Greedy Algorithms: Exchange Property

Algorithm Design & Analysis

Spring 2019
Outline

Greedy Algorithms: Exchange Property
- Job Scheduling: Minimizing Lateness
- Minimum-Cost Spanning Trees
  - Maximum Greed: Kruskal’s Algorithm
  - Analysis: Kruskal’s Algorithm
  - Moderate Greed: Prim’s Algorithm
  - Analysis: Prim’s Algorithm
  - Reverse Greed: Reverse-Delete Algorithm
  - Allowing Non-Unique Edge Costs
The Exchange Property

**Exchange Property**: Show that an optimal solution can be sequentially transformed into a greedy solution without compromising optimality.
Outline

**Greedy Algorithms: Exchange Property**

Job Scheduling: Minimizing Lateness

Minimum-Cost Spanning Trees
  - Maximum Greed: Kruskal’s Algorithm
  - Analysis: Kruskal’s Algorithm
  - Moderate Greed: Prim’s Algorithm
  - Analysis: Prim’s Algorithm
  - Reverse Greed: Reverse-Delete Algorithm
  - Allowing Non-Unique Edge Costs
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.
  • Only one process can be executed at a time
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.

- Only one process can be executed at a time
- A process must run to completion before another can be executed
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.

- Only one process can be executed at a time
- A process must run to completion before another can be executed
- Each process has a duration \( t_i \) and a deadline \( d_i \)
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.

- Only one process can be executed at a time
- A process must run to completion before another can be executed
- Each process has a duration $t_i$ and a deadline $d_i$
- Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start and finish times) to minimize maximum lateness
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.

- Only one process can be executed at a time
- A process must run to completion before another can be executed
- Each process has a duration $t_i$ and a deadline $d_i$
- Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start and finish times) to minimize maximum lateness
- Lateness of process $i$: $L_i = \max\{0, f_i - d_i\}$
Minimizing Lateness

The Problem: A list of processes needs to be scheduled.

- Only one process can be executed at a time
- A process must run to completion before another can be executed
- Each process has a duration $t_i$ and a deadline $d_i$
- Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start and finish times) to minimize maximum lateness
- Lateness of process $i$: $L_i = \max\{0, f_i - d_i\}$
- Resource is first available at time 0
How to Minimize Lateness

Possible strategies

---

Let's try it... 

Input: 

- $(5, 8)$, $(4, 12)$, $(6, 7)$

**Shortest First:**

- $(4, 12)$ → $(0, 4)$, $(5, 8)$ → $(4, 9)$, $(6, 7)$ → $(9, 15)$

Gives latenesses: 0, 1, 8: Max. lateness = 8.

**Early deadlines first:**

- $(6, 7)$ → $(0, 6)$, $(5, 8)$ → $(6, 11)$, $(4, 12)$ → $(11, 15)$

Gives latenesses: 0, 3, 3 Max. lateness = 3.

**Shortest slack first:**

- $(6, 7)$ → $(0, 6)$, $(5, 8)$ → $(6, 11)$, $(4, 12)$ → $(11, 15)$

Gives latenesses: 0, 3, 3 Max. lateness = 3.

Uh-oh: A tie!
How to Minimize Lateness

Possible strategies

• Shortest jobs first
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first

Let's try it....

Input:

- (5, 8),
- (4, 12),
- (6, 7)

Shortest First:

- (4, 12) → (0, 4),
- (5, 8) → (4, 9),
- (6, 7) → (9, 15)

Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first:

- (6, 7) → (0, 6),
- (5, 8) → (6, 11),
- (4, 12) → (11, 15)

Gives latenesses: 0, 3, 3: Max. lateness = 3.

Shortest slack first:

- (6, 7) → (0, 6),
- (5, 8) → (6, 11),
- (4, 12) → (11, 15)

Gives latenesses: 0, 3, 3: Max. lateness = 3.

Uh-oh: A tie!
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)

Let's try it...

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) → (0, 4), (5, 8) → (4, 9), (6, 7) → (9, 15)
Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first: (6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)
Gives latenesses: 0, 3, 3: Max. lateness = 3.

Shortest slack first: (6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)
Gives latenesses: 0, 3, 3: Max. lateness = 3.

Uh-oh: A tie!
How to Minimize Lateness

Possible strategies

• Shortest jobs first (get more done faster!)
• Earlier deadlines first (triage!)
• Do jobs with shortest slack time first

Let's try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) → (0, 4), (5, 8) → (4, 9), (6, 7) → (9, 15)
Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first: (6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)
Gives latenesses: 0, 3, 3 Max. lateness = 3.

Shortest slack first: (6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)
Gives latenesses: 0, 3, 3 Max. lateness = 3.

Uh-oh: A tie!
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First:
Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: $(5, 8), (4, 12), (6, 7)$

Shortest First: $(4, 12) \rightarrow (0, 4), (5, 8) \rightarrow (4, 9), (6, 7) \rightarrow (9, 15)$
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....
Input: $(5, 8), (4, 12), (6, 7)$
Shortest First: $(4, 12) \rightarrow (0, 4), (5, 8) \rightarrow (4, 9), (6, 7) \rightarrow (9, 15)$
Gives latenesses: $0, 1, 8$: Max. lateness $= 8$. 
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) → (0, 4), (5, 8) → (4, 9), (6, 7) → (9, 15)

Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first:
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: $(5, 8), (4, 12), (6, 7)$

Shortest First: $(4, 12) \rightarrow (0, 4), (5, 8) \rightarrow (4, 9), (6, 7) \rightarrow (9, 15)$

Gives latenesses: $0, 1, 8$: Max. lateness $= 8$.

