Chapter 4. Greedy Algorithms

Algorithmics and Programming III

FIB

Albert Oliveras Enric Rodríguez

Q1 2019–2020

Version November 4, 2019

Example: Coin Exchange

- What is the minimum number of coins that add up to 63 cents of €?

Example: Coin Exchange

- What is the minimum number of coins that add up to 63 cents of €?
  - 4 coins: $50 + 10 + 2 + 1$

Example: Coin Exchange

- What is the minimum number of coins that add up to 63 cents of €?
  - 4 coins: $50 + 10 + 2 + 1$
- Which algorithm do we use to give change?
  - While there is money to give, use the largest possible coin

Example: Coin Exchange

- What is the minimum number of coins that add up to 63 cents of €?
  - 4 coins: $50 + 10 + 2 + 1$
- Which algorithm do we use to give change?
  - While there is money to give, use the largest possible coin
- What if we add a coin of 8 cents?
  - The previous *greedy* algorithm uses 4 coins: 50 + 10 + 2 + 1

Example: Coin Exchange

- What is the minimum number of coins that add up to 63 cents of €?
  - 4 coins: $50 + 10 + 2 + 1$
- Which algorithm do we use to give change?
  - While there is money to give, use the largest possible coin
- What if we add a coin of 8 cents?
  - The previous *greedy* algorithm uses 4 coins: 50 + 10 + 2 + 1
  - However, there is a solution with only 3 coins: 50 + 8 + 5

What are Greedy Algorithms?

- An algorithm is greedy if it builds up a solution in small steps, taking a decision at each step myopically to optimize a certain criterion

What are Greedy Algorithms?

- An algorithm is greedy if it builds up a solution in small steps, taking a decision at each step myopically to optimize a certain criterion
- We may want to do three things:
  1. EASY: invent a greedy algorithm
  2. MODERATE: invent a **correct** greedy algorithm
  3. DIFFICULT: **prove** that a greedy algorithm is correct

What are Greedy Algorithms?

- An algorithm is greedy if it builds up a solution in small steps, taking a decision at each step myopically to optimize a certain criterion

- We may want to do three things:
  1. EASY: invent a greedy algorithm
  2. MODERATE: invent a **correct** greedy algorithm
  3. DIFFICULT: **prove** that a greedy algorithm is correct

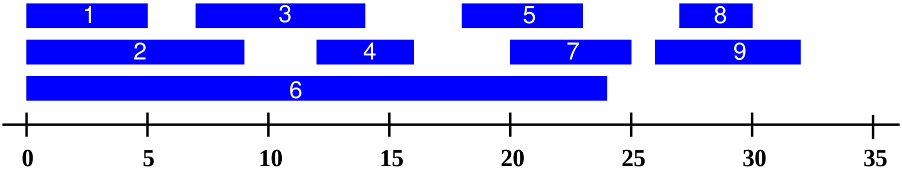
- In the following we will:
  - Introduce and revisit greedy algorithms for well-known problems
  - Prove them correct using a variety of techniques

Interval Scheduling Problem

- Imagine we have a single lab room and N courses that request to use it
- We know the starting time $s(i)$ and the finishing time $f(i)$ of each course
- Requests of two overlapping courses cannot be both met
- We wish to find out the maximum number of requests can be accepted

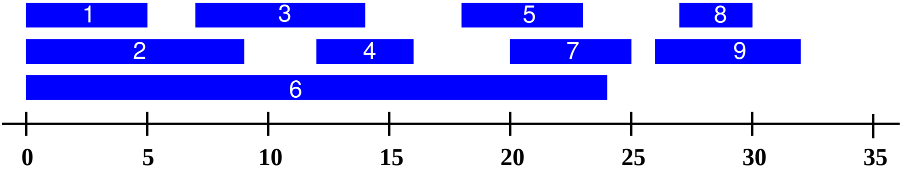
Interval Scheduling Problem

- Imagine we have a single lab room and N courses that request to use it
- We know the starting time $s(i)$ and the finishing time $f(i)$ of each course
- Requests of two overlapping courses cannot be both met
- We wish to find out the maximum number of requests can be accepted
- Example:



- $\{1,3,5,8\}$ or $\{2,4,7,9\}$ are feasible choices. Are they optimal?

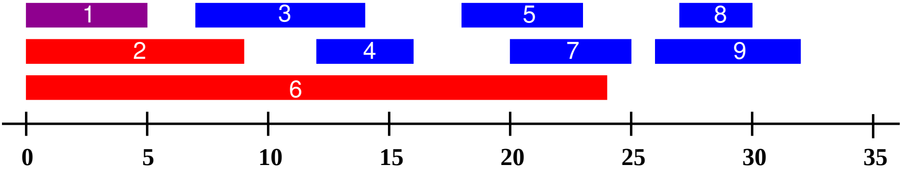
Interval Scheduling Problem



GREEDY ALGORITHM 1:

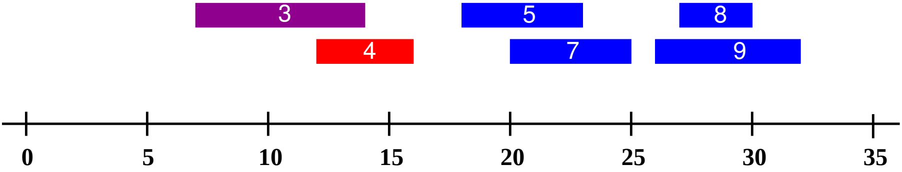
- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course i that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:

Interval Scheduling Problem



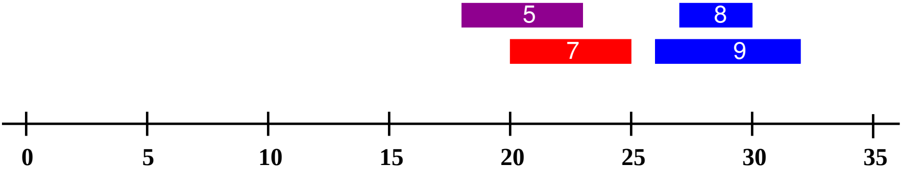
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course *i* that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping



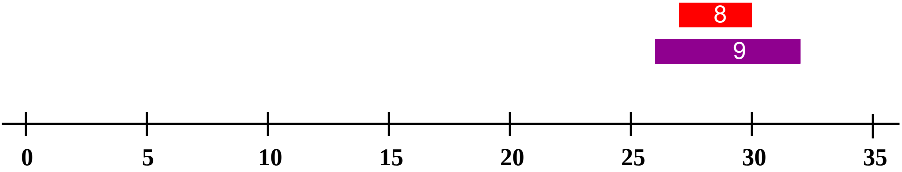
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course i that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping
  2. Choose 3. Course 4 is discarded.



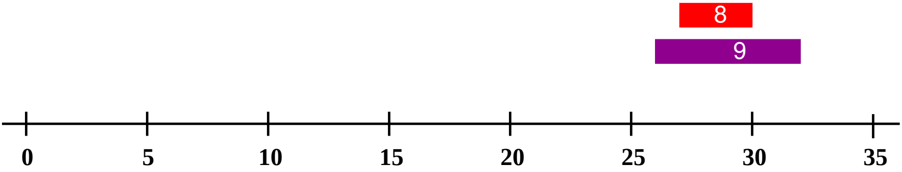
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course *i* that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.



GREEDY ALGORITHM 1:

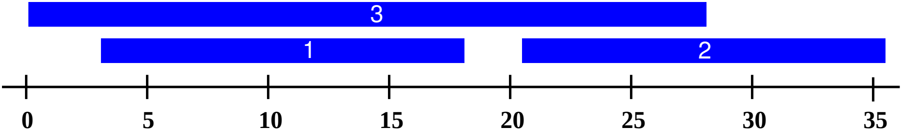
- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course i that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.
  4. Choose 9. Course 8 is discarded.



GREEDY ALGORITHM 1:

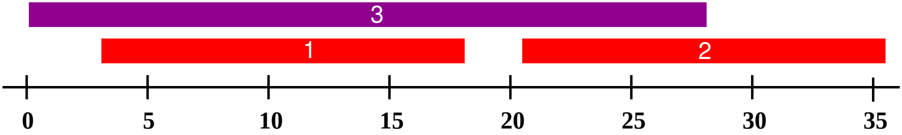
- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course *i* that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.
  4. Choose 9. Course 8 is discarded.
- Is this algorithm correct? Can we cook up an example for which it fails?

Interval Scheduling Problem



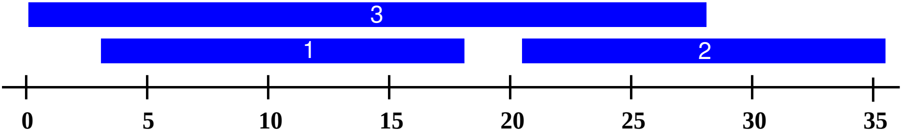
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:



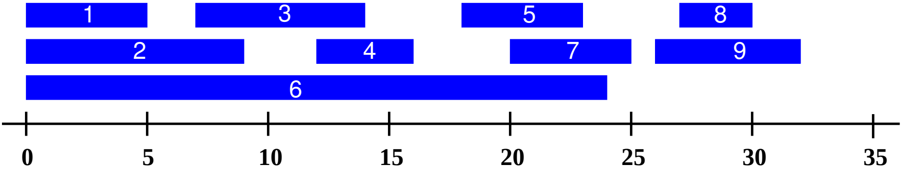
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 3. Courses 1 and 2 are discarded due to overlapping.



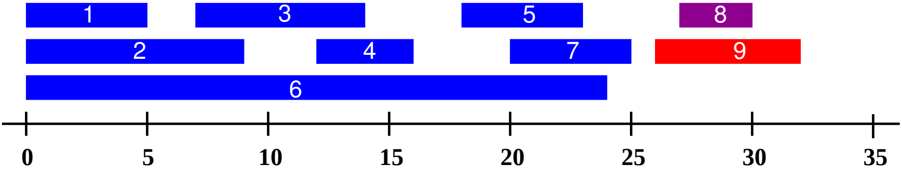
GREEDY ALGORITHM 1:

- Idea: start using the room as soon as possible
- Algorithm: while there are available courses, select the available course that starts as soon as possible. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 3. Courses 1 and 2 are discarded due to overlapping.
- But $\{1, 2\}$ is a better solution. So this greedy algorithm is not correct.



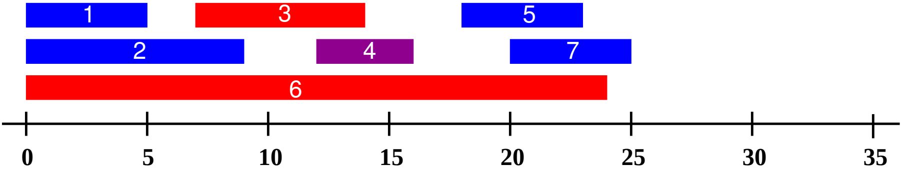
GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:



GREEDY ALGORITHM 2:

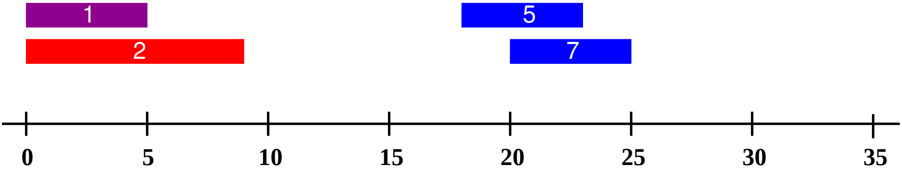
- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.



GREEDY ALGORITHM 2:

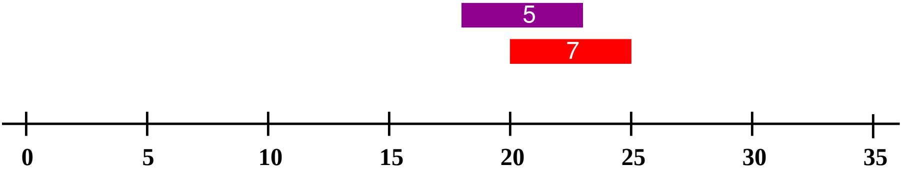
- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 4. Courses 3 and 6 are discarded.

Interval Scheduling Problem



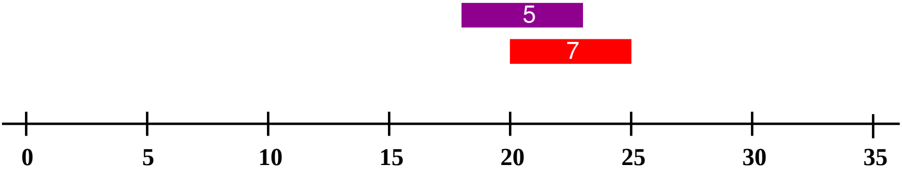
GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 4. Courses 3 and 6 are discarded.
  3. Choose 1. Course 2 is discarded.



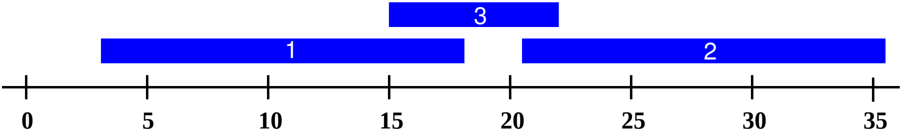
GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 4. Courses 3 and 6 are discarded.
  3. Choose 1. Course 2 is discarded.
  4. Choose 5. Course 7 is discarded.



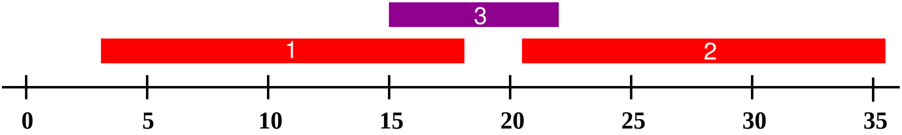
GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 4. Courses 3 and 6 are discarded.
  3. Choose 1. Course 2 is discarded.
  4. Choose 5. Course 7 is discarded.
- Is this algorithm correct? Can we cook up an example for which it fails?



