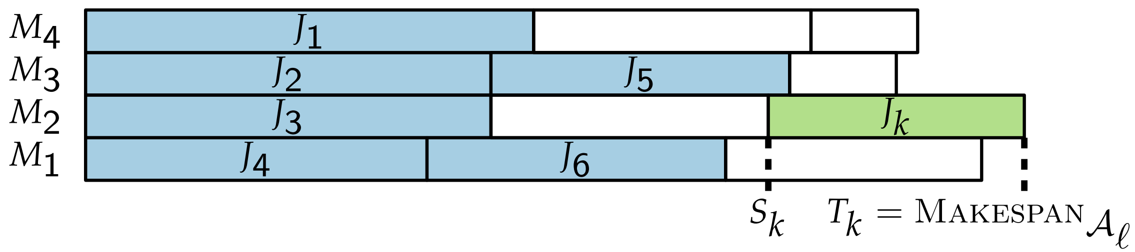
Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

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)

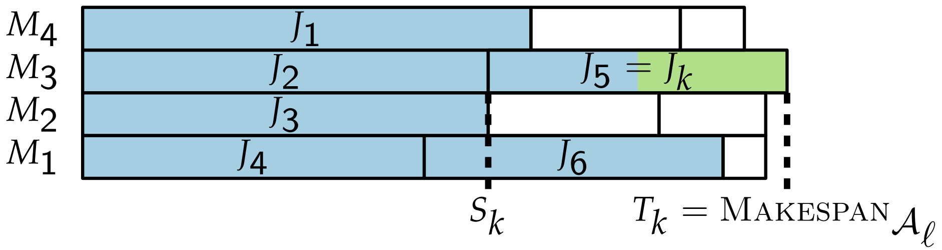
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

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

Proof. Let J_k be the last job with start time S_k and finish time $T_k = \text{Makespan}$

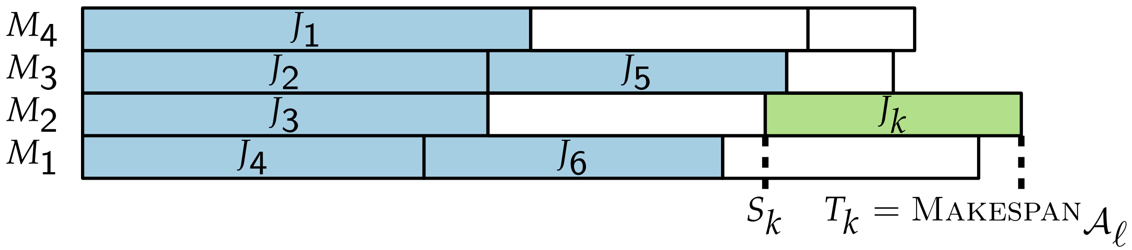
Case 1. J_k is one of the longest ℓ jobs $J_1, \ldots, J_\ell$.

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



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

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



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 reamining jobs $J_{\ell+1}, \ldots, J_n$

Proof of **Case 2**.

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

$$T_k = S_k + p_k$$

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.

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

Proof of **Case 2**.

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

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

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 reamining 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.

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

Proof of **Case 2**.

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

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

Multiprocessor Scheduling – PTAS (proof)

Theorem 8.
For constant $1 \leq \ell \leq n$, 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 reamining jobs $J_{\ell+1}, \ldots, J_n$

Proof of **Case 2**.

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

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

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.

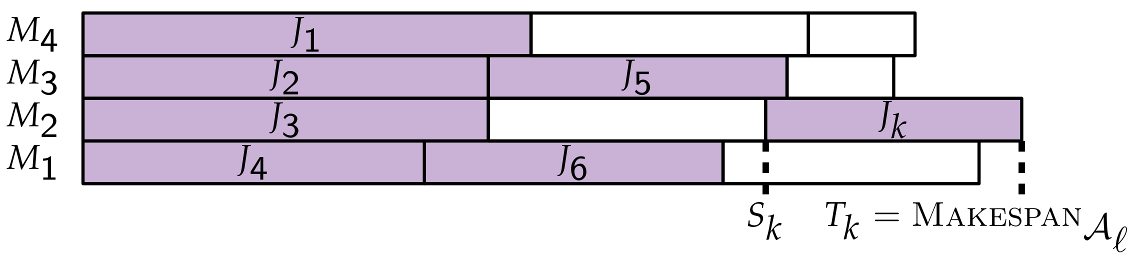
$\mathcal{A}_\ell(J_1, \ldots, J_n, m)$
Sort jobs in descending order of runtime
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally
Use ListScheduling for the reamining jobs $J_{\ell+1}, \ldots, J_n$

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⋆; each has lenght $\geq p_k$
  - ⋆ on average, each machine has more than $\frac{\ell}{m}$ of the $\ell + 1$ jobs
  - at least one machine achieves the average

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

can we do better?



Multiprocessor Scheduling – PTAS (proof)

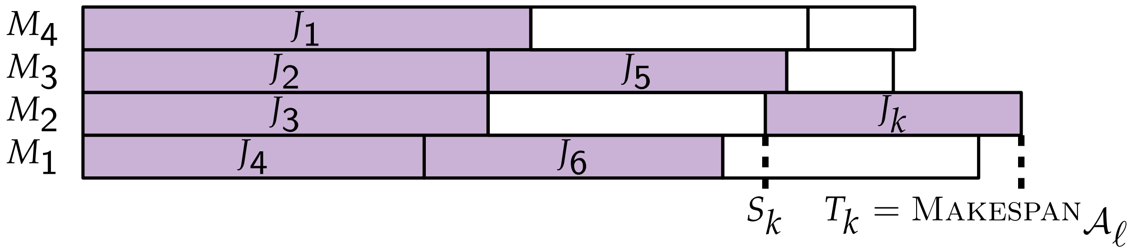
Theorem 8.
For constant $1 \leq \ell \leq n$, the algorithm $\mathcal{A}_\ell$ is a $1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{\ell}{m} \rfloor}$-approximation algorithm.

$\mathcal{A}_\ell(J_1, \ldots, J_n,\ m)$
Sort jobs in descending order of runtime
Schedule the ℓ longest jobs $J_1, \ldots, J_\ell$ optimally
Use ListScheduling for the reamining 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⋆; each has lenght $\geq p_k$
  - ⋆ 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).
- Not all problems can be approximated (Max Clique).
- Study of approximability of NP-hard problems yields a more fine-grained classification of the difficulty.

Discussion

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

Discussion

- Only "easy" NP-hard problems admit FPTAS (PTAS).
- Not all problems can be approximated (Max Clique).
- Study of approximability of NP-hard problems yields a more fine-grained classification of the difficulty.
- Approximation algorithms exist also for non-NP-hard problems
- Approximation algorithms can be of various types: greedy, local search, geometric, DP, ...
- One important technique is LP-relaxation (next lecture).
- Min Vertex Coloring on planar graphs can be approximated with an additive approximation guarantee of 2.
- Christofides' approximation algorithm for Metric TSP has approximation factor 1.5.

Literature

Main references

- [Jansen, Margraf Ch3] “Approximative Algorithmen und Nichtapproximierbarkeit”
- [Williamson, Shmoys Ch3] “The Design of Approximation Algorithms”

Another book recommendation:

- [Vazirani] “Approximation Algorithms”

and don’t forget our lecture

- **Approximation Algorithms**.

For more precise definitions see

- https://go.uniwue.de/approxdef