Early deadlines first:

$(6, 7) \rightarrow (0, 6), (5, 8) \rightarrow (6, 11), (4, 12) \rightarrow (11, 15)$
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: $(5, 8), (4, 12), (6, 7)$

Shortest First: $(4, 12) \rightarrow (0, 4), (5, 8) \rightarrow (4, 9), (6, 7) \rightarrow (9, 15)$
Gives latenesses: $0, 1, 8$: Max. lateness $= 8$.

Early deadlines first:
$(6, 7) \rightarrow (0, 6), (5, 8) \rightarrow (6, 11), (4, 12) \rightarrow (11, 15)$
Gives latenesses: $0, 3, 3$ Max. lateness $= 3$. 
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) → (0, 4), (5, 8) → (4, 9), (6, 7) → (9, 15)

Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first:
(6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)

Gives latenesses: 0, 3, 3 Max. lateness = 3.

Shortest slack first:
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) → (0, 4), (5, 8) → (4, 9), (6, 7) → (9, 15)

Gives latenesses: 0, 1, 8: Max. lateness = 8.

Early deadlines first:
(6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)

Gives latenesses: 0, 3, 3 Max. lateness = 3.

Shortest slack first:
(6, 7) → (0, 6), (5, 8) → (6, 11), (4, 12) → (11, 15)
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: (5, 8), (4, 12), (6, 7)

Shortest First: (4, 12) $\rightarrow$ (0, 4), (5, 8) $\rightarrow$ (4, 9), (6, 7) $\rightarrow$ (9, 15)
Gives latenesses: 0, 1, 8: Max. lateness $= 8$.

Early deadlines first:
(6, 7) $\rightarrow$ (0, 6), (5, 8) $\rightarrow$ (6, 11), (4, 12) $\rightarrow$ (11, 15)
Gives latenesses: 0, 3, 3 Max. lateness $= 3$.

Shortest slack first:
(6, 7) $\rightarrow$ (0, 6), (5, 8) $\rightarrow$ (6, 11), (4, 12) $\rightarrow$ (11, 15)
Gives latenesses: 0, 3, 3 Max. lateness $= 3$.  

Uh-oh: A tie!
How to Minimize Lateness

Possible strategies

- Shortest jobs first (get more done faster!)
- Earlier deadlines first (triage!)
- Do jobs with shortest slack time first (slack of job $i$ is $d_i - t_i$)

Let’s try it....

Input: $(5, 8), (4, 12), (6, 7)$

Shortest First: $(4, 12) \rightarrow (0, 4), (5, 8) \rightarrow (4, 9), (6, 7) \rightarrow (9, 15)$
Gives latenesses: $0, 1, 8$: Max. lateness $= 8$.

Early deadlines first:
$(6, 7) \rightarrow (0, 6), (5, 8) \rightarrow (6, 11), (4, 12) \rightarrow (11, 15)$
Gives latenesses: $0, 3, 3$ Max. lateness $= 3$.

Shortest slack first:
$(6, 7) \rightarrow (0, 6), (5, 8) \rightarrow (6, 11), (4, 12) \rightarrow (11, 15)$
Gives latenesses: $0, 3, 3$ Max. lateness $= 3$.

Uh-oh: A tie!
Earliest Deadline First

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

**Algorithm 1** Scheduling to Minimize Lateness
Earliest Deadline First

Idea: Inspect processes in order of earliest deadline; assume we’ve sorted them by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

**Algorithm 2** Scheduling to Minimize Lateness


**Earliest Deadline First**

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

---

**Algorithm 3** Scheduling to Minimize Lateness

```
procedure MIN_LATENESS(\( J_1, \ldots, J_n \))
\>
\( J_i = (t_i, d_i) \)
```
Earliest Deadline First

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

**Algorithm 4** Scheduling to Minimize Lateness

```
procedure MINLATENESS(J_1, \ldots, J_n) \quad \triangleright J_i = (t_i, d_i)

Ensure: Schedule \( S_1, \ldots, S_n \) where \( S_i = (s_i, f_i) \) and where maximum lateness is minimized
```
Earliest Deadline First

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

**Algorithm 5** Scheduling to Minimize Lateness

```plaintext
procedure MINLATENESS($J_1, \ldots, J_n$)  \Comment{$J_i = (t_i, d_i)$}
Ensure: Schedule $S_1, \ldots, S_n$ where $S_i = (s_i, f_i)$ and where maximum lateness is minimized
Sort jobs so that $d_1, \leq \ldots \leq d_n$
```
Earliest Deadline First

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

**Algorithm 6** Scheduling to Minimize Lateness

```plaintext
procedure MINLATENESS($J_1, \ldots, J_n$) \hfill $J_i = (t_i, d_i)$

Ensure: Schedule $S_1, \ldots, S_n$ where $S_i = (s_i, f_i)$ and where maximum lateness is minimized

Sort jobs so that $d_1, \leq \ldots \leq d_n$

Set `nextStart` ← 0
```
Greedy Algorithms: Exchange Property

Earliest Deadline First

Idea: Inspect processes in order of earliest deadline; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

Algorithm 7 Scheduling to Minimize Lateness

```plaintext
procedure MINLATENESS(J_1, \ldots, J_n)  \triangleright J_i = (t_i, d_i)
Ensure: Schedule \( S_1, \ldots, S_n \) where \( S_i = (s_i, f_i) \) and where maximum lateness is minimized
Sort jobs so that \( d_1, \leq \ldots \leq d_n \)
Set nextStart ← 0
for \( i \leftarrow 1 \ldots n \) do
```

```plaintext
\ldots
```

```plaintext
end procedure
```
**Earliest Deadline First**