GREEDY ALGORITHM 2:

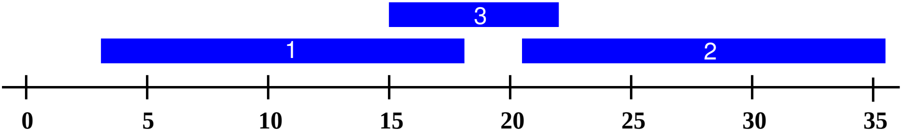
- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:



GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 3. Courses 1 and 2 are discarded due to overlapping.

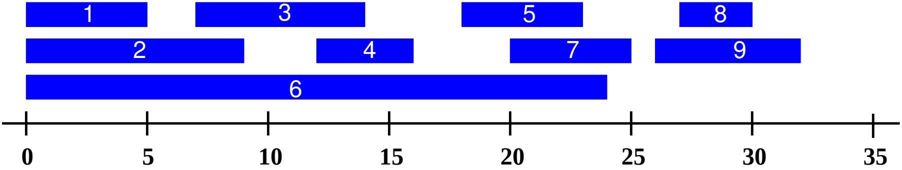
Interval Scheduling Problem



GREEDY ALGORITHM 2:

- Idea: choosing very long tasks might not be a good idea
- Algorithm: while there are available courses, select the available course i for which $f(i) - s(i)$ is smallest. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 3. Courses 1 and 2 are discarded due to overlapping.
- But $\{1, 2\}$ is a better solution. So this greedy algorithm is not correct.

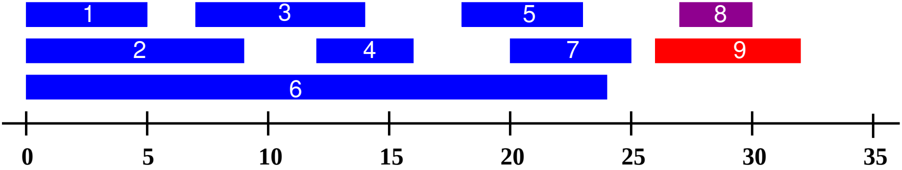
Interval Scheduling Problem



GREEDY ALGORITHM 3:

- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:

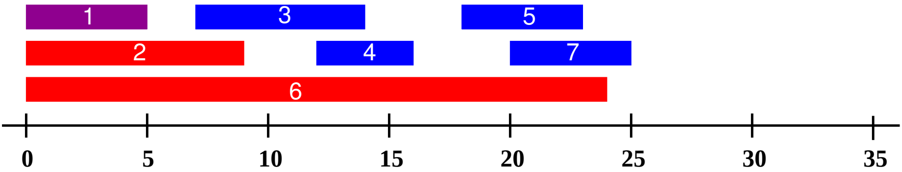
Interval Scheduling Problem



GREEDY ALGORITHM 3:

- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.

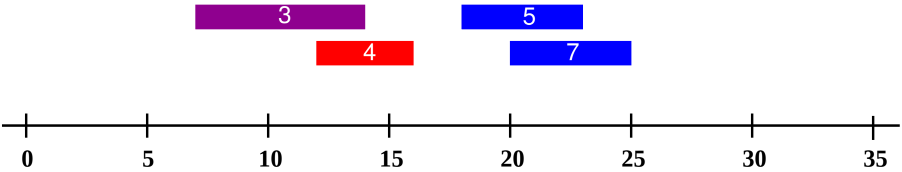
Interval Scheduling Problem



GREEDY ALGORITHM 3:

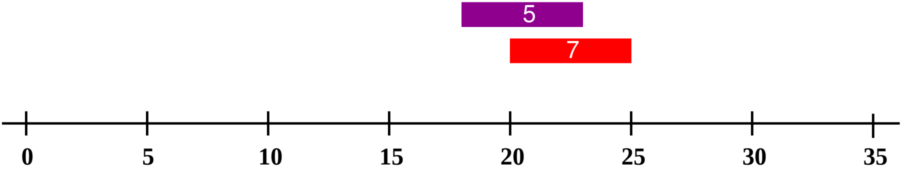
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 1. Courses 2 and 6 are discarded.

Interval Scheduling Problem



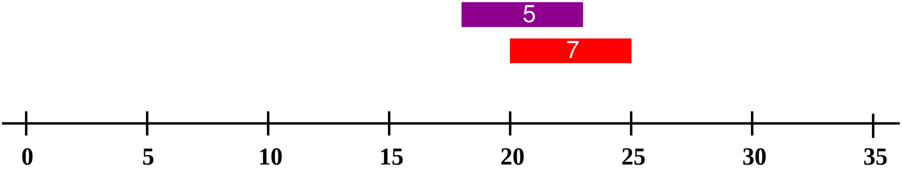
GREEDY ALGORITHM 3:

- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 1. Courses 2 and 6 are discarded.
  3. Choose 3. Course 4 is discarded.



GREEDY ALGORITHM 3:

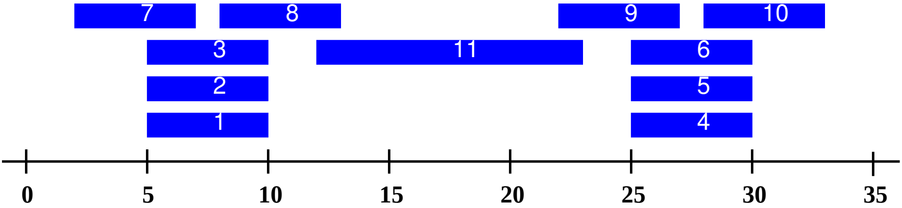
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 1. Courses 2 and 6 are discarded.
  3. Choose 3. Course 4 is discarded.
  4. Choose 5. Course 7 is discarded.



GREEDY ALGORITHM 3:

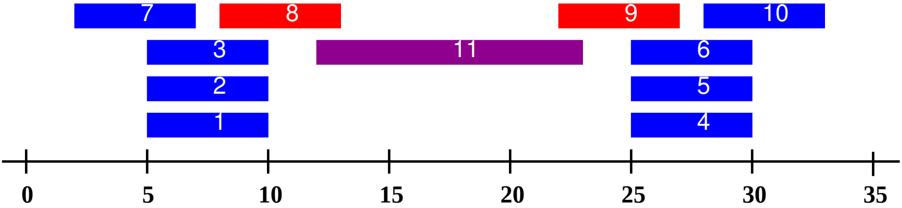
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 8. Course 9 is discarded due to overlapping.
  2. Choose 1. Courses 2 and 6 are discarded.
  3. Choose 3. Course 4 is discarded.
  4. Choose 5. Course 7 is discarded.
- Is this algorithm correct? Can we cook up an example for which it fails?

Interval Scheduling Problem



GREEDY ALGORITHM 3:

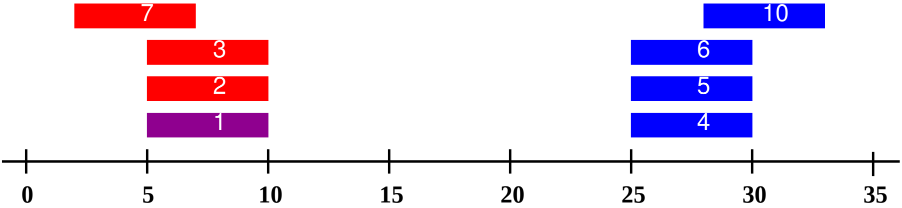
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course i that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:



GREEDY ALGORITHM 3:

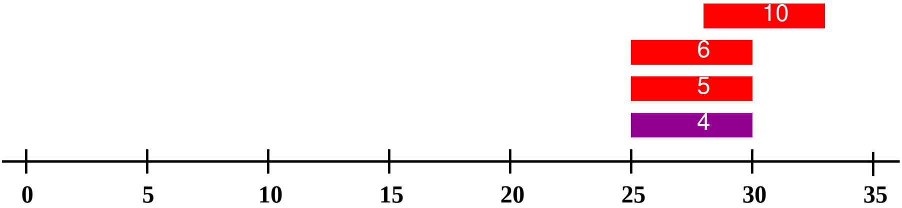
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 11. Courses 8 and 9 are discarded due to overlap.

Interval Scheduling Problem



GREEDY ALGORITHM 3:

- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course i that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 11. Courses 8 and 9 are discarded due to overlap.
  2. Choose 1. Courses 2, 3 and 7 are discarded.

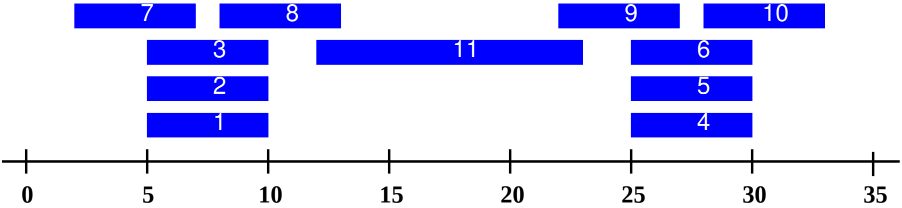


GREEDY ALGORITHM 3:

- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:

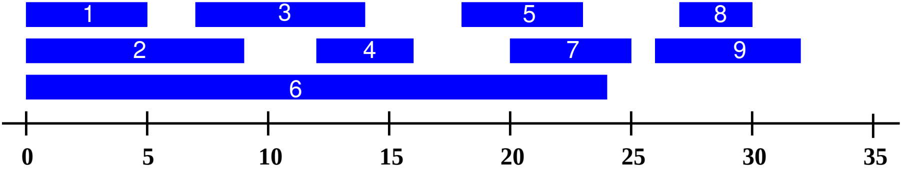
  1. Choose 11. Courses 8 and 9 are discarded due to overlap.
  2. Choose 1. Courses 2, 3 and 7 are discarded.
  3. Choose 4. Courses 5, 6 and 10 are discarded.

Interval Scheduling Problem



GREEDY ALGORITHM 3:

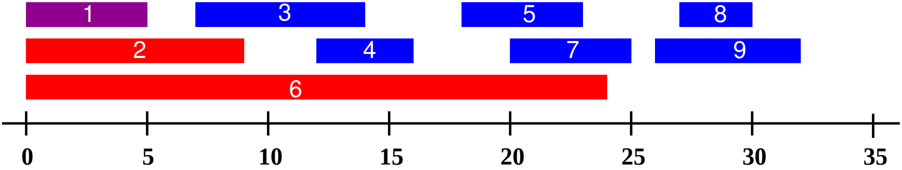
- Idea: choose tasks that compete with few other tasks
- Algorithm: while there are available courses, select the available course *i* that overlaps with the fewest number of other available tasks. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with *i* as non-available.
- Algorithm execution:
  1. Choose 11. Courses 8 and 9 are discarded due to overlap.
  2. Choose 1. Courses 2, 3 and 7 are discarded.
  3. Choose 4. Courses 5, 6 and 10 are discarded.
- But $\{7, 8, 9, 10\}$ is better. So this greedy algorithm is not correct.



GREEDY ALGORITHM 4:

- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:

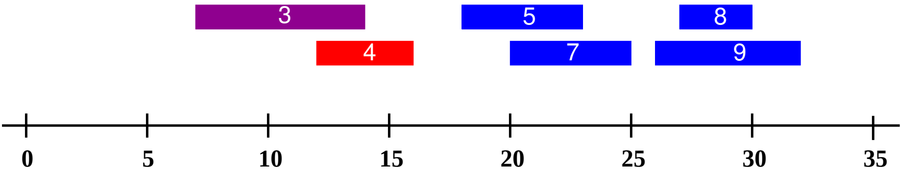
Interval Scheduling Problem



GREEDY ALGORITHM 4:

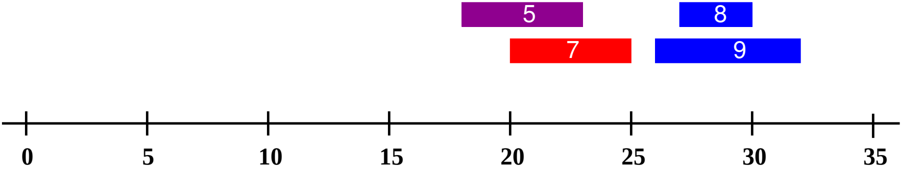
- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping.

Interval Scheduling Problem



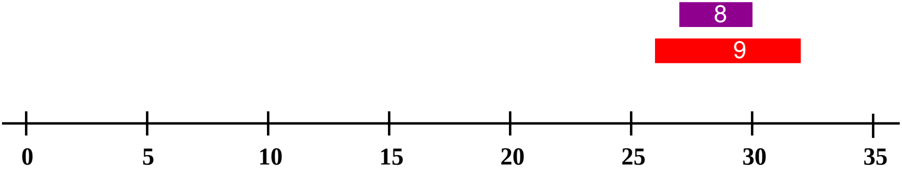
GREEDY ALGORITHM 4:

- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping.
  2. Choose 3. Course 4 is discarded.



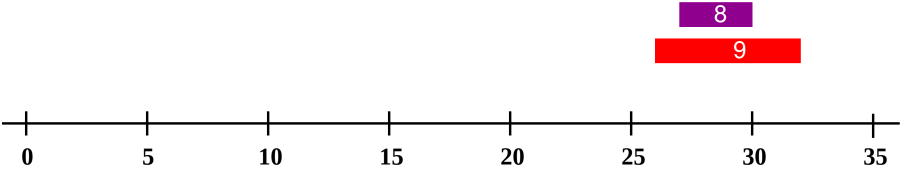
GREEDY ALGORITHM 4:

- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping.
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.



GREEDY ALGORITHM 4:

- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping.
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.
  4. Choose 8. Course 9 is discarded.



