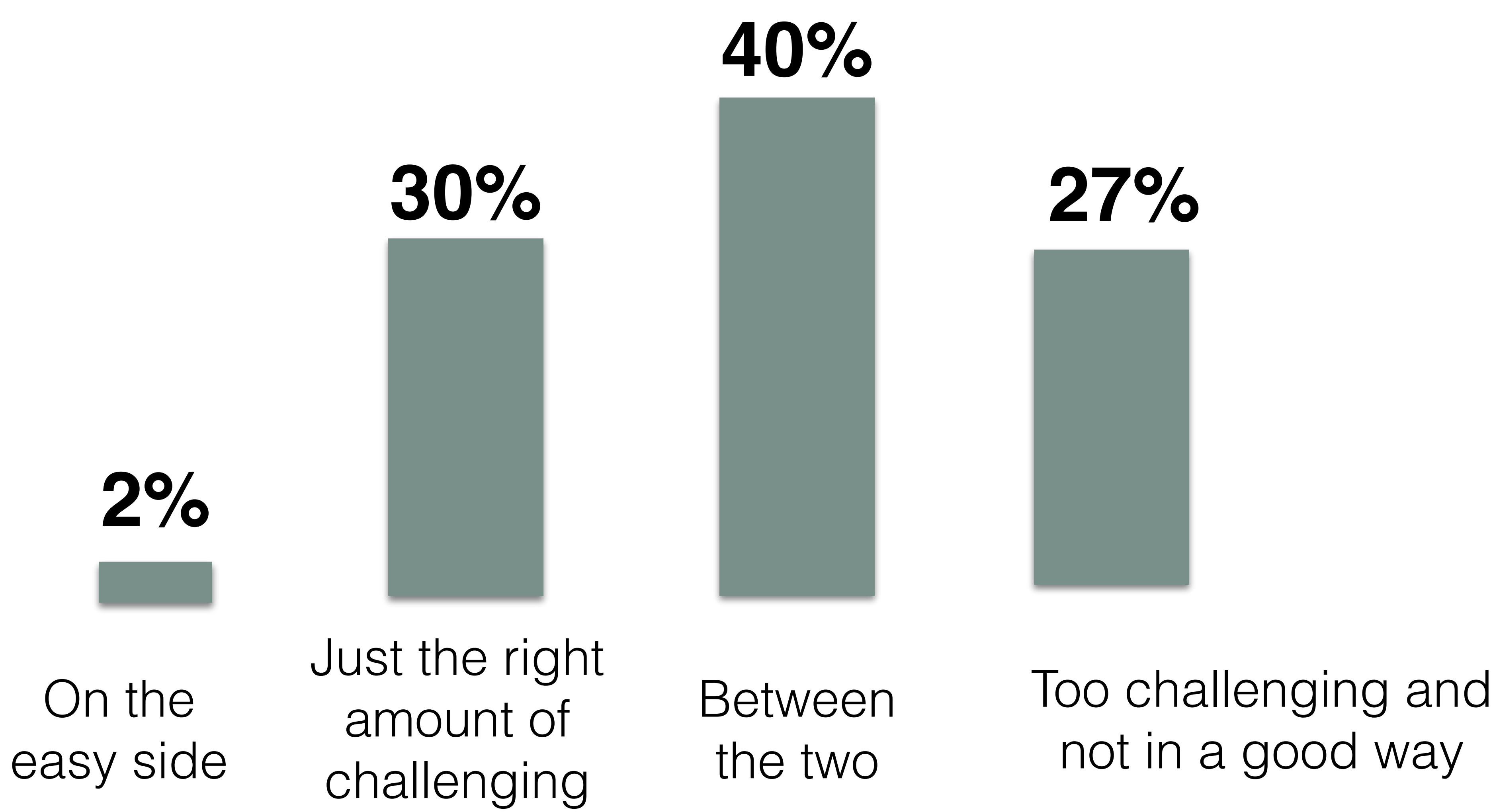
Greedy Algorithms: Minimizing Lateness

Reminders/Logistics

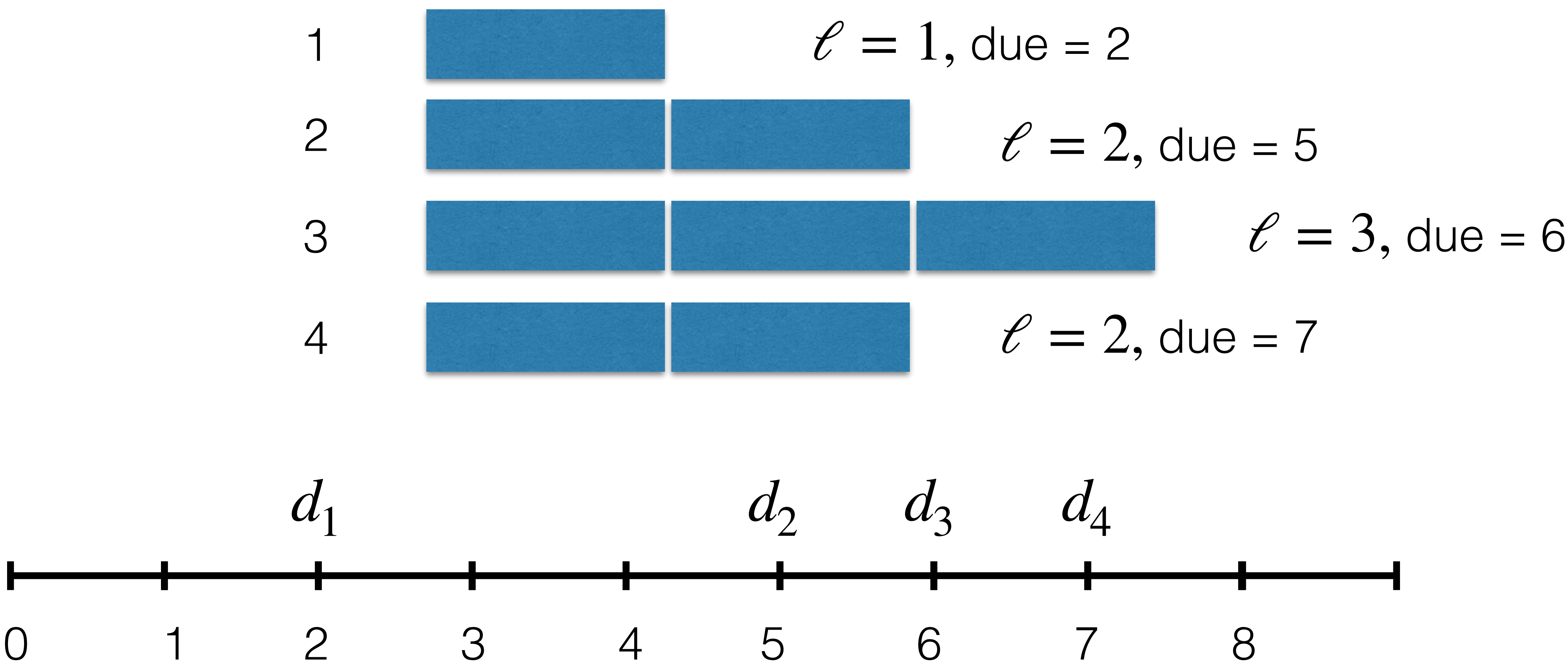
- CS256 URL: https://williams-cs.github.io/cs256-s21-www/
- Graded HW 1 will be returned today; solution on GLOW
- Feedback question response: Question 5 was a prime culprit



Minimizing Lateness: Exchange Argument

Minimizing Lateness: Problem

- You have n homework assignments of different lengths, with different deadlines
- How should you schedule your time to minimize lateness?
- Example:



Minimizing Lateness: Problem

Let's formalize the problem. The input is:

- A list of assignments or "jobs" that need to be scheduled
- Each job j has a **length** t_j and a **deadline** d_j

The output is a **schedule of all jobs**, what does it look like?

- s_j = start time for job j (selected by the algorithm)
- $f_j = s_j + t_j$ finish time

- Restrictions on the schedule:
  - Only one job can be scheduled at a given time
  - A job must run to completion before another can be executed

Minimizing Lateness: Problem

What makes a schedule good?