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

**Algorithm 8** Scheduling to Minimize Lateness

```algorithm
procedure MINLATENESS(\( J_1, \ldots, J_n \)) \> \( J_i = (t_i, d_i) \)
Ensure: Schedule \( S_1, \ldots, S_n \) where \( S_i = (s_i, f_i) \) and where maximum lateness is minimized

Sort jobs so that \( d_1, \leq \ldots \leq d_n \)
Set \( \text{nextStart} \leftarrow 0 \)
for \( i \leftarrow 1 \ldots n \) do
    \( s_i \leftarrow \text{nextStart} \) and \( f_i \leftarrow \text{nextStart} + t_i \)
```

Earliest Deadline First

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

---

**Algorithm 9** Scheduling to Minimize Lateness

```plaintext
procedure MINLATENESS($J_1, \ldots, J_n$)  \> $J_i = (t_i, d_i)$

Ensure: Schedule $S_1, \ldots, S_n$ where $S_i = (s_i, f_i)$ and where maximum lateness is minimized

Sort jobs so that $d_1, \leq \ldots \leq d_n$

Set $nextStart \leftarrow 0$

for $i \leftarrow 1 \ldots n$ do

    $s_i \leftarrow nextStart$ and $f_i \leftarrow nextStart + t_i$

    $nextStart \leftarrow f_i$
```

Earliest Deadline First

Idea: Inspect processes in order of earliest deadline; assume we’ve sorted them by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \).

Algorithm 10 Scheduling to Minimize Lateness

procedure \textsc{MinLate}\textsc{ness}(J_1, \ldots, J_n) \quad \triangleright J_i = (t_i, d_i)

Ensure: Schedule \( S_1, \ldots, S_n \) where \( S_i = (s_i, f_i) \) and where maximum lateness is minimized

Sort jobs so that \( d_1, \leq \ldots \leq d_n \)

Set \textit{nextStart} \leftarrow 0

for \( i \leftarrow 1 \ldots n \) do

\( s_i \leftarrow \text{nextStart} \) and \( f_i \leftarrow \text{nextStart} + t_i \)

\textit{nextStart} \leftarrow f_i
**Earliest Deadline First**

Idea: Inspect processes in order of *earliest deadline*; assume we’ve sorted them by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

---

**Algorithm 11** Scheduling to Minimize Lateness

```
procedure MINLATENESS($J_1, \ldots, J_n$) \Comment{$J_i = (t_i, d_i)$}  
Ensure: Schedule $S_1, \ldots, S_n$ where $S_i = (s_i, f_i)$ and where maximum lateness is minimized

Sort jobs so that $d_1, \leq \ldots \leq d_n$
Set $nextStart \leftarrow 0$
for $i \leftarrow 1 \ldots n$ do
    $s_i \leftarrow nextStart$ and $f_i \leftarrow nextStart + t_i$
end procedure
```
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource idle time.
- MinLateness produces a schedule with no inversions.
- An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\).
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we'll prove this).
- A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness (we'll prove this).
- Any two schedules with no idle time and no inversions have the same maximum lateness (we'll prove this).
- Thus, any schedule with no idle time and no inversions minimizes maximum lateness.
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource *idle time*
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource idle time
- MinLateness produces a schedule with no inversions
Minimizing Maximum Lateness: Correctness Proof

Key Observations

• MinLateness produces a schedule with no resource *idle time*
• MinLateness produces a schedule with no *inversions*
  • An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource *idle time*
- MinLateness produces a schedule with no *inversions*
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Any two schedules with no idle time and no inversions have the same maximum lateness (we'll prove this)

Thus, any schedule with no idle time and no inversions minimizes maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource *idle time*
- MinLateness produces a schedule with no *inversions*
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)

Any two schedules with no idle time and no inversions have the same maximum lateness (we’ll prove this)

Thus, any schedule with no idle time and no inversions minimizes maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource idle time
- MinLateness produces a schedule with no inversions
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)
- A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Key Observations

• MinLateness produces a schedule with no resource idle time
• MinLateness produces a schedule with no inversions
  • An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
• A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)
• A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness (we’ll prove this)
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource idle time
- MinLateness produces a schedule with no inversions
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)
- A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness (we’ll prove this)
- Any two schedules with no idle time and no inversions have the same maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource *idle time*
- MinLateness produces a schedule with no *inversions*
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)
- A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness (we’ll prove this)
- Any two schedules with no idle time and no inversions have the same maximum lateness (we’ll prove this)
Minimizing Maximum Lateness: Correctness Proof

Key Observations

- MinLateness produces a schedule with no resource idle time
- MinLateness produces a schedule with no inversions
  - An inversion in a schedule consists of pairs \((s_i, f_i), (s_j, f_j)\) with \(i < j\) and \(d_i > d_j\)
- A schedule with idle time can be converted to one with no idle time without increasing maximum lateness (we’ll prove this)
- A schedule with \(k > 0\) inversions can be transformed into one with \(k - 1\) inversions without increasing the maximum lateness (we’ll prove this)
- Any two schedules with no idle time and no inversions have the same maximum lateness (we’ll prove this)
- Thus, any schedule with no idle time and no inversions minimizes maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with idle time can be converted to one with no idle time without increasing maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Proof:
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Proof:

- Suppose for some $i$, $g = f_i - s_{i+1} > 0$. 
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Proof:

• Suppose for some $i$, $g = f_i - s_{i+1} > 0$. ($g$ units of idle time)
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Proof:

- Suppose for some $i$, $g = f_i - s_{i+1} > 0$. ($g$ units of idle time)
- For each $j > i$, replace $(s_j, f_j)$ with $(s_j - g, f_j - g)$. 
Minimizing Maximum Lateness: Correctness Proof

**Observation**

*A schedule with idle time can be converted to one with no idle time without increasing maximum lateness*