GREEDY ALGORITHM 4:

- Idea: choose tasks that finish early
- Algorithm: while there are available courses, select the available course i that has the lowest $f(i)$ among the available ones. If there is a tie, pick the one with the lowest index. Mark all courses that overlap with i as non-available.
- Algorithm execution:
  1. Choose 1. Courses 2 and 6 are discarded due to overlapping.
  2. Choose 3. Course 4 is discarded.
  3. Choose 5. Course 7 is discarded.
  4. Choose 8. Course 9 is discarded.
- Is this algorithm correct? Can we cook up an example for which it fails?

Interval Scheduling Problem

We will prove that the previous algorithm is correct.
Our strategy is the following:

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$.

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$.
- Let us consider an optimal subset of courses $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ for $\mathcal{S}$. We can order them such that for all $1 \leq i < m : f(o_i) \leq s(o_{i+1})$.

Interval Scheduling Problem

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \dots, a_k\}$.
- Let us consider an optimal subset of courses $\mathcal{O} = \{o_1, o_2, \dots, o_m\}$ for $\mathcal{S}$. We can order them such that for all $1 \leq i < m : f(o_i) \leq s(o_{i+1})$.
- We want to prove that $|\mathcal{A}| = |\mathcal{O}|$ (i.e. $k = m$) (we do not ask $\mathcal{A} = \mathcal{O}$ as there might be multiple optimal solutions).

Interval Scheduling Problem

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$.
- Let us consider an optimal subset of courses $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ for $\mathcal{S}$. We can order them such that for all $1 \leq i < m : f(o_i) \leq s(o_{i+1})$.
- We want to prove that $|\mathcal{A}| = |\mathcal{O}|$ (i.e. $k = m$) (we do not ask $\mathcal{A} = \mathcal{O}$ as there might be multiple optimal solutions).
- We know that $k \leq m$ because $\mathcal{O}$ is optimal.

Interval Scheduling Problem

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$.
- Let us consider an optimal subset of courses $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ for $\mathcal{S}$. We can order them such that for all $1 \leq i < m$: $f(o_i) \leq s(o_{i+1})$.
- We want to prove that $|\mathcal{A}| = |\mathcal{O}|$ (i.e. $k = m$) (we do not ask $\mathcal{A} = \mathcal{O}$ as there might be multiple optimal solutions).
- We know that $k \leq m$ because $\mathcal{O}$ is optimal.
- To see $k \geq m$ we will prove that our algorithm always “stays ahead”: for all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Interval Scheduling Problem

We will prove that the previous algorithm is correct.
Our strategy is the following:

- Let $\mathcal{S}$ be an arbitrary input set of courses, for which the previous algorithm returns a subset of compatible courses $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$.
- Let us consider an optimal subset of courses $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ for $\mathcal{S}$. We can order them such that for all $1 \leq i < m$: $f(o_i) \leq s(o_{i+1})$.
- We want to prove that $|\mathcal{A}| = |\mathcal{O}|$ (i.e. $k = m$) (we do not ask $\mathcal{A} = \mathcal{O}$ as there might be multiple optimal solutions).
- We know that $k \leq m$ because $\mathcal{O}$ is optimal.
- To see $k \geq m$ we will prove that our algorithm always "stays ahead": for all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.
- The intuition is that at each step our algorithm has a better subproblem, as it works with courses in the interval $[f(a_r), \ldots]$ rather than in $[f(o_r), \ldots]$

Interval Scheduling Problem

Proposition

For all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Proposition

For all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Proof: (induction on r)

Base case ($r = 1$): true as a_1 is the task with lowest finishing time.

Proposition

For all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Proof: (induction on r)

Base case ($r = 1$): true as a_1 is the task with lowest finishing time.

Induction step ($r > 1$ and assume the claim is true for $r - 1$):

We have ordered the tasks in $\mathcal{O}$ so that $f(o_{r-1}) \leq s(o_r)$.
This, together with the HI $f(a_{r-1}) \leq f(o_{r-1})$, implies $f(a_{r-1}) \leq s(o_r)$.

Proposition

For all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Proof: (induction on r)

Base case ($r = 1$): true as a_1 is the task with lowest finishing time.

Induction step ($r > 1$ and assume the claim is true for $r - 1$):

We have ordered the tasks in $\mathcal{O}$ so that $f(o_{r-1}) \leq s(o_r)$.
This, together with the HI $f(a_{r-1}) \leq f(o_{r-1})$, implies $f(a_{r-1}) \leq s(o_r)$.

As $f(a_1) \leq \ldots \leq f(a_{r-1})$, we have o_r does not overlap with any of $a_1, \ldots, a_{r-1}$

Proposition

For all $1 \leq r \leq k$ we have $f(a_r) \leq f(o_r)$.

Proof: (induction on r)

Base case ($r = 1$): true as a_1 is the task with lowest finishing time.

Induction step ($r > 1$ and assume the claim is true for $r - 1$):

We have ordered the tasks in $\mathcal{O}$ so that $f(o_{r-1}) \leq s(o_r)$.
This, together with the HI $f(a_{r-1}) \leq f(o_{r-1})$, implies $f(a_{r-1}) \leq s(o_r)$.

As $f(a_1) \leq \ldots \leq f(a_{r-1})$, we have o_r does not overlap with any of $a_1, \ldots, a_{r-1}$

So when our algorithm selected a_r, one of the candidates to choose was o_r.
As it chooses the one with the smallest finishing time, we have $f(a_r) \leq f(o_r)$.
$\square$

Interval Scheduling Problem

Remember:

$\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ (the algorithm solution)

$\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ (optimal)

Proposition

$k = m$

Proof: (by contradiction)

Assume $k \neq m$.

Interval Scheduling Problem

Remember:

$\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ (the algorithm solution)

$\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ (optimal)

Proposition

$k = m$

Proof: (by contradiction)

Assume $k \neq m$.
As $\mathcal{O}$ is optimal, we have that $k \leq m$.
This, together with $k \neq m$, implies that $k < m$, and so o_{k+1} exists.

Interval Scheduling Problem

Remember:

$\mathcal{A} = \{a_1, a_2, \dots, a_k\}$ (the algorithm solution)

$\mathcal{O} = \{o_1, o_2, \dots, o_m\}$ (optimal)

Proposition

$k = m$

Proof: (by contradiction)

Assume $k \neq m$.
As $\mathcal{O}$ is optimal, we have that $k \leq m$.
This, together with $k \neq m$, implies that $k < m$, and so o_{k+1} exists.

The previous claim states that $f(a_k) \leq f(o_k)$.
Since $f(o_k) \leq s(o_{k+1})$, we have $f(a_k) \leq s(o_{k+1})$.

Interval Scheduling Problem

Remember:

$\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ (the algorithm solution)

$\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ (optimal)

Proposition

$k = m$

Proof: (by contradiction)

Assume $k \neq m$.
As $\mathcal{O}$ is optimal, we have that $k \leq m$.
This, together with $k \neq m$, implies that $k < m$, and so o_{k+1} exists.

The previous claim states that $f(a_k) \leq f(o_k)$.
Since $f(o_k) \leq s(o_{k+1})$, we have $f(a_k) \leq s(o_{k+1})$.

As $f(a_1) \leq \ldots \leq f(a_k)$, we have o_{k+1} does not overlap with any of $a_1, \ldots, a_k$.
So o_{k+1} was still available when our algorithm finished: contradiction!
(because it finishes when no available course exists).

$\square$

Interval Scheduling Problem

```
struct Task {
  int id;
  int starting;
  int finishing;
};

bool before (const Task& t1, const Task& t2) {
  return t1.finishing < t2.finishing;
}

int main(){
  vector<Task> tasks;
  int s, f, id = 1;
  while (cin >> s >> f) {
    tasks.push_back(Task{id,s,f});
    ++id;
  }
  sort(tasks.begin(),tasks.end(), before);
  vector<int> result = interval_scheduling(tasks);
  for (auto& x: result) cout << x << endl;
}
```

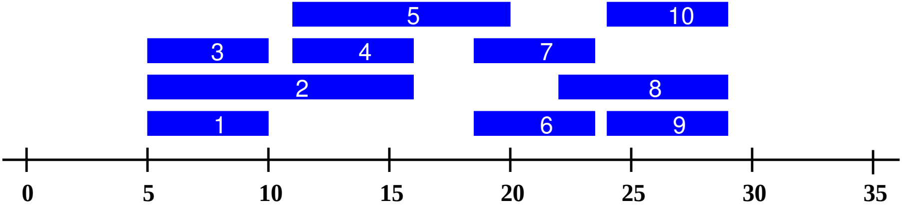
```
// PRE: tasks are sorted increasingly by finishing time
vector<int> interval_scheduling(const vector<Task>& tasks){
  int n = tasks.size();
  vector<int> res;         // Result
  int idx = 0;
  int last_f = INT_MIN;    // Acts as −∞

  while (idx < n) {
    // Look for non-overlapping task w/ smallest finishing
    while (tasks[idx].starting < last_f and idx < n) ++idx;
    if (idx < n) {  // Add tasks[idx] to solution
      res.push_back(tasks[idx].id);
      last_f = tasks[idx].finishing;
    }
  }
  return res;
}
```

Interval Partitioning Problem

- Imagine we have N courses that need to be allocated to some rooms
- Again, we know starting time $s(i)$ and finishing time $f(i)$ of each course
- The difference is that now:
  - we have to allocate all courses
  - we have an unlimited number of rooms
- We wish to find out the minimum number of rooms needed so that no two overlapping courses are assigned to the same room

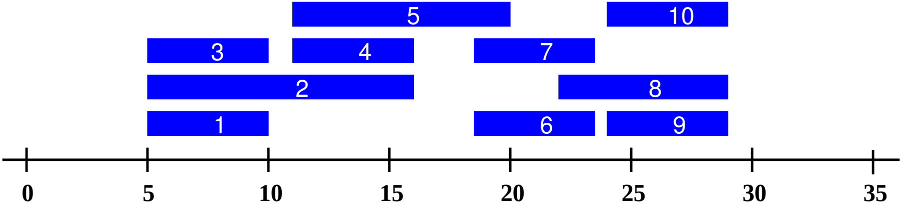
Interval Partitioning Problem

- Scheduling using 4 rooms:

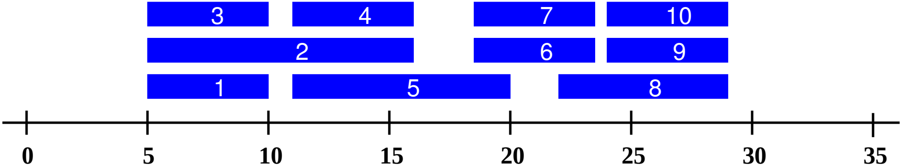


Interval Partitioning Problem

- Scheduling using 4 rooms:

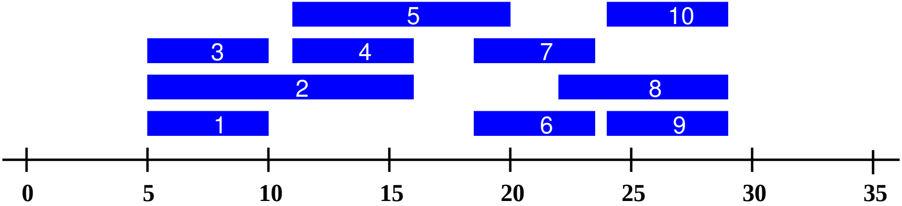


- Scheduling using 3 rooms:

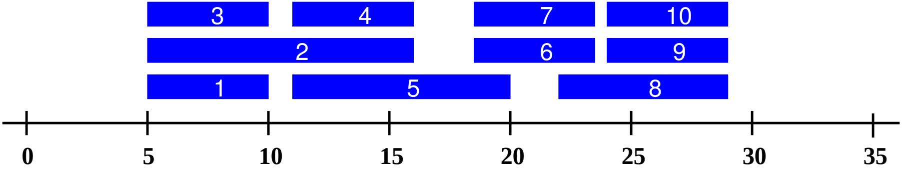


Interval Partitioning Problem

- Scheduling using 4 rooms:



- Scheduling using 3 rooms:



- Latter is optimal, as $\{1, 2, 3\}$ are overlapping and hence need 3 rooms.

- Let us define the depth of the problem as
  the maximum number of courses that overlap on any instant of time
- **CLAIM:** The number of rooms needed is $\geq$ the depth of the problem
- If d is the depth,
  can we find algorithm that schedules the courses with exactly d rooms?

Interval Partitioning Problem

- Let us define the depth of the problem as
  the maximum number of courses that overlap on any instant of time
- **CLAIM:** The number of rooms needed is $\geq$ the depth of the problem
- If d is the depth,
  can we find algorithm that schedules the courses with exactly d rooms?
- **GREEDY ALGORITHM:**
  1. Select d rooms (only these are available)
  2. Sort courses by starting time
  3. For each course c (in the order), select one room that is not used by the preceding courses (in the order) that overlap with c

Interval Partitioning Problem

- Let us define the depth of the problem as
  the maximum number of courses that overlap on any instant of time
- **CLAIM:** The number of rooms needed is $\geq$ the depth of the problem
- If d is the depth,
  can we find algorithm that schedules the courses with exactly d rooms?
- **GREEDY ALGORITHM:**
  1. Select d rooms (only these are available)
  2. Sort courses by starting time
  3. For each course c (in the order), select one room that is not used by the preceding courses (in the order) that overlap with c
- We should prove:
  - In the second step there is always at least one free room
  - No overlapping courses get the same rooms (easy!)

Interval Partitioning Problem

Proposition

In the second step of the algorithm there is always one free room to choose

Proposition

In the second step of the algorithm there is always one free room to choose

Proof: (by contradiction)

Assume it is not. Let c be the first course that has no free room to choose.

Proposition

In the second step of the algorithm there is always one free room to choose

Proof: (by contradiction)

Assume it is not. Let c be the first course that has no free room to choose.

This means that at time $s(c)$,
all d rooms are occupied by previous courses that are active at that moment.

Proposition

In the second step of the algorithm there is always one free room to choose

Proof: (by contradiction)

Assume it is not. Let c be the first course that has no free room to choose.