- Let us define lateness of job j as

$$\ell_j = \begin{cases} 0 & \text{if } f_j \le d_j \\ f_j - d_j & \text{if } f_j > d_j \end{cases}$$

- Maximum lateness $L = \max_j \ell_j$

Goal: Make maximum lateness as small as possible, minimize maximum lateness

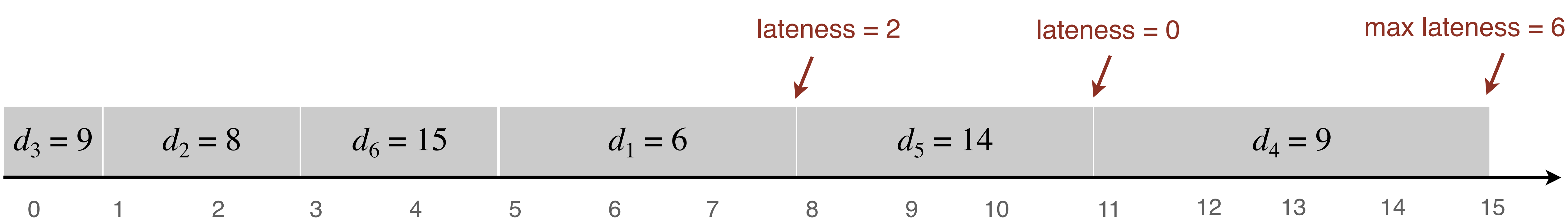
Minimizing Lateness: Problem

- Let us define lateness of job j as

$$\ell_j = \begin{cases} 0 & \text{if } f_j \leq d_j \\ f_j - d_j & \text{if } f_j > d_j \end{cases}$$

- Maximum lateness $L = \max_j \ell_j$

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Possible Greedy Approaches

- Observation:
    - Never hurts to schedule jobs consecutively with no "idle time" in between
    - Can start the first job at time 0
    - Schedule is then determined by order of jobs
    - What order should we pick?

Possible Greedy Approaches

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

Shortest job first (Counter-example)

	1	2
t_j	1	10
d_j	100	10

Gives max lateness: 1
OPT max lateness: 0

Possible Greedy Approaches

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

Shortest slack first (Counter-example)

	1	2
t_j	1	10
d_j	2	10

Gives max lateness: 9
OPT max lateness: 1

Possible Greedy Approaches

Earliest deadline first?

- How all computer scientists schedule their work
- Order jobs by their deadline $d_1 \leq d_2 \leq \ldots \leq d_n$ and schedule them in that order
    - **Intuition:** get the jobs due first done first
- This approach is optimal (We will show this)

Greedy Solution

Assuming jobs are ordered by deadline $d_1 \leq d_2 \leq \ldots \leq d_n$ the greedy ordering is simply $G = 1,2,\ldots,n$

- All jobs are scheduled consecutively (no idle time)

Claim: G is optimal, that is, the earliest-deadline-first algorithm produces a schedule that minimizes maximum lateness

- Will prove this through proof by exchange argument
- High level idea: assume G is not optimal, then there exists an optimal solution $O \neq G$ that produces a different ordering or jobs; we will show that we can modify O to produce G one job at a time, without ever increasing lateness

Exchange Argument: I

Let O be an optimal solution, such that $O \neq G$.

- Since $G = \{1,2,\ldots,n\}$, where $d_1 \leq d_2 \leq \ldots \leq d_n$ and $O \neq G$, O must have an inversion
  - A pair of jobs (i, j) is an **inversion** if job j is scheduled before i but i's deadline is earlier ($d_j > d_i$)

a schedule with an inversion

recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Adjacent Inversions

Observation. If an idle-free schedule has an inversion, then it has an adjacent inversion.

> Recall. i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

Proof. [Contradiction]

- Let i, j be any two **closest** non-adjacent inversions
- Let k be element immediately to the right of j.
- Case 1. $[d_j > d_k]$ Then j, k is an adjacent and closer inversion ($\Rightarrow\Leftarrow$)
- Case 2. $[d_j < d_k]$ Since $d_i < d_j$, this means that i, k is a closer inversion ($\Rightarrow\Leftarrow$) ∎

	j	k			i		

Exchange Argument: II

Let O be an optimal solution, such that $O \neq G$.