**Proof:**

- Suppose for some \( i \), \( g = f_i - s_{i+1} > 0 \). (\( g \) units of idle time)
- For each \( j > i \), replace \((s_j, f_j)\) with \((s_j - g, f_j - g)\).
- The schedule now has one fewer gap since now \( s_{i+1} = f_i \).
Minimizing Maximum Lateness: Correctness Proof

Observation

A schedule with idle time can be converted to one with no idle time without increasing maximum lateness

Proof:

• Suppose for some \( i \), \( g = f_i - s_{i+1} > 0 \). \( (g \) units of idle time)
• For each \( j > i \), replace \( (s_j, f_j) \) with \( (s_j - g, f_j - g) \).
• The schedule now has one fewer gap since now \( s_{i+1} = f_i \).
• But maximum lateness has not increased since finish times have only decreased
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with $k > 0$ inversions can be transformed into one with $k - 1$ inversions without increasing the maximum lateness.
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

Proof:
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with $k > 0$ inversions can be transformed into one with $k - 1$ inversions without increasing the maximum lateness

Proof:

- For some $i$, there's an inversion between jobs $i$ and $i + 1$
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

Proof:

- For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
- Swapping those jobs doesn’t increase the maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

Proof:

- For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
- Swapping those jobs doesn’t increase the maximum lateness
  - Lateness doesn’t change for jobs \( 1, \ldots, i - 1 \) or \( i + 2, \ldots n \)
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

Proof:

• For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
• Swapping those jobs doesn’t increase the maximum lateness
  • Lateness doesn’t change for jobs \( 1, \ldots, i - 1 \) or \( i + 2, \ldots n \)
  • \( J_{i+1} \) finishes earlier (it’s now in slot \( i \)) so it’s lateness decreases
Minimizing Maximum Lateness: Correctness Proof

Observation
A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

Proof:

- For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
- Swapping those jobs doesn’t increase the maximum lateness
  - Lateness doesn’t change for jobs 1, \ldots, \( i - 1 \) or \( i + 2, \ldots n \)
  - \( J_{i+1} \) finishes earlier (it’s now in slot \( i \)) so it’s lateness decreases
  - \( L_i \) goes from \( f_i - d_i \) (or 0) to \( f_{i+1} - d_i \) (or 0)
Minimizing Maximum Lateness: Correctness Proof

**Observation**

*A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness*

**Proof:**

- For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
- Swapping those jobs doesn’t increase the maximum lateness
  - Lateness doesn’t change for jobs \( 1, \ldots, i - 1 \) or \( i + 2, \ldots n \)
  - \( J_{i+1} \) finishes earlier (it’s now in slot \( i \)) so it’s lateness decreases
  - \( L_i \) goes from \( f_i - d_i \) (or 0) to \( f_{i+1} - d_i \) (or 0)
  - But \( f_{i+1} - d_i < f_{i+1} - d_{i+1} = L_{i+1} \) (original lateness of \( J_{i+1} \)) since \( d_i > d_{i+1} \)
Minimizing Maximum Lateness: Correctness Proof

**Observation**

A schedule with \( k > 0 \) inversions can be transformed into one with \( k - 1 \) inversions without increasing the maximum lateness

**Proof:**

- For some \( i \), there’s an inversion between jobs \( i \) and \( i + 1 \)
- Swapping those jobs doesn’t increase the maximum lateness
  - Lateness doesn’t change for jobs \( 1, \ldots, i - 1 \) or \( i + 2, \ldots n \)
  - \( J_{i+1} \) finishes earlier (it’s now in slot \( i \)) so its lateness decreases
  - \( L_i \) goes from \( f_i - d_i \) (or 0) to \( f_{i+1} - d_i \) (or 0)
  - But \( f_{i+1} - d_i < f_{i+1} - d_{i+1} = L_{i+1} \) (original lateness of \( J_{i+1} \)) since \( d_i > d_{i+1} \)
- Thus the maximum lateness of the set of jobs does not increase
Minimizing Maximum Lateness: Correctness Proof

Observation
Minimizing Maximum Lateness: Correctness Proof

Observation
Any two schedules with no idle time and no inversions have the same maximum lateness
Minimizing Maximum Lateness: Correctness Proof

Observation

Any two schedules with no idle time and no inversions have the same maximum lateness

- Two schedules with no inversions differ only in order of jobs with same deadlines
Minimizing Maximum Lateness: Correctness Proof

Observation

Any two schedules with no idle time and no inversions have the same maximum lateness

- Two schedules with no inversions differ only in order of jobs with same deadlines
- Consider the jobs $J_i, \ldots, J_k$ that having deadline $d$. 
Minimizing Maximum Lateness: Correctness Proof

Observation

Any two schedules with no idle time and no inversions have the same maximum lateness

• Two schedules with no inversions differ only in order of jobs with same deadlines
• Consider the jobs $J_i, \ldots, J_k$ that having deadline $d$
• They appear consecutively in the schedule with latenesses $f_i - d, \ldots, f_k - d$
Minimizing Maximum Lateness: Correctness Proof

Observation
Any two schedules with no idle time and no inversions have the same maximum lateness

- Two schedules with no inversions differ only in order of jobs with same deadlines
- Consider the jobs $J_i, \ldots, J_k$ that having deadline $d$.
- They appear consecutively in the schedule with latenesses $f_i - d, \ldots, f_k - d$
- Let $f = f_k$, so maximum lateness of these jobs is $f - d$. 
Minimizing Maximum Lateness: Correctness Proof

Observation

Any two schedules with no idle time and no inversions have the same maximum lateness

- Two schedules with no inversions differ only in order of jobs with same deadlines
- Consider the jobs $J_i, \ldots, J_k$ that having deadline $d$.
- They appear consecutively in the schedule with latenesses $f_i - d, \ldots, f_k - d$
- Let $f = f_k$, so maximum lateness of these jobs is $f - d$.
- But this holds regardless of the relative ordering of $J_i, \ldots, J_k$ in the schedule
Outline