This means that at time $s(c)$,
all d rooms are occupied by previous courses that are active at that moment.

Those courses, together with c, form a set of $d + 1$ overlapping courses,
which is a contradiction with the definition of the depth d.

Interval Partitioning Problem

```
bool overlap(const Task& t1, const Task& t2) {
  return t1.finishing > t2.starting and
         t2.finishing > t1.starting;  }

bool before (const Task& t1, const Task& t2) {
  return t1.starting < t2.starting;  }

int main(){
  vector<Task> tasks;
  int s, f, id = 1;
  while (cin >> s >> f) {
    tasks.push_back(Task{id,s,f});
    ++id;
  }
  sort(tasks.begin(),tasks.end(), before);
  vector<int> labelling = interval_part(tasks);
  for (uint i = 0; i < tasks.size(); ++i)
  cout << "Task " <<  tasks[i].id << " gets room " <<
      labelling[i] << endl;
}
```

Interval Partitioning Problem

```
// PRE: tasks are sorted increasingly by starting time
vector<int> interval_part (const vector<Task>& tasks) {

  int n = tasks.size();
  vector<int> labelling(n,-1); // -1 for unassigned label

  for (int i = 0; i < n; ++i) {
    // Mark not available labels for the i-th task
    vector<int> available_label(n, true); // large UB
    for (int j = 0; j < i; ++j)
      if (overlap(tasks[i], tasks[j]))
         available_label[labelling[j]] = false;

    int l = 0;
    while (not available_label[l]) ++l;
    labelling[i] = l;
  }

  return labelling;
}
```

Lateness Minimization Problem

- Imagine we have a set of N tasks that request to use the same resource, of which we only have one
- For each task i we know its duration $du(i)$ and its deadline $d(i)$ (ideally, the task should be finished before this time)

Lateness Minimization Problem

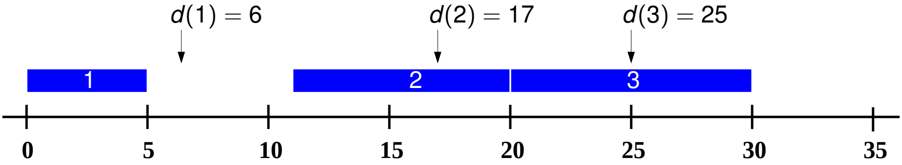
- Imagine we have a set of N tasks that request to use the same resource, of which we only have one
- For each task i we know its duration $du(i)$ and its deadline $d(i)$ (ideally, the task should be finished before this time)
- Given a task with starting time $s(i)$, we define its lateness as $\ell(i) = \max\{\ (s(i) + du(i)\,)\ -\ d(i),\ \ 0\}$.
- Our goal is to find starting times for all tasks so as to minimize the maximum lateness, i.e.

$$\min \max_{i=1}^{N} \ell(i)$$

- Imagine we have a set of N tasks that request to use the same resource, of which we only have one
- For each task i we know its duration $du(i)$ and its deadline $d(i)$ (ideally, the task should be finished before this time)
- Given a task with starting time $s(i)$, we define its lateness as $\ell(i) = \max\{\ (s(i) + du(i)\,)\ -\ d(i),\ \ 0\}$.
- Our goal is to find starting times for all tasks so as to minimize the maximum lateness, i.e.

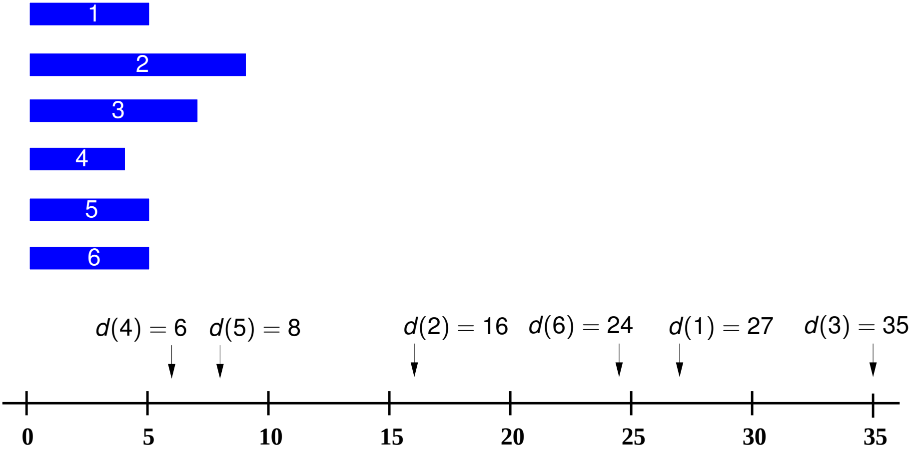
$$\min \max_{i=1}^{N} \ell(i)$$

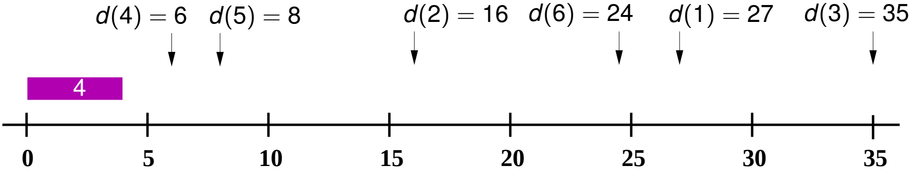
- Example:



- This non-optimal schedule has a maximum lateness of 5

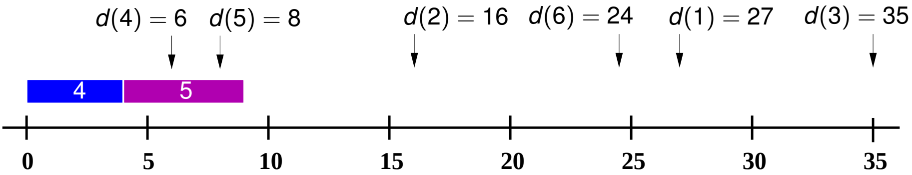
Lateness Minimization Problem





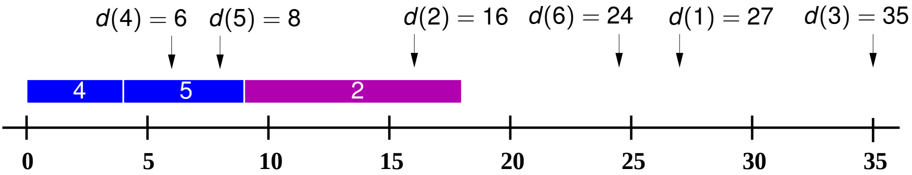
GREEDY ALGORITHM

- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.



GREEDY ALGORITHM

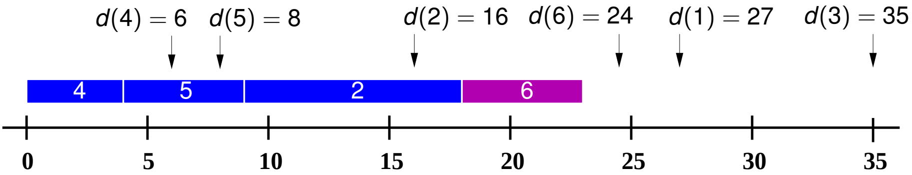
- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.



GREEDY ALGORITHM

- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.
  3. Choose task 2. Starts at 9, finishes at 18. Lateness is 2.

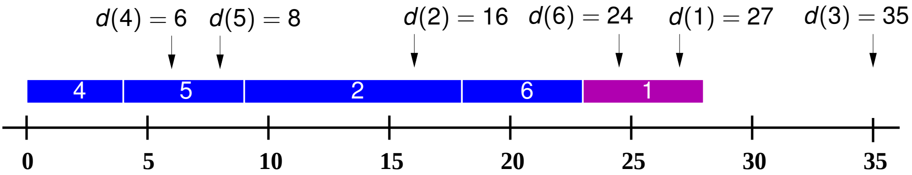
Lateness Minimization Problem



GREEDY ALGORITHM

- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.
  3. Choose task 2. Starts at 9, finishes at 18. Lateness is 2.
  4. Choose task 6. Starts at 18, finishes at 23. Lateness is 0.

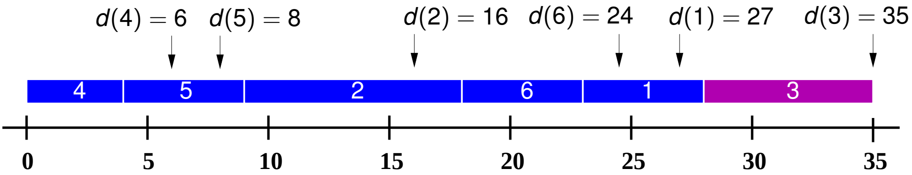
Lateness Minimization Problem



GREEDY ALGORITHM

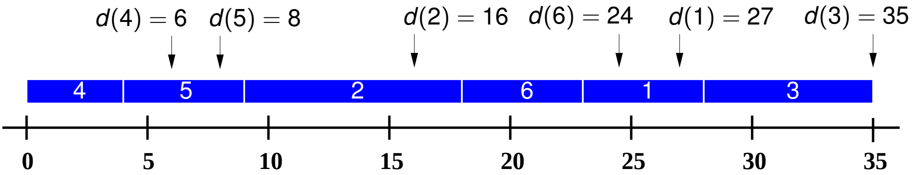
- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.
  3. Choose task 2. Starts at 9, finishes at 18. Lateness is 2.
  4. Choose task 6. Starts at 18, finishes at 23. Lateness is 0.
  5. Choose task 1. Starts at 23, finishes at 28. Lateness is 1.

Lateness Minimization Problem



GREEDY ALGORITHM

- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.
  3. Choose task 2. Starts at 9, finishes at 18. Lateness is 2.
  4. Choose task 6. Starts at 18, finishes at 23. Lateness is 0.
  5. Choose task 1. Starts at 23, finishes at 28. Lateness is 1.
  6. Choose task 3. Starts at 28, finishes at 35. Lateness is 0.



GREEDY ALGORITHM

- Idea: schedule first the tasks with earliest deadline
- Algorithm: while there are tasks to schedule, pick the one with earliest deadline and place it right after the task scheduled in the previous step.
- Algorithm execution:
  1. Choose task 4. Starts at 0, finishes at 4. Lateness is 0.
  2. Choose task 5. Starts at 4, finishes at 9. Lateness is 1.
  3. Choose task 2. Starts at 9, finishes at 18. Lateness is 2.
  4. Choose task 6. Starts at 18, finishes at 23. Lateness is 0.
  5. Choose task 1. Starts at 23, finishes at 28. Lateness is 1.
  6. Choose task 3. Starts at 28, finishes at 35. Lateness is 0.

     Maximum lateness is 2

Lateness Minimization Problem

We will prove that the previous algorithm produces optimal schedules.
Our proof strategy is the following:

- It is obvious that our algorithm produces schedules with no idle time, i.e. time in which the resource is unused but there are still pending tasks

Lateness Minimization Problem

We will prove that the previous algorithm produces optimal schedules.
Our proof strategy is the following:

- It is obvious that our algorithm produces schedules with no idle time, i.e. time in which the resource is unused but there are still pending tasks
- We call an inversion a pair of tasks (i, j) s.t. $d(i) > d(j)$ but $s(i) < s(j)$. It is clear that our algorithm produces schedules with no inversions

Lateness Minimization Problem

We will prove that the previous algorithm produces optimal schedules.
Our proof strategy is the following:

- It is obvious that our algorithm produces schedules with no idle time, i.e. time in which the resource is unused but there are still pending tasks
- We call an inversion a pair of tasks (i, j) s.t. $d(i) > d(j)$ but $s(i) < s(j)$. It is clear that our algorithm produces schedules with no inversions

1. We prove all schedules with no inversions and no idle time have the same maximum lateness

Lateness Minimization Problem

We will prove that the previous algorithm produces optimal schedules.
Our proof strategy is the following:

- It is obvious that our algorithm produces schedules with no idle time, i.e. time in which the resource is unused but there are still pending tasks
- We call an inversion a pair of tasks (i, j) s.t. $d(i) > d(j)$ but $s(i) < s(j)$. It is clear that our algorithm produces schedules with no inversions

1. We prove all schedules with no inversions and no idle time have the same maximum lateness
2. We prove there's an optimal schedule with no inversions and no idle time

Lateness Minimization Problem

Proposition

All schedules with no inversions and no idle time have the same max lateness

Proof: In a schedule with no inversions, if $d(i) > d(j)$ then $s(i) > s(j)$, i.e., task i necessarily follows task j.

Lateness Minimization Problem

Proposition

All schedules with no inversions and no idle time have the same max lateness

Proof: In a schedule with no inversions, if $d(i) > d(j)$ then $s(i) > s(j)$, i.e., task i necessarily follows task j.

Hence, the relative order between tasks in two schedules with no inversions is the same except maybe for tasks with the same deadline.

Lateness Minimization Problem

Proposition

All schedules with no inversions and no idle time have the same max lateness

Proof: In a schedule with no inversions, if $d(i) > d(j)$ then $s(i) > s(j)$, i.e., task i necessarily follows task j.

Hence, the relative order between tasks in two schedules with no inversions is the same except maybe for tasks with the same deadline.

Consider now two schedules with no inversions and no idle time.

In both schedules tasks with the same deadline d will appear consecutively (after all tasks with deadlines $< d$ and before all tasks with deadlines $> d$), but maybe in a different order.

Proposition

All schedules with no inversions and no idle time have the same max lateness

Proof: In a schedule with no inversions, if $d(i) > d(j)$ then $s(i) > s(j)$, i.e., task i necessarily follows task j.

Hence, the relative order between tasks in two schedules with no inversions is the same except maybe for tasks with the same deadline.

Consider now two schedules with no inversions and no idle time.

In both schedules tasks with the same deadline d will appear consecutively (after all tasks with deadlines $< d$ and before all tasks with deadlines $> d$), but maybe in a different order.

The maximum lateness over tasks with the same deadline d is defined by the last scheduled task with deadline d and is the same in both schedules.