- Since $G = \{1, 2, \ldots, n\}$, and $O \neq G$, O must have an inversion
- O must have at least one adjacent inversion
- **Claim**: We can swap adjacent inverted jobs without increasing maximum lateness.

a schedule with an inversion

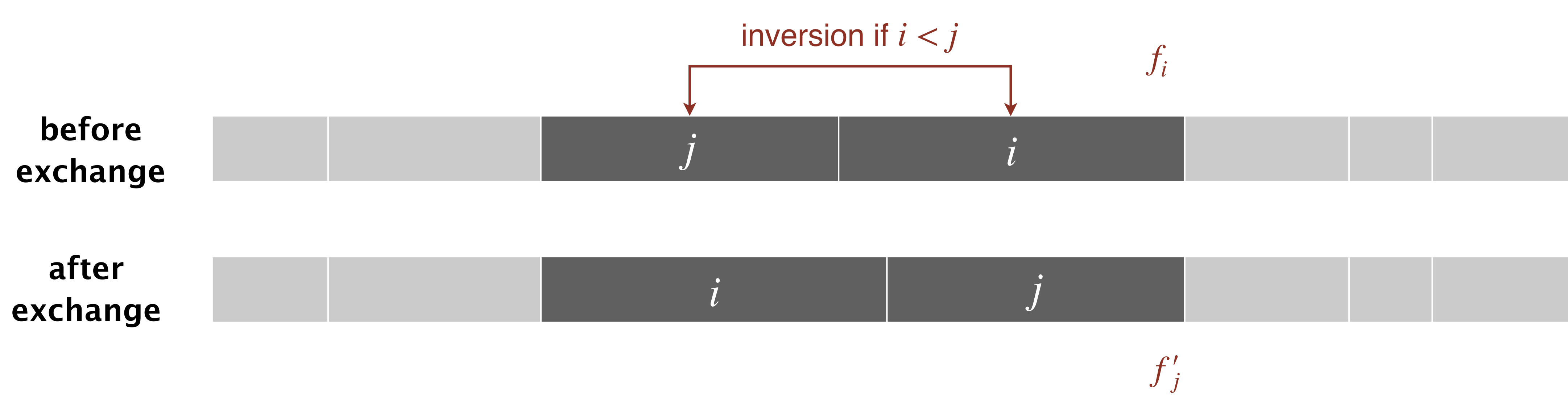
recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Swapping Inversions

Claim: We can swap adjacent inverted jobs without increasing maximum lateness.

Proof. Let i, j be adjacent inverted jobs with $i < j$. Let ℓ be the lateness before swapping them and ℓ' after the swap.

- $\ell_k = \ell'_k \qquad \forall k \neq i, j$ (swap doesn't affect other jobs)
- $\ell'_i \leq \ell_i \qquad$ (lateness of i improves after swap)
- $\ell'_j = \quad f'_j - d_j \quad = f_i - d_j \quad \leq f_i - d_i \quad \leq \ell_i$ ∎



Exchange Argument: III

Let O be an optimal solution, such that $O \neq G$.

- Since $G = \{1,2,\ldots,n\}$, and $O \neq G$, O must have an inversion
- O must have at least one adjacent inversion (i, j)
- **Claim**: We can swap adjacent inverted jobs (i, j) without increasing maximum lateness.
- Let O' be the new solution with adjacent inversion swapped, then
  - Max lateness of O' is no bigger than that of O (still optimal)
  - O' has one less inversion than O

O (optimal) $\rightarrow$ O' (optimal) $\rightarrow$ O'' (optimal) $\rightarrow \cdots \rightarrow$ G (optimal)

Exchange Argument: IV

Summarizing and final proof.

- We started with an optimal solution O that is different than greedy solution G
- Without loss of generality assume O has no idle time
- If O has an inversion, must be adjacent, exchanging them
  - decreases # of inversions by 1 without increasing max lateness, we repeat until no inversions
- G is a schedule with no idle time and no inversions
- Thus, we have transformed O to G without ever increasing maximum lateness
- Greedy is thus optimal ∎

Exchange Argument

General Pattern. An inductive exchange argument

- You start with an arbitrary optimal solution, that is different from the greedy solution
- Find the “first” place where the two solutions differ
- Argue that we can exchange the optimal choice for the greedy choice without making the solution worse (although the exchange may make it better)
- Show that you can iteratively perform the exchange step until you get the greedy solution