Greedy Algorithms: Exchange Property

Job Scheduling: Minimizing Lateness

Minimum-Cost Spanning Trees
  Maximum Greed: Kruskal’s Algorithm
  Analysis: Kruskal’s Algorithm
  Moderate Greed: Prim’s Algorithm
  Analysis: Prim’s Algorithm
  Reverse Greed: Reverse-Delete Algorithm
  Allowing Non-Unique Edge Costs
Minimum-Cost Spanning Trees

Figure: A Graph $G$ with Positive Edge-Weights
Minimum-Cost Spanning Trees

Figure: A Min-Cost Spanning Tree for $G$
Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications
Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications

• Classic Application: Underground Cable (Power, Telecom, ...)

Greedy Algorithms: Exchange Property
Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications

- Classic Application: Underground Cable (Power, Telecom, ...)
- Efficient broadcasting on a computer network (Note: different from shortest paths)
Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications

- Classic Application: Underground Cable (Power, Telecom, ...)
- Efficient broadcasting on a computer network (Note: different from shortest paths)
- Taxonomy (mental maps)
Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications

- Classic Application: Underground Cable (Power, Telecom, ...)
- Efficient broadcasting on a computer network (Note: different from shortest paths)
- Taxonomy (mental maps)
- Reliable subnetwork


Minimum-Cost Spanning Trees

Computing a minimum-cost spanning tree for a graph has many applications:

- Classic Application: Underground Cable (Power, Telecom, ...)
- Efficient broadcasting on a computer network (Note: different from shortest paths)
- Taxonomy (mental maps)
- Reliable subnetwork
- Approximate solutions to harder problems, such as TSP
The Problem
The Problem

Definition
The cost of a subgraph $G' = (V', E')$ of a graph $G = (V, E)$ with edge-costs is the sum of the costs of the edges of $E'$. 
The Problem

Definition
The cost of a subgraph \( G' = (V', E') \) of a graph \( G = (V, E) \) with edge-costs is the sum of the costs of the edges of \( E' \).

Observation
A minimum-cost connected spanning subgraph for a connected graph \( G = (V, E) \) with positive edge costs \( c() \) is a tree.
The Problem

**Definition**
The cost of a subgraph $G' = (V', E')$ of a graph $G = (V, E)$ with edge-costs is the sum of the costs of the edges of $E'$.

**Observation**
A minimum-cost connected spanning subgraph for a connected graph $G = (V, E)$ with positive edge costs $c()$ is a tree.

**Definition**
A spanning tree $T$ of a graph $G = (V, E)$ with edge costs is minimum-cost if no other spanning tree has lower cost.
The Problem

Definition
The cost of a subgraph \( G' = (V', E') \) of a graph \( G = (V, E) \) with edge-costs is the sum of the costs of the edges of \( E' \).

Observation
A minimum-cost connected spanning subgraph for a connected graph \( G = (V, E) \) with positive edge costs \( c() \) is a tree.

Definition
A spanning tree \( T \) of a graph \( G = (V, E) \) with edge costs is minimum-cost if no other spanning tree has lower cost.

We will assume that all edge-costs are distinct; we’ll relax this assumption at the end of class.
Cuts and Cut-Edges
A cut in a graph $G = (V, E)$ is a partition of $V$ into two sets $\{S, V - S\}$. The edges $E(S, V - S)$ with one endpoint in each set are called cut edges.
Cuts and Cut-Edges

Definition

A cut in a graph \( G = (V, E) \) is a partition of \( V \) into two sets \( \{S, V - S\} \). The edges \( E(S, V - S) \) with one endpoint in each set are called cut edges.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$. 
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components.
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge.
- $S$ and $V - S$ are the vertex sets of the (two) components of $T - \{e\}$.
- Adding an edge $e$ of $G - T$ to $T$ creates a unique cycle in $T + \{e\}$; the cycle contains $e$.
- For any cycle $C$ and cut $\{S, V - S\}$, $|E(C) \cap E(S, V - S)|$ is even.
  - That is, any cycle and any cut share an even number of edges.
  - So if a cycle intersects a cut, they share at least two edges.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$.

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components.
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge
  - $S$ and $V - S$ are the vertex sets of the (two) components of $T - \{e\}$
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$.

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components.
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge.
  - $S$ and $V - S$ are the vertex sets of the (two) components of $T - \{e\}$.
- Adding an edge $e$ of $G - T$ to $T$ creates a unique cycle in $T + \{e\}$; the cycle contains $e$. 


Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$.

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components.
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge.
  - $S$ and $V - S$ are the vertex sets of the (two) components of $T - \{e\}$.
- Adding an edge $e$ of $G - T$ to $T$ creates a unique cycle in $T + \{e\}$; the cycle contains $e$.
- For any cycle $C$ and cut $\{S, V - S\}$, $|E(C) \cap E(S, V - S)|$ is even.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let $G = (V, E)$ be a graph and $T$ be a spanning tree of $G$.

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components.
- Every edge $e$ of $T$ defines a cut in $G$ that has $e$ as a cut edge.
  - $S$ and $V - S$ are the vertex sets of the (two) components of $T - \{e\}$.
- Adding an edge $e$ of $G - T$ to $T$ creates a unique cycle in $T + \{e\}$; the cycle contains $e$.
- For any cycle $C$ and cut $\{S, V - S\}$, $|E(C) \cap E(S, V - S)|$ is even.
  - That is, any cycle and any cut share an even number of edges.
Trees, Cycles, & Cuts

Trees, cycles, and cuts relate to one another in useful ways.