Lateness Minimization Problem

Proposition

All schedules with no inversions and no idle time have the same max lateness

Proof: In a schedule with no inversions, if $d(i) > d(j)$ then $s(i) > s(j)$, i.e., task i necessarily follows task j.

Hence, the relative order between tasks in two schedules with no inversions is the same except maybe for tasks with the same deadline.

Consider now two schedules with no inversions and no idle time.

In both schedules tasks with the same deadline d will appear consecutively (after all tasks with deadlines $< d$ and before all tasks with deadlines $> d$), but maybe in a different order.

The maximum lateness over tasks with the same deadline d is defined by the last scheduled task with deadline d and is the same in both schedules.

This happens for all deadlines d,
so the two schedules have the same maximum lateness.

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof:

Let us consider an optimal schedule S.

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof:

Let us consider an optimal schedule S.

If it has idle time, we can transform it into an optimal one with no idle time by shifting tasks to the left.

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof:

Let us consider an optimal schedule S.

If it has idle time, we can transform it into an optimal one with no idle time by shifting tasks to the left.

So we can assume that S is an optimal schedule with no idle time.

Lateness Minimization Problem

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof:

Let us consider an optimal schedule S.

If it has idle time, we can transform it into an optimal one with no idle time by shifting tasks to the left.

So we can assume that S is an optimal schedule with no idle time.

If it has no inversions, we are done.

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof (continued):

Otherwise let us pick an inversion (a, b).

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof (continued):

Otherwise let us pick an inversion (a, b).

Let $a_0 = a, a_1, a_2, \ldots, a_{n-1}, a_n = b$ be the sequence of tasks between a and b.

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof (continued):

Otherwise let us pick an inversion (a, b).

Let $a_0 = a, a_1, a_2, \ldots, a_{n-1}, a_n = b$ be the sequence of tasks between a and b.

Since $d(a) > d(b)$ there has to be at least a pair of consecutive tasks (a_k, a_{k+1}) such that $d(a_k) > d(a_{k+1})$. Let us denote one such pair by (i, j).

Lateness Minimization Problem

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof (continued):

Otherwise let us pick an inversion (a, b).

Let $a_0 = a, a_1, a_2, \ldots, a_{n-1}, a_n = b$ be the sequence of tasks between a and b.

Since $d(a) > d(b)$ there has to be at least a pair of consecutive tasks (a_k, a_{k+1}) such that $d(a_k) > d(a_{k+1})$. Let us denote one such pair by (i, j).

We will now swap i and j in the schedule and the resulting schedule S'

- will have one less inversion
- will have a maximum lateness no larger than S

Proposition

There is an optimal schedule with no inversions and no idle time.

Proof (continued):

Otherwise let us pick an inversion (a, b).

Let $a_0 = a, a_1, a_2, \ldots, a_{n-1}, a_n = b$ be the sequence of tasks between a and b.

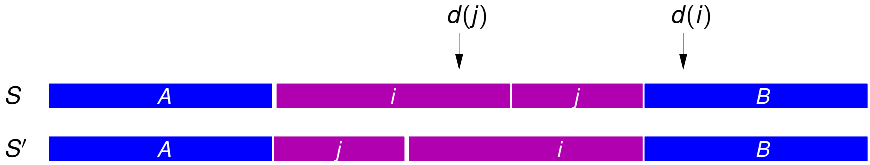
Since $d(a) > d(b)$ there has to be at least a pair of consecutive tasks (a_k, a_{k+1}) such that $d(a_k) > d(a_{k+1})$. Let us denote one such pair by (i, j).

We will now swap i and j in the schedule and the resulting schedule S'

- will have one less inversion
- will have a maximum lateness no larger than S

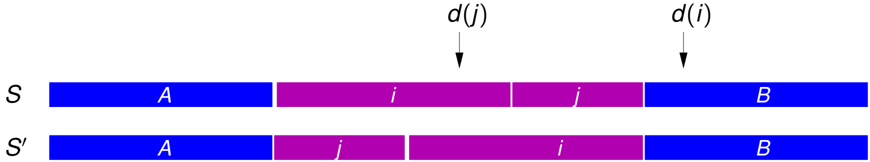
By repeating this swapping process,
we will end up having an optimal schedule with no inversions.

Proof (continued):



- S' has one less inversion than S:

Proof (continued):



- S' has one less inversion than S:

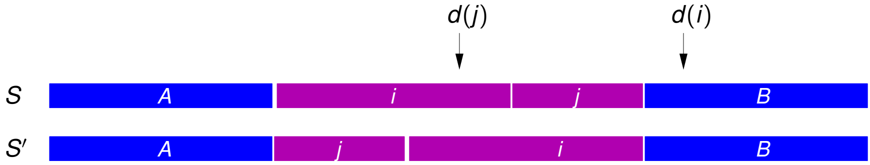
  No new inversions among tasks in A or B are created.

  Inversions in S' of the form (k, j), (k, i) with k in A belong to S too.

  Inversions in S' of the form (i, k), (j, k) with k in B belong to S too.

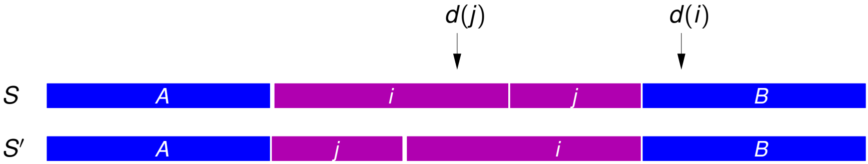
Lateness Minimization Problem

Proof (continued):



- The maximum lateness of S' is no larger than the one of S:

Proof (continued):

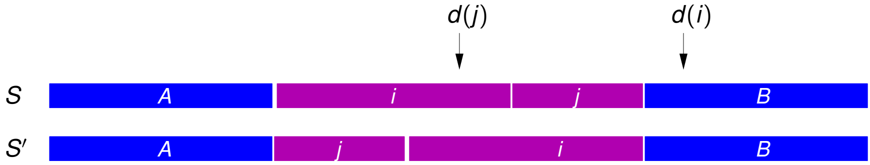


- The maximum lateness of S' is no larger than the one of S:

  In S' only the finishing times of i and j are changed.

  As the finishing time of j is smaller, only i may have a larger lateness

Proof (continued):



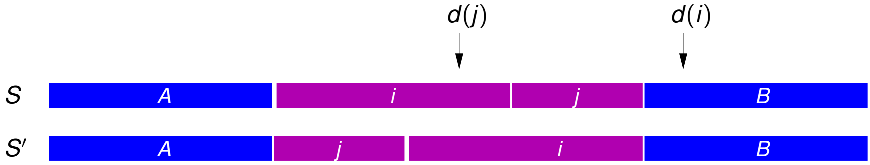
- The maximum lateness of S' is no larger than the one of S:

  In S' only the finishing times of i and j are changed.

  As the finishing time of j is smaller, only i may have a larger lateness

  If i is late in S', its lateness is $s'(i) + du(i) - d(i) = s(j) + du(j) - d(i)$.

Proof (continued):



- The maximum lateness of S' is no larger than the one of S:

  In S' only the finishing times of i and j are changed.

  As the finishing time of j is smaller, only i may have a larger lateness

  If i is late in S', its lateness is $s'(i) + du(i) - d(i) = s(j) + du(j) - d(i)$.

  But since $d(i) > d(j)$, we have i cannot be later in S' than j in S:

$$\ell'(i) = s(j) + du(j) - d(i) < s(j) + du(j) - d(j) = \ell(j).$$

$\square$

```
struct Task {
  int id;
  int duration;
  int deadline;
  Task(int i, int du, int de):
  id(i), duration(du), deadline(de){}  };

bool before (const Task& t1, const Task& t2) {
  return t1.deadline < t2.deadline; }

int main(){
  vector<Task> tasks;
  int du, de, id = 1;
  while (cin >> du >> de) {
    tasks.push_back(Task(id,du,de));  ++id;}
  sort(tasks.begin(), tasks.end(), before);
  vector<int> starting_time = lateness(tasks);
  for (uint i = 0; i < tasks.size(); ++i)
     cout << "Task " <<  tasks[i].id << " starts at " <<
         starting_time[i] << endl;
}
```

Lateness Minimization Problem

```
// PRE: tasks is sorted increasingly by deadline
vector<int> lateness(const vector<Task>& tasks) {
  int n = tasks.size();
  vector<int> starting_time(n);

  int t = 0;
  for (int i = 0; i < n; ++i) {
    starting_time[i] = t;
    t += tasks[i].duration;
  }

  return starting_time;
}
```

Dijkstra's Algorithm

Dijkstra's algorithm for shortest paths

Input: Graph $G = (V, E)$, directed or not;
non-negative weights in the edges $\{w(u, v) \mid (u, v) \in E\}$;
initial vertex s;

Output: For all vertex u reachable from s,
$dist(u) =$ distance from s to u (denoted in the following by $\delta(s, u)$)

DIJKSTRA (G,w,s)
  for all vertex $u \in V$ **do** $dist(u) = \infty$
  $dist(s) = 0$
  H =create-priority-queue(V) (using *dist* as keys)
  while not empty(H)
    u =remove-min(H) $\leftarrow$ This is the **greedy** part
    for all edge $(u, v) \in E$ **do**
      if $dist(v) > dist(u) + w(u, v)$ (relax edge)
        $dist(v) = dist(u) + w(u, v)$
        decrement-key(H, v)

Proof of Dijkstra's Algorithm

Proposition

If $dist(u) < \infty$, then $dist(u)$ corresponds to the length of some path from s to u

Proof of Dijkstra's Algorithm

Proposition

If *dist*$(u) < \infty$, then *dist*(u) corresponds to the length of some path from s to u

Proof:
We will prove that whenever *dist* is updated, the property is preserved.
We will do that by induction on the step in which the update was made.

Proof of Dijkstra's Algorithm

Proposition

If $dist(u) < \infty$, then $dist(u)$ corresponds to the length of some path from s to u

Proof:
We will prove that whenever *dist* is updated, the property is preserved.
We will do that by induction on the step in which the update was made.

The base case is the first update: $dist(s) = 0$. The property clearly holds.

Proof of Dijkstra's Algorithm

Proposition

If *dist*$(u) < \infty$, then *dist*(u) corresponds to the length of some path from s to u

Proof:
We will prove that whenever *dist* is updated, the property is preserved.
We will do that by induction on the step in which the update was made.

The base case is the first update: $dist(s) = 0$. The property clearly holds.

Consider now the k-th update.
Assume the property holds up to the $k-1$-th update.
The k-th update is done with the command

$$dist(v) = dist(u) + w(u, v)$$

Since $dist(u) < +\infty$ corresponds to the length of some path from s to u, and there is an edge $u \to v$ with weight $w(u, v)$, the property is true. □

Proof of Dijkstra's Algorithm

Proposition

If *dist*$(u) < \infty$, then *dist*(u) corresponds to the length of some path from s to u

Proof:

We will prove that whenever *dist* is updated, the property is preserved.
We will do that by induction on the step in which the update was made.

The base case is the first update: $dist(s) = 0$. The property clearly holds.

Consider now the k-th update.
Assume the property holds up to the $k-1$-th update.
The k-th update is done with the command

$$dist(v) = dist(u) + w(u,v)$$

Since $dist(u) < +\infty$ corresponds to the length of some path from s to u, and there is an edge $u \to v$ with weight $w(u,v)$, the property is true. □

Corollary

For all vertices u, it always holds that $dist(u) \geq \delta(s,u)$.

Proof of Dijkstra's Algorithm

Proposition

When a vertex u is removed from H, it holds that $dist(u) = \delta(s, u)$

Proof of Dijkstra's Algorithm

Proposition

When a vertex u is removed from H, it holds that $dist(u) = \delta(s, u)$

Proof (by contradiction):

Let us assume it is not the case.
Let u be the first vertex that is removed from H with $dist(u) \neq \delta(s, u)$.

Proof of Dijkstra's Algorithm

Proposition

When a vertex u is removed from H, it holds that $dist(u) = \delta(s, u)$

Proof (by contradiction):

Let us assume it is not the case.
Let u be the first vertex that is removed from H with $dist(u) \neq \delta(s, u)$.

By the previous corollary, $dist(u) > \delta(s, u)$

Proposition

When a vertex u is removed from H, it holds that $dist(u) = \delta(s, u)$

Proof (by contradiction):

Let us assume it is not the case.
Let u be the first vertex that is removed from H with $dist(u) \neq \delta(s, u)$.

By the previous corollary, $dist(u) > \delta(s, u)$

We also know that $u \neq s$.

Proof of Dijkstra's Algorithm

Proposition

When a vertex u is removed from H, it holds that $dist(u) = \delta(s, u)$

Proof (by contradiction):

Let us assume it is not the case.
Let u be the first vertex that is removed from H with $dist(u) \neq \delta(s, u)$.

By the previous corollary, $dist(u) > \delta(s, u)$

We also know that $u \neq s$.

Now let us consider the shortest path from s to u.

Before we removed u from H, this path went from $s \notin H$ to $u \in H$.

Let $x \to y$ be the first edge in this path from a vertex not in H to a vertex in H.
(Note that it could be $s = x$ or $y = u$).

This path is of the form $s \rightsquigarrow x \to y \rightsquigarrow u$.

Proof (continued):

Let us consider the state of the algorithm right after x was removed from H. When processing edge $x \to y$, two things may happen:

- If $dist(y) \leq dist(x) + w(x, y)$, nothing is done
- If $dist(y) > dist(x) + w(x, y)$, the edge is relaxed and $dist$ is updated so that $dist(y) = dist(x) + w(x, y)$.

So after processing edge (x, y) we have $dist(y) \leq dist(x) + w(x, y)$.

Proof (continued):

Let us consider the state of the algorithm right after x was removed from H. When processing edge $x \to y$, two things may happen:

- If $dist(y) \leq dist(x) + w(x, y)$, nothing is done
- If $dist(y) > dist(x) + w(x, y)$, the edge is relaxed and $dist$ is updated so that $dist(y) = dist(x) + w(x, y)$.