Greedy: Takeaway

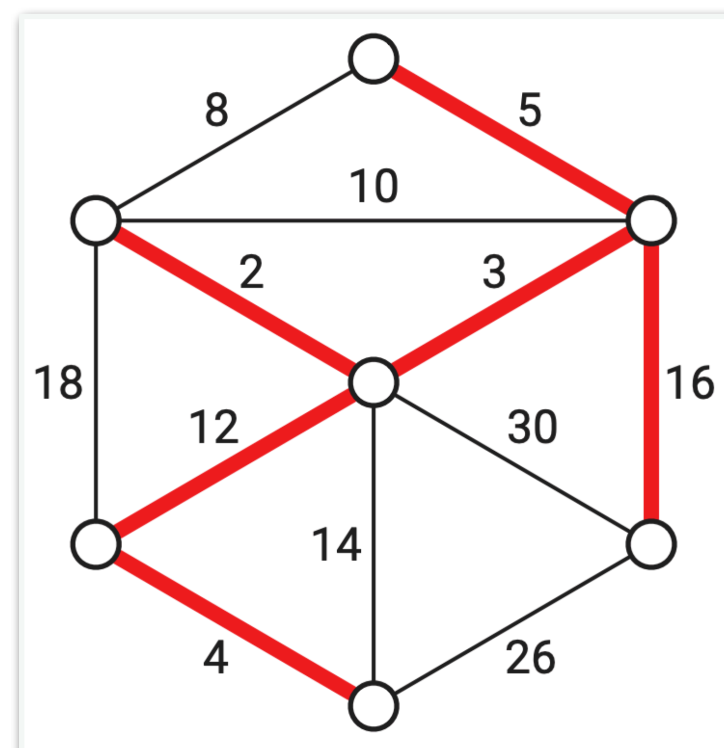
- Takeaway: greedy algorithms **do not usually** work
- When greedy algorithms work, it is because the problem has structure that greedy can take advantage of