Let \( G = (V, E) \) be a graph and \( T \) be a spanning tree of \( G \)

Observations:

- Deleting a cycle edge from a graph does not change the number of connected components
- Every edge \( e \) of \( T \) defines a cut in \( G \) that has \( e \) as a cut edge
  - \( S \) and \( V - S \) are the vertex sets of the (two) components of \( T - \{e\} \)
- Adding an edge \( e \) of \( G - T \) to \( T \) creates a unique cycle in \( T + \{e\} \); the cycle contains \( e \)
- For any cycle \( C \) and cut \( \{S, V - S\} \), \( |E(C) \cap E(S, V - S)| \) is even.
  - That is, any cycle and any cut share an even number of edges
  - So if a cycle intersects a cut, they share at least two edges
Properties of Min-Cost Spanning Trees

Observations:

• If $T$ is a MCST of $G$ and $e \in E(G) - E(T)$, then $e$ is the highest cost edge on the unique cycle in $T + \{e\}$.

• For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

• For any cycle in $G$, its highest-cost edge is in no MCST of $G$. 
Properties of Min-Cost Spanning Trees

Observations:

• If $T$ is a MCST of $G$ and $e \in E(G) - E(T)$, then $e$ is the highest cost edge on the unique cycle in $T + \{e\}$. 
Properties of Min-Cost Spanning Trees

Observations:

- If \( T \) is a MCST of \( G \) and \( e \in E(G) - E(T) \), then \( e \) is the highest cost edge on the unique cycle in \( T + \{e\} \).
- For any cut in \( G \), its lowest-cost edge is in every MCST of \( G \).
Properties of Min-Cost Spanning Trees

Observations:

• If $T$ is a MCST of $G$ and $e \in E(G) - E(T)$, then $e$ is the highest cost edge on the unique cycle in $T + \{e\}$.

• For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

• For any cycle in $G$, its highest-cost edge is in no MCST of $G$. 
Proof of Cut Property

For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

Proof.
Proof of Cut Property

For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

Proof.

• Let $T$ be any MCST of $G$, let $\{S, V - S\}$ be any cut of $G$, and let $e$ be the cheapest edge of the cut.
Proof of Cut Property

For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

Proof.

- Let $T$ be any MCST of $G$, let $\{S, V - S\}$ be any cut of $G$, and let $e$ be the cheapest edge of the cut.
- If $e \not\in T$, then $T + \{e\}$ contains a unique cycle $C$, and $e \in C$.
Proof of Cut Property

For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

Proof.

- Let $T$ be any MCST of $G$, let $\{S, V - S\}$ be any cut of $G$, and let $e$ be the cheapest edge of the cut.
- If $e \not\in T$, then $T + \{e\}$ contains a unique cycle $C$, and $e \in C$
- So $C$ contains another edge $e' \in E(S, V - S)$. 

\[\square\]
**Proof of Cut Property**

For any cut in $G$, its lowest-cost edge is in every MCST of $G$.

*Proof.*

- Let $T$ be any MCST of $G$, let $\{S, V - S\}$ be any cut of $G$, and let $e$ be the cheapest edge of the cut.
- If $e \notin T$, then $T + \{e\}$ contains a unique cycle $C$, and $e \in C$.
- So $C$ contains another edge $e' \in E(S, V - S)$.
- But $T + \{e\} - \{e'\}$ is a tree with lower cost than $T \Rightarrow \Leftarrow$. 

\[\square\]
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$. 
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
- Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

• Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
• Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$
• $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$
**Proof of Cycle Property**

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

*Proof.*

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
- Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
- $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$.
- So $C$ contains another cut edge $e'$ of $\{S, V - S\}$. 
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
- Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
- $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$.
- So $C$ contains another cut edge $e'$ of $\{S, V - S\}$.
- And $c(e') < c(e)$.
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

• Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
• Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
• $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$.
• So $C$ contains another cut edge $e'$ of $\{S, V - S\}$.
• And $c(e') < c(e)$.
• So, $T - \{e\} + \{e'\}$ is a spanning tree cheaper than $T$. 
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
- Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
- $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$.
- So $C$ contains another cut edge $e'$ of $\{S, V - S\}$.
- And $c(e') < c(e)$.
- So, $T - \{e\} + \{e'\}$ is a spanning tree cheaper than $T$.
- So $T$ is not a MCST of $G$. 

\[\square\]
Proof of Cycle Property

For any cycle in $G$, its highest-cost edge is in no MCST of $G$.

Proof.

- Suppose tree $T$ contains the highest-cost edge $e$ of cycle $C$.
- Let $\{S, V - S\}$ be the cut obtained by removing $e$ from $T$.
- $e \in C \cap E(S, V - S)$, so $|C \cap E(S, V - S)| > 0$.
- So $C$ contains another cut edge $e'$ of $\{S, V - S\}$.
- And $c(e') < c(e)$.
- So, $T - \{e\} + \{e'\}$ is a spanning tree cheaper than $T$.
- So $T$ is not a MCST of $G$.