So after processing edge (x, y) we have $dist(y) \leq dist(x) + w(x, y)$.

This property is preserved from that moment on:

- $dist(x)$ is not changed ($dist(x) = \delta(s, x)$, so it cannot decrease)
- $dist(y)$ can only decrease

Proof of Dijkstra's Algorithm

Proof (continued):

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u, it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.
So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.

Proof of Dijkstra's Algorithm

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u,
it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.

So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.

And as $dist(y) \geq \delta(s, y)$ by a previous corollary, $dist(y) = \delta(s, y)$.

Proof of Dijkstra's Algorithm

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u,
it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.
So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.
And as $dist(y) \geq \delta(s, y)$ by a previous corollary, $dist(y) = \delta(s, y)$.
Just before removing u from H we had $dist(u) > \delta(s, u)$, so $y \neq u$.

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u,
it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.
So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.
And as $dist(y) \geq \delta(s, y)$ by a previous corollary, $dist(y) = \delta(s, y)$.
Just before removing u from H we had $dist(u) > \delta(s, u)$, so $y \neq u$.
Finally, since the shortest path from s to u is $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$,
we know that $\delta(s, y) \leq \delta(s, u)$ (note that weights are non-negative).

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u, it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.

So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.

And as $dist(y) \geq \delta(s, y)$ by a previous corollary, $dist(y) = \delta(s, y)$.

Just before removing u from H we had $dist(u) > \delta(s, u)$, so $y \neq u$.

Finally, since the shortest path from s to u is $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, we know that $\delta(s, y) \leq \delta(s, u)$ (note that weights are non-negative).

Altogether,

$$dist(y) = \delta(s, y) \leq \delta(s, u) < dist(u)$$

from which we conclude $dist(y) < dist(u)$ just before u was removed from H.

Proof (continued):

Now consider the state of the algorithm right before u was removed from H.
We have that $dist(y) \leq dist(x) + w(x, y)$.

Since $s \rightsquigarrow x \rightarrow y$ is part of the shortest path between s and u,
it is the shortest path between s and y and so $\delta(s, y) = \delta(s, x) + w(x, y)$.
So $dist(y) \leq dist(x) + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.
And as $dist(y) \geq \delta(s, y)$ by a previous corollary, $dist(y) = \delta(s, y)$.
Just before removing u from H we had $dist(u) > \delta(s, u)$, so $y \neq u$.
Finally, since the shortest path from s to u is $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$,
we know that $\delta(s, y) \leq \delta(s, u)$ (note that weights are non-negative).
Altogether,

$$dist(y) = \delta(s, y) \leq \delta(s, u) < dist(u)$$

from which we conclude $dist(y) < dist(u)$ just before u was removed from H.
But then the algorithm should have chosen y instead of u: contradiction! $\square$

Proof of Dijkstra's Algorithm

Theorem:

Dijkstra's Algorithm computes the distances from s to all reachable vertices

Proof of Dijkstra's Algorithm

Theorem:

Dijkstra's Algorithm computes the distances from s to all reachable vertices

Proof: We have already proved the following facts:

- When we remove vertex u from H it holds that $dist(u) = \delta(s, u)$
- It always holds that $dist(u) \geq \delta(s, u)$
- Whenever we update $dist(u)$, we strictly decrease it

Proof of Dijkstra's Algorithm

Theorem:

Dijkstra's Algorithm computes the distances from s to all reachable vertices

Proof: We have already proved the following facts:

- When we remove vertex u from H it holds that $dist(u) = \delta(s, u)$
- It always holds that $dist(u) \geq \delta(s, u)$
- Whenever we update $dist(u)$, we strictly decrease it

So once an element u is removed from H,
its value $dist(u)$ cannot be changed and remains equal to $\delta(s, u)$.

Proof of Dijkstra's Algorithm

Theorem:

Dijkstra's Algorithm computes the distances from s to all reachable vertices

Proof: We have already proved the following facts:

- When we remove vertex u from H it holds that $dist(u) = \delta(s, u)$
- It always holds that $dist(u) \geq \delta(s, u)$
- Whenever we update $dist(u)$, we strictly decrease it

So once an element u is removed from H,
its value $dist(u)$ cannot be changed and remains equal to $\delta(s, u)$.

But at the end of the algorithm, H is empty.
As all vertices are eventually removed, all distances are duly computed $\square$

General Algorithm

Let $G = (V, E)$ be an undirected connected graph with weights $\omega : E \to \mathbb{R}$.
Here we will consider that any set of edges $A \subseteq E$ induces a subgraph of G

General Algorithm

Let $G = (V, E)$ be an undirected connected graph with weights $\omega : E \rightarrow \mathbb{R}$.
Here we will consider that any set of edges $A \subseteq E$ induces a subgraph of G

Definition

A spanning tree of G is a subgraph $A \subseteq E$ of G such that

- it is a tree (i.e., connected and acyclic)
- it contains all vertices of G

General Algorithm

Let $G = (V, E)$ be an undirected connected graph with weights $\omega : E \to \mathbb{R}$.
Here we will consider that any set of edges $A \subseteq E$ induces a subgraph of G

Definition

A spanning tree of G is a subgraph $A \subseteq E$ of G such that

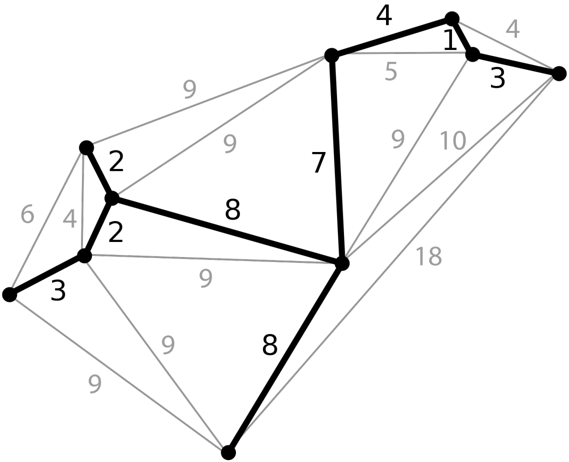
- it is a tree (i.e., connected and acyclic)
- it contains all vertices of G

Definition

A minimum spanning tree (MST) of G is a spanning tree A of G such that its total weight

$$\omega(A) = \sum_{e \in A} \omega(e)$$

is minimum among all spanning trees of G



Source: `https://commons.wikimedia.org/wiki/File:Minimum_spanning_tree.svg`

General Algorithm

There are many different algorithms for computing MST's.

They typically follow the scheme:

```
A = ∅;
Cand = E;
while (|A| ≠ |V| − 1) {
  choose e ∈ Cand that does not close a cycle in A;
  A = A ∪ {e};
  Cand = Cand - {e}
}
```

General Algorithm

Definition

A set $A \subseteq E$ is promising if A is a subset of a MST of G

General Algorithm

Definition

A set $A \subseteq E$ is promising if A is a subset of a MST of G

Definition

A cut in a graph G is a partition of the set of vertices V, i.e.,
a pair (C, C') such that $C, C' \neq \emptyset$ and

- $C \cup C' = V$
- $C \cap C' = \emptyset$

General Algorithm

Definition

A set $A \subseteq E$ is promising if A is a subset of a MST of G

Definition

A cut in a graph G is a partition of the set of vertices V, i.e.,
a pair (C, C') such that $C, C' \neq \emptyset$ and

- $C \cup C' = V$
- $C \cap C' = \emptyset$

Definition

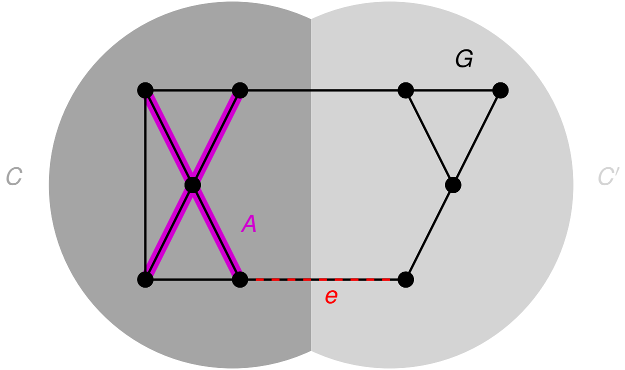
An edge e respects a cut (C, C') if the ends of e belong both to C or to C'; otherwise, we say that e crosses the cut.

General Algorithm

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge of minimum weight among those that cross the cut (C, C').
Then $A \cup \{e\}$ is promising.

General Algorithm



General Algorithm

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge of minimum weight among those that cross the cut (C, C').
Then $A \cup \{e\}$ is promising.

This theorem gives a recipe for designing algorithms for MST:

1. start with an empty set of edges A
2. define a cut of G which is respected by A
3. choose the edge e with minimum weight among those that cross the cut
4. add e to A

General Algorithm

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge of minimum weight among those that cross the cut (C, C').
Then $A \cup \{e\}$ is promising.

This theorem gives a recipe for designing algorithms for MST:

1. start with an empty set of edges A
2. define a cut of G which is respected by A
3. choose the edge e with minimum weight among those that cross the cut
4. add e to A

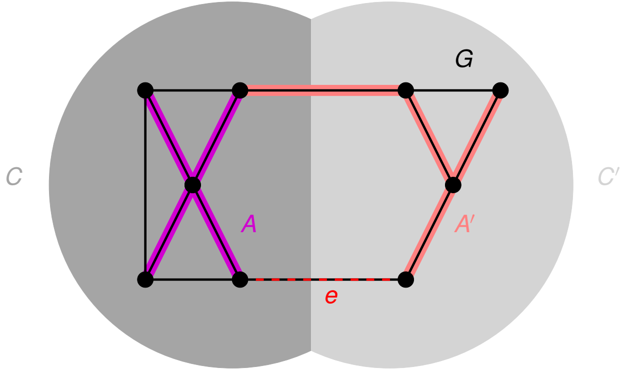
Any algorithm that follows this scheme is automatically correct.

Note that step 3 chooses the edge greedily
(the one with minimum weight among those that cross the cut)

Proof of the theorem

Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

General Algorithm



General Algorithm

Proof of the theorem

Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

If $e \in A'$ then $A \cup \{e\}$ is promising and the theorem holds.

General Algorithm

Proof of the theorem

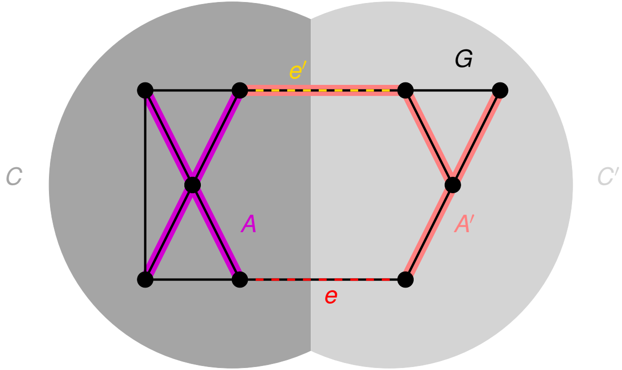
Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

If $e \in A'$ then $A \cup \{e\}$ is promising and the theorem holds.

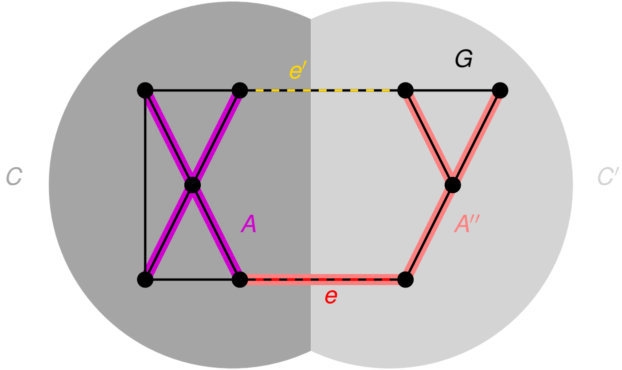
Now let us assume that $e \notin A'$.

As A respects the cut, there is an edge $e' \in A' - A$ that crosses the cut (otherwise, A' would not be connected).

General Algorithm



General Algorithm



Proof of the theorem

Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

If $e \in A'$ then $A \cup \{e\}$ is promising and the theorem holds.

Now let us assume that $e \notin A'$.

As A respects the cut, there is an edge $e' \in A' - A$ that crosses the cut (otherwise, A' would not be connected).

The subgraph $A'' = A' - \{e'\} \cup \{e\}$ is a spanning tree and

$$\omega(A') \leq \omega(A'') = \omega(A') - \omega(e') + \omega(e)$$

Hence $\omega(e') \leq \omega(e)$.

General Algorithm

Proof of the theorem

Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

If $e \in A'$ then $A \cup \{e\}$ is promising and the theorem holds.

Now let us assume that $e \notin A'$.

As A respects the cut, there is an edge $e' \in A' - A$ that crosses the cut (otherwise, A' would not be connected).

The subgraph $A'' = A' - \{e'\} \cup \{e\}$ is a spanning tree and

$$\omega(A') \leq \omega(A'') = \omega(A') - \omega(e') + \omega(e)$$

Hence $\omega(e') \leq \omega(e)$.

But by definition of e, we have $\omega(e') \geq \omega(e)$, and therefore $\omega(A') = \omega(A'')$.

General Algorithm

Proof of the theorem

Let A' be an MST such that $A \subseteq A'$ (which exists since A is promising).

If $e \in A'$ then $A \cup \{e\}$ is promising and the theorem holds.

Now let us assume that $e \notin A'$.

As A respects the cut, there is an edge $e' \in A' - A$ that crosses the cut (otherwise, A' would not be connected).

The subgraph $A'' = A' - \{e'\} \cup \{e\}$ is a spanning tree and

$$\omega(A') \leq \omega(A'') = \omega(A') - \omega(e') + \omega(e)$$

Hence $\omega(e') \leq \omega(e)$.

But by definition of e, we have $\omega(e') \geq \omega(e)$, and therefore $\omega(A') = \omega(A'')$.