Greedy Graph Algorithms:
Minimum Spanning Trees

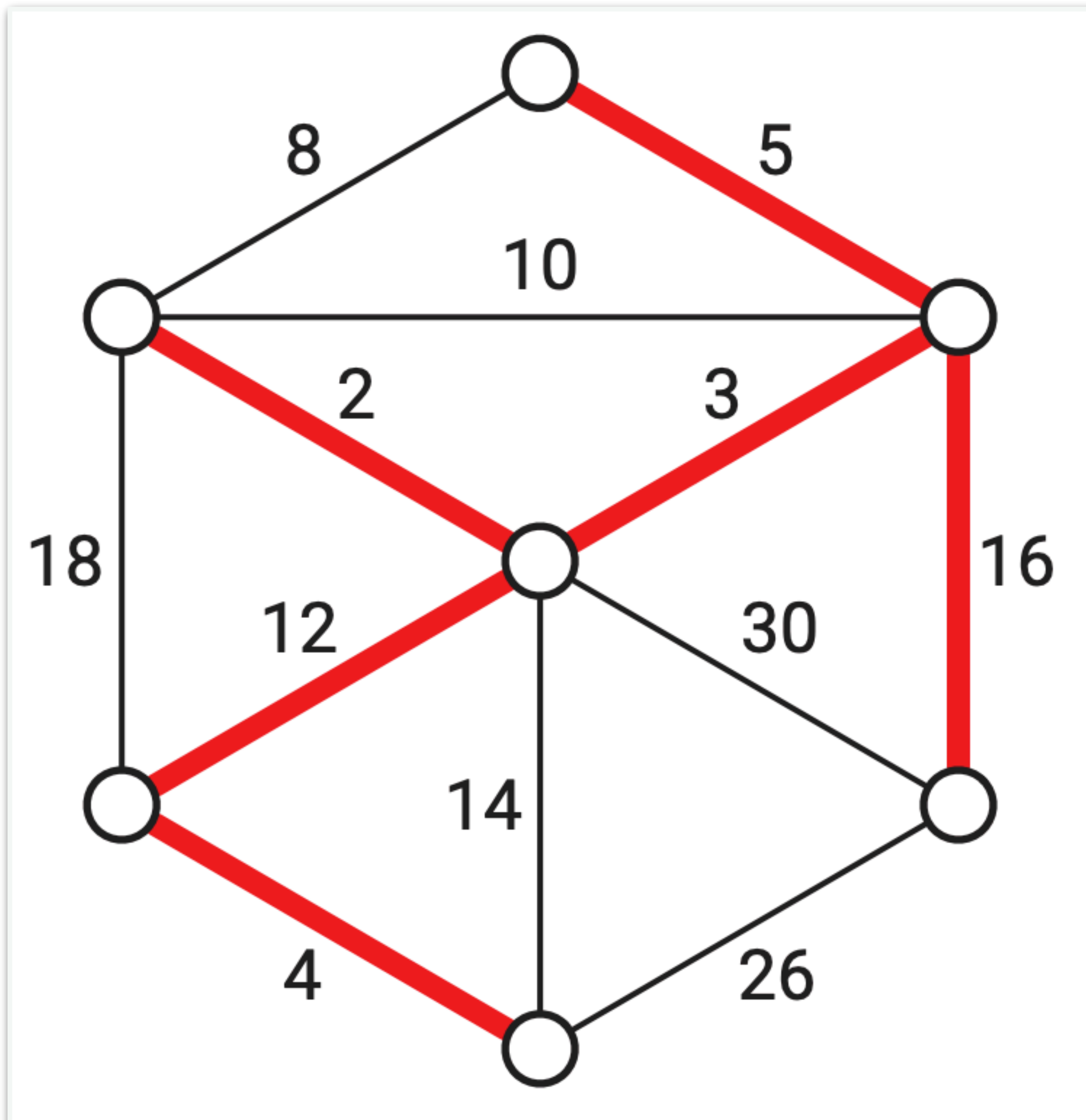
Minimum Spanning Trees

Problem. Given a connected, undirected graph $G = (V, E)$ with edge costs w_e, output **a minimum spanning tree**, i.e., set of edges $T \subseteq E$ such that

- (a spanning tree of G): T connects all vertices
- (has minimum weight): for any other spanning tree T' of G, we have $\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e$



Minimum Cost Spanning Trees



MST: Applications

- Many applications!
  - Classic application:
    - Underground cable (Power, Telecom, etc)
  - Efficient broadcasting on a computer network
  - Approximate solutions to harder problems, such at TSP
  - Real-time face verification

Distinct Edge Weights

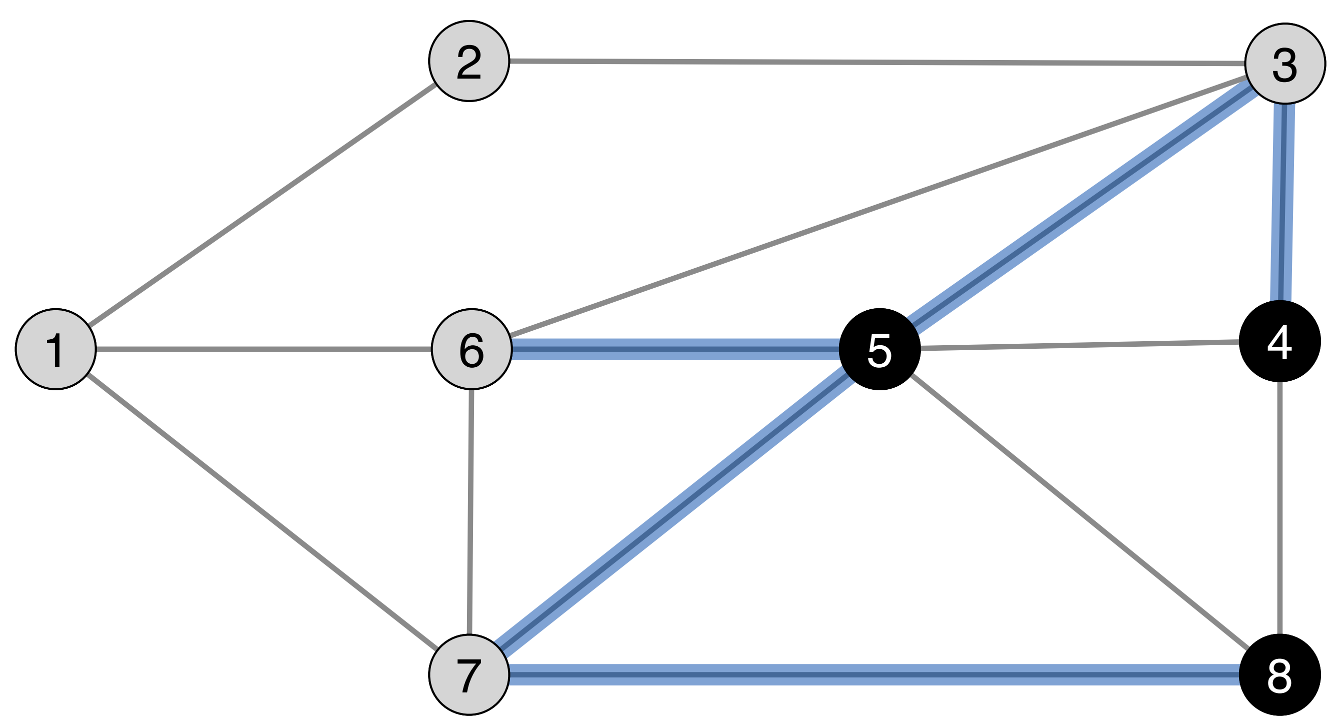
- Annoying subtlety in the problem statement is there may be multiple minimum spanning trees
    - If a graph has edges with same edge, e.g., all edges have weight 1: all spanning trees are min!
- To simplify discussion in our algorithm design, we will assume distinct edge weights