Exchange Property!
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 13 Kruskal’s Algorithm

procedure $\text{Kruskal}(G, c())$  \hfill $\triangleright$ $G = (V, E)$ is connected
\begin{align*}
T &\leftarrow (V, \emptyset) \hfill \triangleright$ The eventual MCST
F &\leftarrow E
\end{align*}
while $|E(T)| < |V| - 1$ do
\begin{align*}
&\text{Remove cheapest edge } e \in F \text{ from } F \\
&\text{if } T + \{e\} \text{ does not contain a cycle then} \\
&\quad \text{Add } e \text{ to } T
\end{align*}
end procedure
Proof of Correctness of Kruskal

Theorem

Kruskal produces a minimum-cost spanning tree of G.
Proof of Correctness of Kruskal

**Theorem**

*Kruskal produces a minimum-cost spanning tree of G.*

The proof has two parts

1. Show \( T \) is a tree by showing it has no cycles and is connected
2. Show \( T \) is minimum-cost by showing each of its edges is contained in every MCST

- \( T \) is a forest at all times: new edges don’t create cycles
- If \( T \) is not connected at top of loop, then \(|E(T)| < |V| - 1\)
- Let \( S \) be the vertex set of a connected component of \( T \). Note that \( \{S, V - S\} \) is a cut of \( G \).
- \( G \) is connected, so \( E(S, V - S) \neq \emptyset \)
- But \(|F|\) decreases at each iteration, so loop must stop repeating, so \( T \) is a tree
Proof of Correctness of Kruskal

Theorem

Kruskal produces a minimum-cost spanning tree of $G$. The proof has two parts

- Show $T$ is a tree by showing it has no cycles and is connected
Proof of Correctness of Kruskal

Theorem

Kruskal produces a minimum-cost spanning tree of G.

The proof has two parts

- Show $T$ is a tree by showing it has no cycles and is connected
- Show $T$ is minimum-cost by showing each of its edges is contained in every MCST
Proof of Correctness of Kruskal

**Theorem**

*Kruskal produces a minimum-cost spanning tree of* \( G \).

The proof has two parts

- Show \( T \) is a tree by showing it has no cycles and is connected
- Show \( T \) is minimum-cost by showing each of its edges is contained in every MCST

\( T \) has no cycles and is connected:
Proof of Correctness of Kruskal

**Theorem**
Kruskal produces a minimum-cost spanning tree of $G$.

The proof has two parts
- Show $T$ is a tree by showing it has no cycles and is connected
- Show $T$ is minimum-cost by showing each of its edges is contained in every MCST

$T$ has no cycles and is connected:
- $T$ is a forest at all times: new edges don’t create cycles
Proof of Correctness of Kruskal

Theorem
Kruskal produces a minimum-cost spanning tree of G.

The proof has two parts

• Show $T$ is a tree by showing it has no cycles and is connected
• Show $T$ is minimum-cost by showing each of its edges is contained in every MCST

$T$ has no cycles and is connected:

• $T$ is a forest at all times: new edges don’t create cycles
• If $T$ is not connected at top of loop, then $|E(T)| < |V| - 1$, so loop repeats
Proof of Correctness of Kruskal

**Theorem**

Kruskal produces a minimum-cost spanning tree of $G$.

The proof has two parts

- Show $T$ is a tree by showing it has no cycles and is connected
- Show $T$ is minimum-cost by showing each of its edges is contained in every MCST

$T$ has no cycles and is connected:

- $T$ is a forest at all times: new edges don’t create cycles
- If $T$ is not connected at top of loop, then $|E(T)| < |V| - 1$, so loop repeats
- Let $S$ be the vertex set of a connected component of $T$. Note that $\{S, V - S\}$ is a cut of $G$. 
Proof of Correctness of Kruskal

Theorem

Kruskal produces a minimum-cost spanning tree of G.

The proof has two parts

• Show $T$ is a tree by showing it has no cycles and is connected
• Show $T$ is minimum-cost by showing each of its edges is contained in every MCST

$T$ has no cycles and is connected:

• $T$ is a forest at all times: new edges don’t create cycles
• If $T$ is not connected at top of loop, then $|E(T)| < |V| - 1$, so loop repeats
• Let $S$ be the vertex set of a connected component of $T$. Note that $\{S, V - S\}$ is a cut of $G$.
• $G$ is connected, so $E(S, V - S) \neq \emptyset$, so $F \neq \emptyset$
Proof of Correctness of Kruskal

Theorem
Kruskal produces a minimum-cost spanning tree of G.

The proof has two parts

• Show $T$ is a tree by showing it has no cycles and is connected
• Show $T$ is minimum-cost by showing each of its edges is contained in every MCST

$T$ has no cycles and is connected:

• $T$ is a forest at all times: new edges don’t create cycles
• If $T$ is not connected at top of loop, then $|E(T)| < |V| - 1$, so loop repeats
• Let $S$ be the vertex set of a connected component of $T$. Note that $\{S, V - S\}$ is a cut of $G$.
• $G$ is connected, so $E(S, V - S) \neq \emptyset$, so $F \neq \emptyset$
• But $|F|$ decreases at each iteration, so loop must stop repeating, so $T$ is a tree
Proof of Correctness of Kruskal

$T$ is an MCST:
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subseteq V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$. 
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subset V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$.
- At this point $T$ contains no edge from $S$ to $V - S$
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subset V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$.
- At this point $T$ contains no edge from $S$ to $V - S$
- So, $e$ is the cheapest cut edge of $E(S, V - S)$ in $G$. 
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subseteq V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$.
- At this point $T$ contains no edge from $S$ to $V - S$.
- So, $e$ is the cheapest cut edge of $E(S, V - S)$ in $G$.
- So $e$ is part of every MCST of $G$. 
**Proof of Correctness of Kruskal**

\(T\) is an MCST:

- Let \(e = \{u, v\}\) be an edge selected by Kruskal.
- Let \(S \subset V\) be the set of vertices reachable from \(u\) in \(T\) just before \(e\) was added to \(T\).
- At this point \(T\) contains no edge from \(S\) to \(V - S\).
- So, \(e\) is the cheapest cut edge of \(E(S, V - S)\) in \(G\).
- So \(e\) is part of every MCST of \(G\).
- So every edge of \(T\) is in every MCST of \(G\).
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subset V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$.
- At this point $T$ contains no edge from $S$ to $V - S$.
- So, $e$ is the cheapest cut edge of $E(S, V - S)$ in $G$.
- So $e$ is part of every MCST of $G$.
- So every edge of $T$ is in every MCST of $G$.
- So $T$ is the only MCST of $G$!
Proof of Correctness of Kruskal

$T$ is an MCST:

- Let $e = \{u, v\}$ be an edge selected by Kruskal.
- Let $S \subseteq V$ be the set of vertices reachable from $u$ in $T$ just before $e$ was added to $T$.
- At this point $T$ contains no edge from $S$ to $V - S$.
- So, $e$ is the cheapest cut edge of $E(S, V - S)$ in $G$.
- So $e$ is part of every MCST of $G$.
- So every edge of $T$ is in every MCST of $G$.
- So $T$ is the only MCST of $G$!