Since A'' is a MST, $A \cup \{e\}$ is promising.

Prim's Algorithm

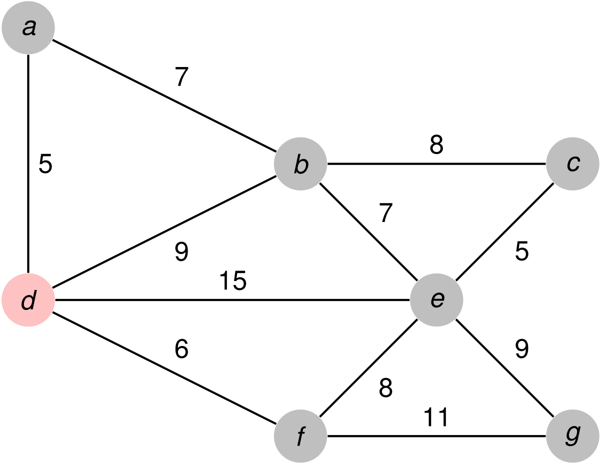
In Prim's Algorithm (also known as Prim-Jarník's Algorithm),
we keep a subset of visited vertices.

The set of vertices is thus partitioned into visited and non-visited vertices.

Each iteration of the algorithm chooses an edge of minimum weight among those that join a visited vertex and a non-visited one.

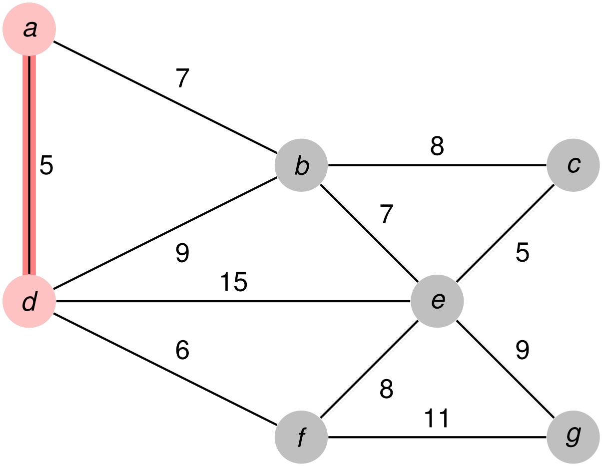
By the theorem, the algorithm is correct.

Prim's Algorithm



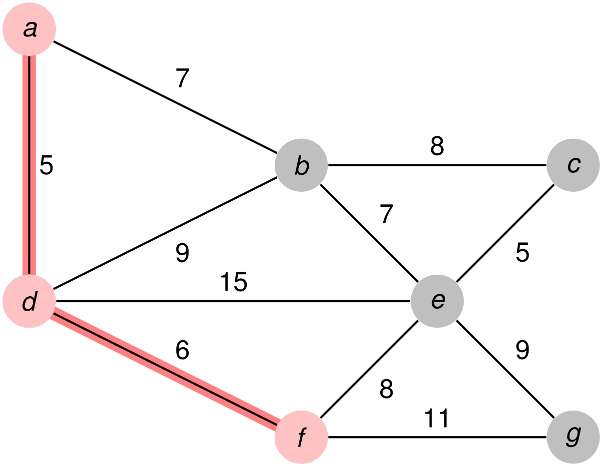
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm



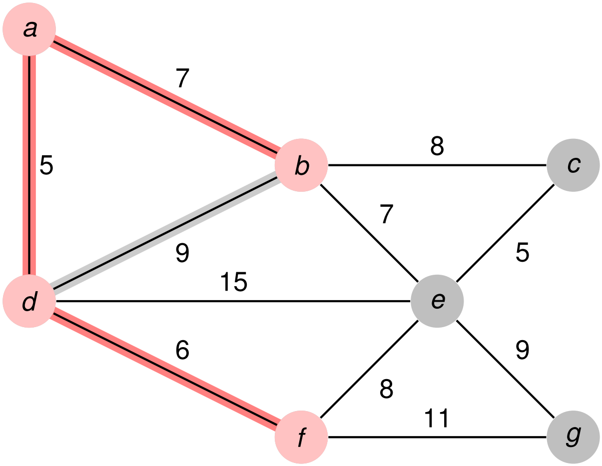
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm



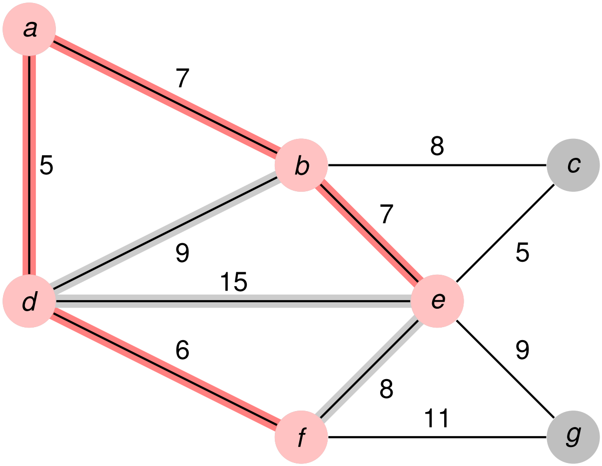
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm



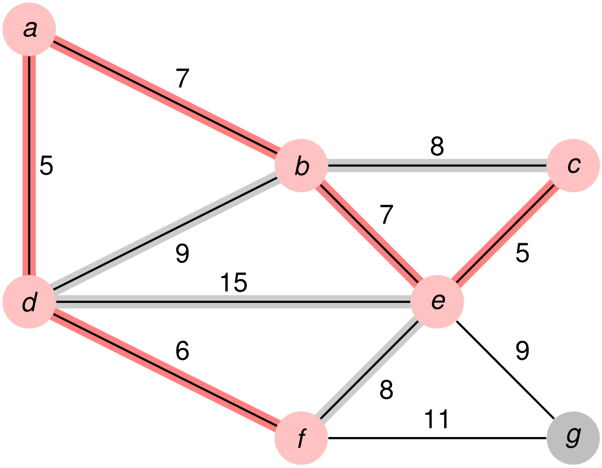
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm



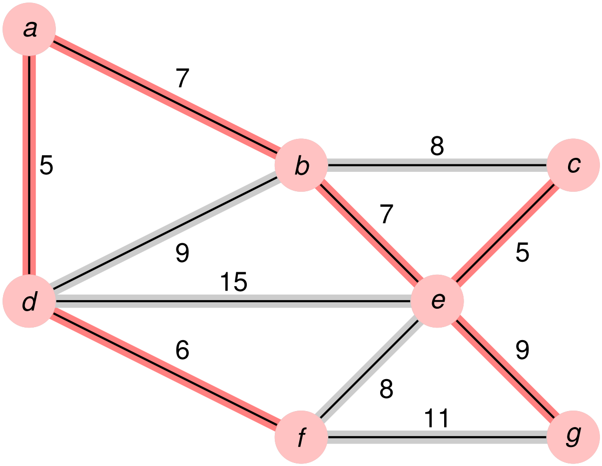
Font: www.texample.net/tikz/examples/author/kjell-magne-fauske

Prim's Algorithm



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim’s Algorithm



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

```
typedef pair< double, pair<int, int> > WEdge;
typedef vector< vector< pair<double, int> > > WGraph;

int Prim(const WGraph& G, vector<pair<int,int>>& MST) {
  vector<bool> used(G.size(), false);
  priority_queue<WEdge,vector<WEdge>,greater<WEdge>> Q;
  used[0] = true;
  for (auto& x : G[0]) Q.push({x.first,{0,x.second}});
  int totalWeight = 0; uint sz = 1;
  while (sz < G.size()) {
    int u = Q.top().second.first;
    int v = Q.top().second.second;
    double wEdge = Q.top().first;  Q.pop();
    if (not used[v]) {
      used[v] = true;
      MST.push_back({u,v});
      totalWeight += wEdge; ++sz;
      for (auto e : G[v]) Q.push({e.first, {v, e.second}});
  }  }
  return totalWeight; }
```

Prim's Algorithm

Theorem

Prim's Algorithm runs in $O(m \log n)$ time

Proof:

In the implementation, the cost of the loop dominates overall cost.

Prim's Algorithm

Theorem

Prim's Algorithm runs in $O(m \log n)$ time

Proof:

In the implementation, the cost of the loop dominates overall cost.

There are $O(m)$ iterations.
The choice of the edge at each iteration takes time $O(\log m)$.
In total choosing an edge has cost $O(m \log m)$.

Prim's Algorithm

Theorem

Prim's Algorithm runs in $O(m \log n)$ time

Proof:

In the implementation, the cost of the loop dominates overall cost.

There are $O(m)$ iterations.
The choice of the edge at each iteration takes time $O(\log m)$.
In total choosing an edge has cost $O(m \log m)$.

Each vertex is marked exactly once.
When vertex x is visited, adding new candidates costs $O(\deg(x) \log m)$.

Theorem

Prim's Algorithm runs in $O(m \log n)$ time

Proof:

In the implementation, the cost of the loop dominates overall cost.

There are $O(m)$ iterations.
The choice of the edge at each iteration takes time $O(\log m)$.
In total choosing an edge has cost $O(m \log m)$.

Each vertex is marked exactly once.
When vertex x is visited, adding new candidates costs $O(\deg(x)\ \log m)$.

In total:

$$O(m \log m) + \sum_{x \in V} O(\deg(x)\ \log m) = O(m \log m) = O(m \log n)$$

□

Kruskal's Algorithm

In Kruskal's Algorithm, at each step we keep:

- a set of forests *SF* (our MST in construction)
- a set of pending edges *P* candidates to be added to *SF*

Kruskal's Algorithm

In Kruskal's Algorithm, at each step we keep:

- a set of forests *SF* (our MST in construction)
- a set of pending edges *P* candidates to be added to *SF*

Initially *SF* is empty and *P* contains all edges in the graph.
At each step, we add to *SF* the edge *e* in *P* of minimum weight that does not create a cycle in *SF*, and remove *e* from *P*.

Kruskal's Algorithm

In Kruskal's Algorithm, at each step we keep:

- a set of forests SF (our MST in construction)
- a set of pending edges P candidates to be added to SF

Initially SF is empty and P contains all edges in the graph.
At each step, we add to SF the edge e in P of minimum weight that does not create a cycle in SF, and remove e from P.

Next we will see the algorithm is correct, as it follows the previous recipe:

1. start with an empty set of edges A
2. define a cut of G which is respected by A
3. choose the edge e with minimum weight among those that cross the cut
4. add e to A

We only need to determine what is the cut of Kruskal's algorithm

Kruskal's Algorithm

Let $G = (V, E)$ be the graph for which we want to find an MST.

At each step, consider $\{u, v\}$ the edge added by Kruskal
(i.e. the edge in P of minimum weight that does not create a cycle in SF)

Kruskal's Algorithm

Let $G = (V, E)$ be the graph for which we want to find an MST.

At each step, consider $\{u, v\}$ the edge added by Kruskal
(i.e. the edge in P of minimum weight that does not create a cycle in SF)

We define the cut as follows:

$$\begin{aligned} C &= \{w \in V \mid u \text{ has a path to } w \text{ in } SF\} \\ C' &= V \setminus C \end{aligned}$$

$\{u, v\}$ clearly crosses the cut, as we know it does not create a cycle in SF.

Kruskal's Algorithm

Let $G = (V, E)$ be the graph for which we want to find an MST.

At each step, consider $\{u, v\}$ the edge added by Kruskal
(i.e. the edge in P of minimum weight that does not create a cycle in SF)

We define the cut as follows:

$$\begin{array}{rcl} C & = & \{w \in V \mid u \text{ has a path to } w \text{ in } SF\} \\ C' & = & V \setminus C \end{array}$$

$\{u, v\}$ clearly crosses the cut, as we know it does not create a cycle in SF.

In general, any edge e crosses the cut iff it does not create a cycle in SF.

Let $G = (V, E)$ be the graph for which we want to find an MST.

At each step, consider $\{u, v\}$ the edge added by Kruskal
(i.e. the edge in P of minimum weight that does not create a cycle in SF)

We define the cut as follows:

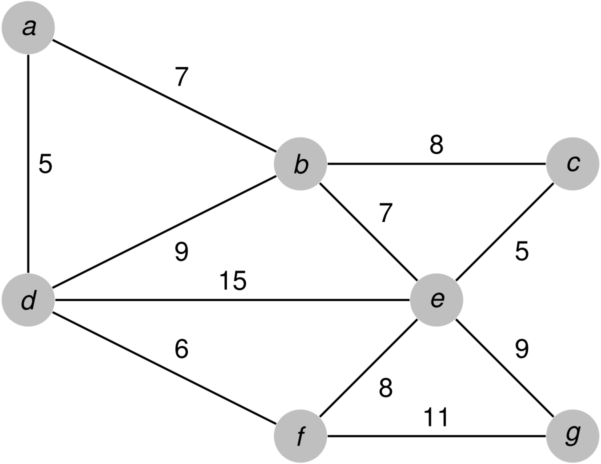
$$\begin{array}{rcl} C & = & \{w \in V \mid u \text{ has a path to } w \text{ in } SF\} \\ C' & = & V \setminus C \end{array}$$

$\{u, v\}$ clearly crosses the cut, as we know it does not create a cycle in SF.

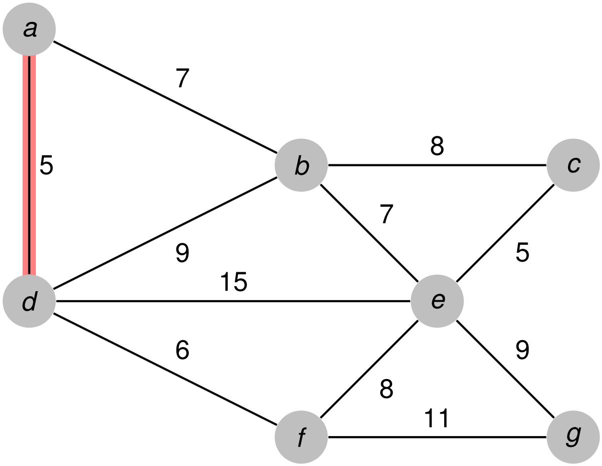
In general, any edge e crosses the cut iff it does not create a cycle in SF.

So $\{u, v\}$ is the edge with minimum weight among those crossing the cut.

Kruskal's Algorithm

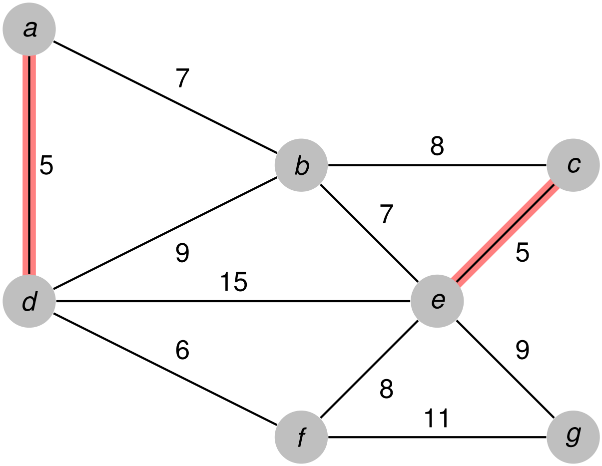


Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`



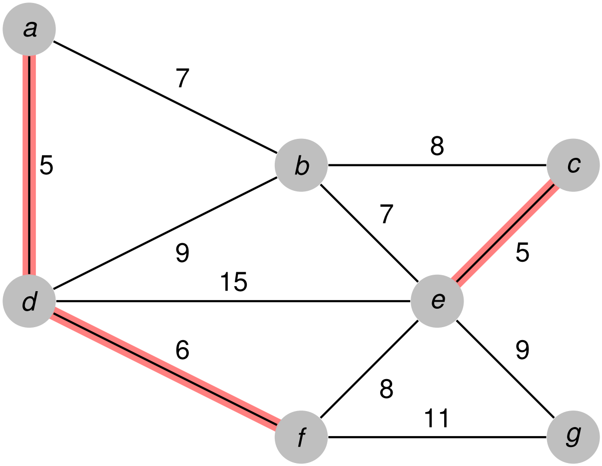
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm



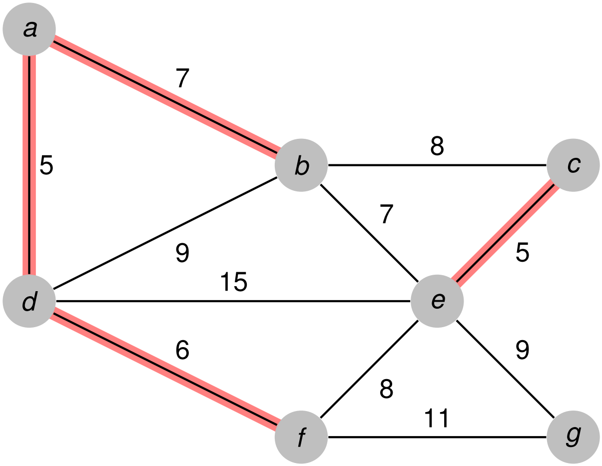
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm



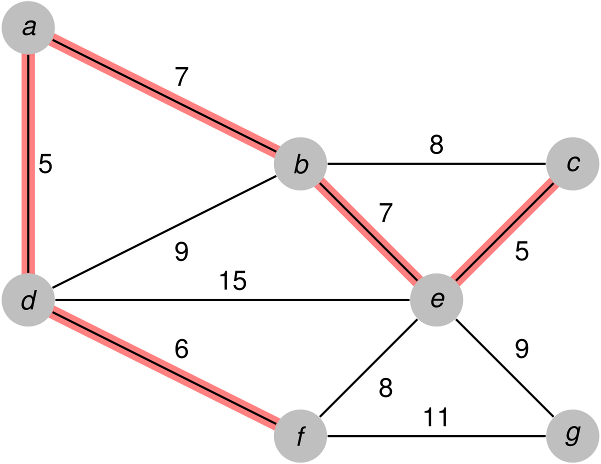
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal’s Algorithm



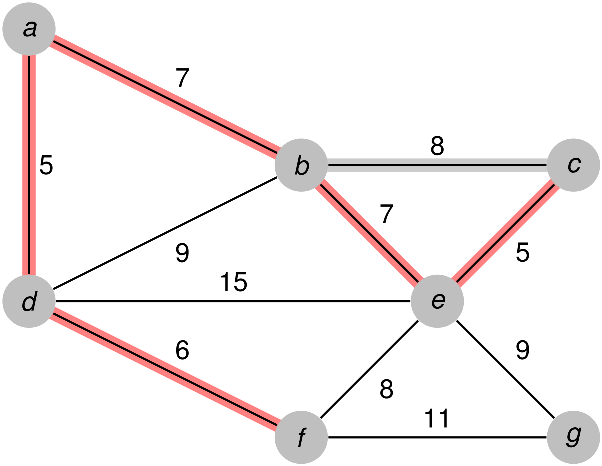
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm



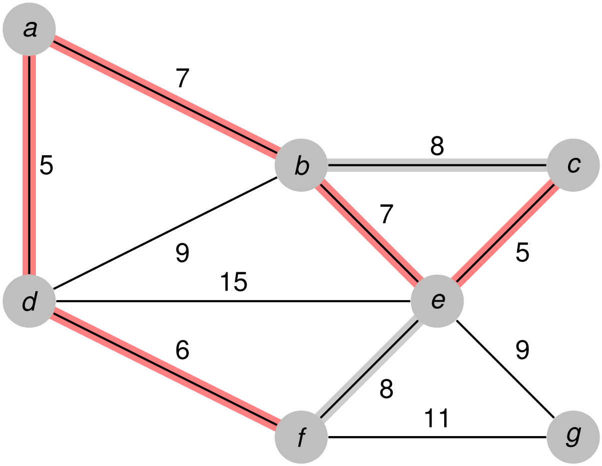
Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm

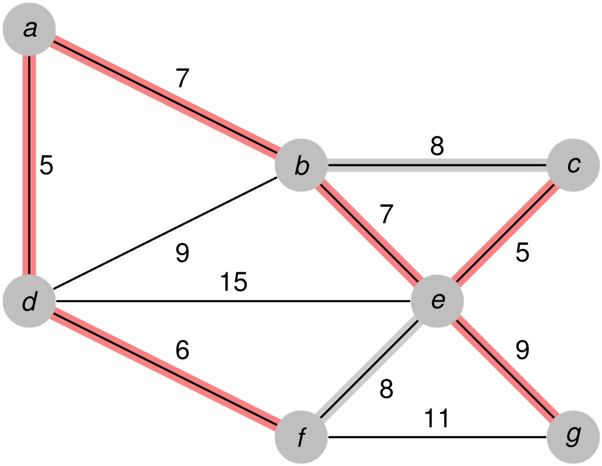


Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Kruskal's Algorithm

```
typedef pair< double, pair<int, int> > WEdge;
typedef vector< vector< pair<double, int> > > WGraph;

int Kruskal(const WGraph& G, vector<pair<int,int>>& MST) {
  int n = G.size();
  UnionFind uf(n);
  int totalWeight = 0;
  priority_queue<WEdge,vector<WEdge>,greater<WEdge>> Q;
  for (int v = 0; v < n; ++v)
    for (auto& x: G[v]) if (v < x.second)
       Q.push({x.first,{v,x.second}});
  int sz = 0;
  while (sz < n - 1) {
    int u = Q.top().second.first;
    int v = Q.top().second.second;
    double wEdge = Q.top().first; Q.pop();
    if (uf.find(u) != uf.find(v)) {
      totalWeight += wEdge; ++sz;
      MST.push_back({u,v}); uf.merge(u,v);
    } }
  return totalWeight; }
```

Kruskal's Algorithm

```
 class UnionFind {
   vector<int> rep;
   vector<int> size; // only valid for representative elements
   vector<vector<int>> elementsInClass; // only valid for reps
public:
   UnionFind(int n) : rep(n), size(n), elementsInClass(n) {
     for (int i = 0; i < n; ++i)
       {rep[i] = i; size[i] = 1; elementsInClass[i] = {i}; } }
   int find (int e) {return rep[e];}
   void merge(int e1, int e2) {
     int r1 = find(e1); int r2 = find(e2);
     if (size[r1] > size[r2]) { // r1 new rep
       size[r1] += size[r2];
       for (auto& x: elementsInClass[r2])
         {rep[x] = r1; elementsInClass[r1].push_back(x);} }
     else { // r2 new rep
       size[r2] += size[r1];
       for (auto& x: elementsInClass[r1])
         {rep[x] = r2; elementsInClass[r2].push_back(x); } } }
 };
```

Kruskal's Algorithm

Theorem

Kruskal's Algorithm runs in $O(m \log n)$ time

Kruskal's Algorithm

Theorem

Kruskal's Algorithm runs in $O(m \log n)$ time

Proof:

Let us consider the contribution to the cost of:

```
priority_queue<WEdge,vector<WEdge>,greater<WEdge>> Q;
for (int v = 0; v < n; ++v)
  for (auto& x: G[v]) if (v < x.second)
     Q.push({x.first,{v,x.second}});
```

Adding m edges to Q takes $O(m \log m)$ time.

Kruskal's Algorithm

Proof (continued):

Let us consider the contribution to the cost of the loop:

```
int sz = 0;
while (sz < n - 1) {
  int u = Q.top().second.first;
  int v = Q.top().second.second;
  double wEdge = Q.top().first; Q.pop();
  if (uf.find(u) != uf.find(v)) {
    totalWeight += wEdge; ++sz;
    MST.push_back({u,v}); uf.merge(u,v);
  } }
```

Kruskal's Algorithm

Proof (continued):

Let us consider the contribution to the cost of the loop:

```
int sz = 0;
while (sz < n - 1) {
  int u = Q.top().second.first;
  int v = Q.top().second.second;
  double wEdge = Q.top().first; Q.pop();
  if (uf.find(u) != uf.find(v)) {
    totalWeight += wEdge; ++sz;
    MST.push_back({u,v}); uf.merge(u,v);
  } }
```

- Code in red: The loop is executed at most m times.
  At each iteration we pop one element from Q,
  which takes $O(\log m)$ time, and request 2 Finds from the Union-Find.

Proof (continued):

Let us consider the contribution to the cost of the loop:

```
int sz = 0;
while (sz < n - 1) {
  int u = Q.top().second.first;
  int v = Q.top().second.second;
  double wEdge = Q.top().first; Q.pop();
  if (uf.find(u) != uf.find(v)) {
    totalWeight += wEdge; ++sz;
    MST.push_back({u,v}); uf.merge(u,v);
  } }
```

- Code in red: The loop is executed at most m times.
  At each iteration we pop one element from Q,
  which takes $O(\log m)$ time, and request 2 Finds from the Union-Find.
- Code in blue: Overall the algorithm makes $n-1$ Merges.

Kruskal's Algorithm

Proof (continued):

Let us consider the contribution to the cost of the loop:

```
int sz = 0;
while (sz < n - 1) {
  int u = Q.top().second.first;
  int v = Q.top().second.second;
  double wEdge = Q.top().first; Q.pop();
  if (uf.find(u) != uf.find(v)) {
    totalWeight += wEdge; ++sz;
    MST.push_back({u,v}); uf.merge(u,v);
  } }
```

- Code in red: The loop is executed at most m times.
  At each iteration we pop one element from Q,
  which takes $O(\log m)$ time, and request 2 Finds from the Union-Find.
- Code in blue: Overall the algorithm makes $n - 1$ Merges.

All in all, the cost is $O(m \log m)$ plus the cost of $O(m)$ Finds and $O(n)$ Merges.

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time
- **Merge:** In a sequence of $O(n)$ Merges, every element can change representative at most $O(\log n)$ times because every time it changes, the size of its class at least doubles. Since each change needs $O(n)$ time, the Union-Find needs $O(n \log n)$ time to process the Merges.

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time
- **Merge:** In a sequence of $O(n)$ Merges, every element can change representative at most $O(\log n)$ times because every time it changes, the size of its class at least doubles. Since each change needs $O(n)$ time, the Union-Find needs $O(n \log n)$ time to process the Merges.

The total time is hence $O(m \log m) + O(m) + O(n \log n)$.

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time
- **Merge:** In a sequence of $O(n)$ Merges, every element can change representative at most $O(\log n)$ times because every time it changes, the size of its class at least doubles. Since each change needs $O(n)$ time, the Union-Find needs $O(n \log n)$ time to process the Merges.

The total time is hence $O(m \log m) + O(m) + O(n \log n)$.

The graph is connected (otherwise there is no spanning tree). So $m \geq n - 1$.

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time
- **Merge:** In a sequence of $O(n)$ Merges, every element can change representative at most $O(\log n)$ times because every time it changes, the size of its class at least doubles. Since each change needs $O(n)$ time, the Union-Find needs $O(n \log n)$ time to process the Merges.

The total time is hence $O(m \log m) + O(m) + O(n \log n)$.

The graph is connected (otherwise there is no spanning tree). So $m \geq n - 1$.

Also since m is $O(n^2)$ we know that $O(\log m) = O(\log n)$.

Kruskal's Algorithm

Proof (continued):

In the previous Union-Find implementation:

- **Find:** costs constant time
- **Merge:** In a sequence of $O(n)$ Merges, every element can change representative at most $O(\log n)$ times because every time it changes, the size of its class at least doubles. Since each change needs $O(n)$ time, the Union-Find needs $O(n \log n)$ time to process the Merges.

The total time is hence $O(m \log m) + O(m) + O(n \log n)$.

The graph is connected (otherwise there is no spanning tree). So $m \geq n - 1$.

Also since m is $O(n^2)$ we know that $O(\log m) = O(\log n)$.

Hence, Kruskal's Algorithm runs in $O(m \log n)$. □