Corollary

A graph without repeated edge lengths has a unique MCST.
Moderate Greed: Prim’s Algorithm

Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans.
Moderate Greed: Prim’s Algorithm

Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans. Prim maintains a subtree $T = (V', E')$ of $G$ and adds the cheapest cut edge of $E(V', V - V')$ (in $G$) to $T$. 
**Moderate Greed: Prim’s Algorithm**

Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans. Prim maintains a subtree $T = (V', E')$ of $G$ and adds the cheapest cut edge of $E(V', V - V')$ (in $G$) to $T$.

---

**Algorithm 16** Prim’s Algorithm

```plaintext
procedure PRIM(G, c())  // G = (V, E) is connected
    Select some $v \in V$; $V' \leftarrow \{v\}$; $T \leftarrow (V', \emptyset)$  // The eventual MCST
    while $|E(T)| < |V| - 1$ do
        Select cheapest edge $e \in E(V', V - V')$
        Add $e$ to $T$  // This adds a new vertex to $T$
    end procedure
```

---
Analysis of Prim’s Algorithm

**Theorem**
Prim produces a minimum-cost spanning tree of G.

**Proof.**
Analysis of Prim’s Algorithm

Theorem

Prim produces a minimum-cost spanning tree of $G$.

Proof.

• $T$ is a tree at all times, and $T$ eventually must span $G$, since $G$ is connected.
Analysis of Prim’s Algorithm

Theorem
Prim produces a minimum-cost spanning tree of $G$.

Proof.

- $T$ is a tree at all times, and $T$ eventually must span $G$, since $G$ is connected.
- The next edge added to $T$ is, in $G$, the cheapest cut edge for some cut.
Analysis of Prim’s Algorithm

Theorem
Prim produces a minimum-cost spanning tree of G.

Proof.

• $T$ is a tree at all times, and $T$ eventually must span $G$, since $G$ is connected.
• The next edge added to $T$ is, in $G$, the cheapest cut edge for some cut.
• That edge, therefore, must be in every MCST
Analysis of Prim’s Algorithm

**Theorem**
Prim produces a minimum-cost spanning tree of $G$.

**Proof.**

- $T$ is a tree at all times, and $T$ eventually must span $G$, since $G$ is connected.
- The next edge added to $T$ is, in $G$, the cheapest cut edge for some cut.
- That edge, therefore, must be in every MCST
- As in Kruskal proof, $T$ is the only MCST of $G$
We can also construct an MCST by throwing away all of the most expensive edges.
**Reverse-Delete Algorithm**

We can also construct an MCST by throwing away all of the most expensive edges.

**Algorithm 18 Reverse-Delete Algorithm**

```plaintext
procedure ReverseDelete(G, c())  \( \triangleright \) \( G = (V, E) \) is connected

while \(|E(G)| > |V| - 1\) do
  Select most expensive edge \( e \in G \) that does not disconnect \( G \)
  Remove \( e \) from \( G \)
end procedure
```
**Reverse-Delete Algorithm**

We can also construct an MCST by throwing away all of the most expensive edges.

**Algorithm 19** Reverse-Delete Algorithm

```plaintext
procedure REVERSEDELETE(G, c())
    G = (V, E) is connected
    while |E(G)| > |V| − 1 do
        Select most expensive edge e ∈ G that does not disconnect G
        Remove e from G
    end procedure
```

When might you ever want to use this algorithm?
Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.
Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

Relaxing Assumption of Distinct Edge-Costs
Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

- For each set of edges having identical costs, perturb their costs by distinct positive values
Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

- For each set of edges having identical costs, perturb their costs by distinct positive values
- Ensure that the sum of all of the perturbations is tiny compared to the actual edge costs.
Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

- For each set of edges having identical costs, perturb their costs by distinct positive values.
- Ensure that the sum of all of the perturbations is tiny compared to the actual edge costs.
- Every spanning tree $T^*$ of perturbed graph $G^*$ corresponds to a spanning tree $T$ of $G$. 

Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

- For each set of edges having identical costs, perturb their costs by distinct positive values
- Ensure that the sum of all of the perturbations is tiny compared to the actual edge costs.
- Every spanning tree $T^*$ of perturbed graph $G^*$ corresponds to a spanning tree $T$ of $G$
- Correspondence preserves relative cost: $c(T_1^*) \leq c(T_2^*)$ iff $c(T_1) \leq c(T_2)$
Relaxing Assumption of Distinct Edge-Costs

Suppose $G$ does not have distinct edge costs.

Idea: Perturbation

- For each set of edges having identical costs, perturb their costs by distinct positive values.
- Ensure that the sum of all of the perturbations is tiny compared to the actual edge costs.
- Every spanning tree $T^*$ of perturbed graph $G^*$ corresponds to a spanning tree $T$ of $G$.
- Correspondence preserves relative cost: $c(T_1^*) \leq c(T_2^*)$ iff $c(T_1) \leq c(T_2)$.
- So $T^*$ is an MCST of $G^*$ iff $T$ is an MCST of $G$. 