Lemma. If all edge weights in a connected graph are distinct, then it has a unique minimum spanning tree.

We will relax the distinct-edge-weight assumption later.

Spanning Trees and Cuts

A **cut** is a partition of the vertices into two **nonempty** subsets S and $V - S$. A **cut edge** of a cut S is an edge with one end point in S and another in $V - S$.



cut S = { 4, 5, 8 }

Cut edges = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

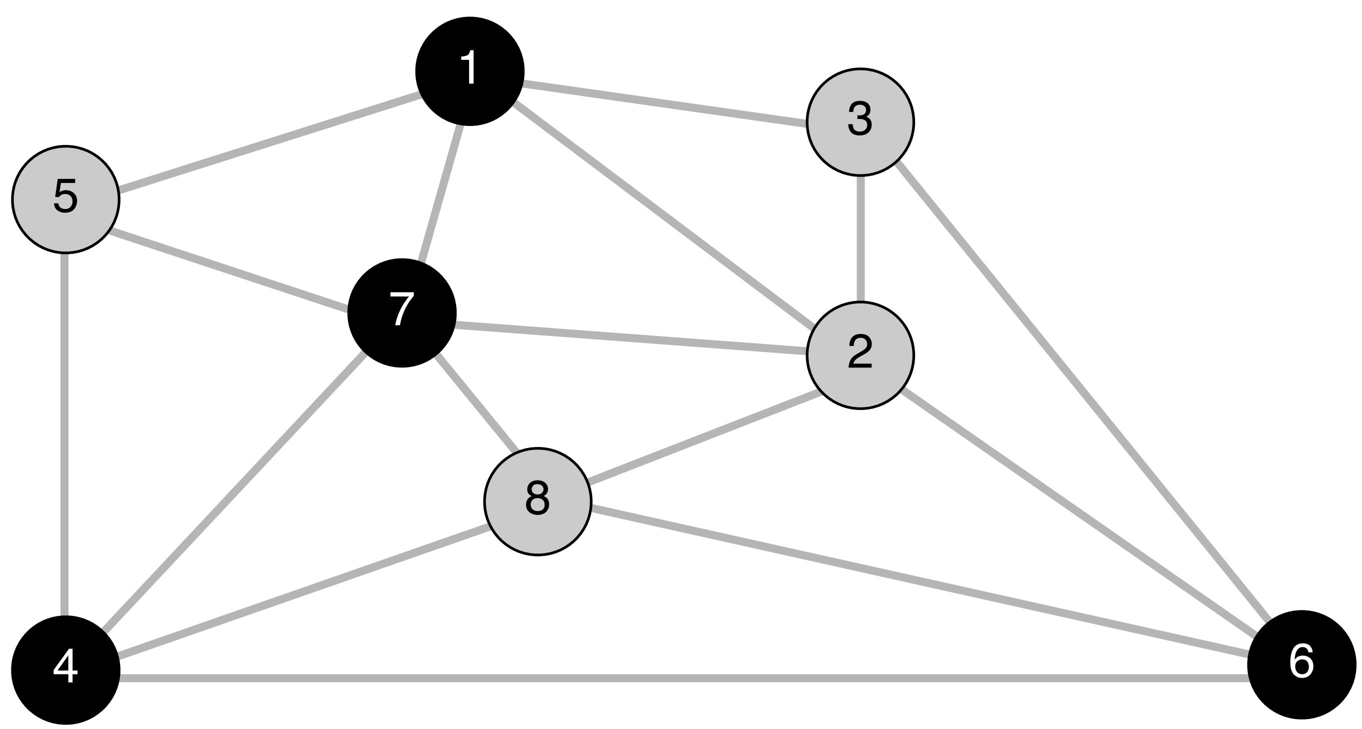
Spanning Trees and Cuts

Question. Consider the cut $S = \{1,4,6,7\}$. Which of the following edges are cut edges with respect to this cut?

A. (1, 7)

B. (5, 7)

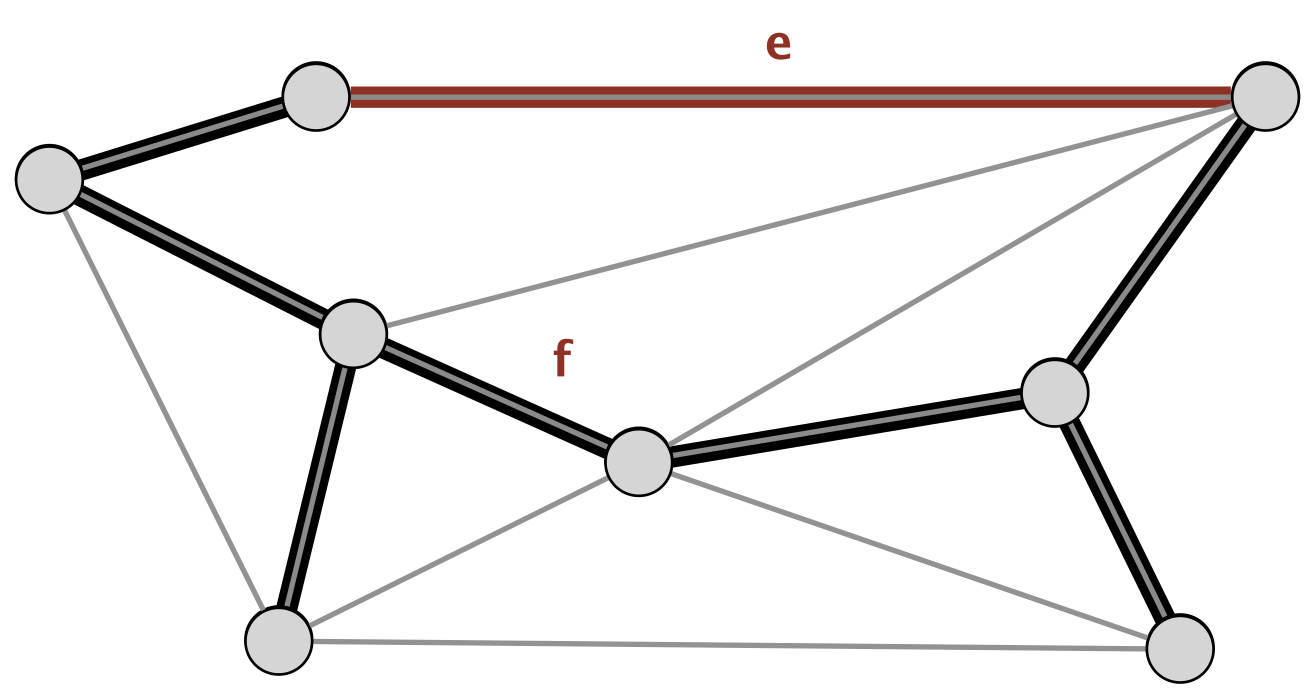
C. (2, 3)



Fundamental Cycle

Let T be a spanning tree of G.

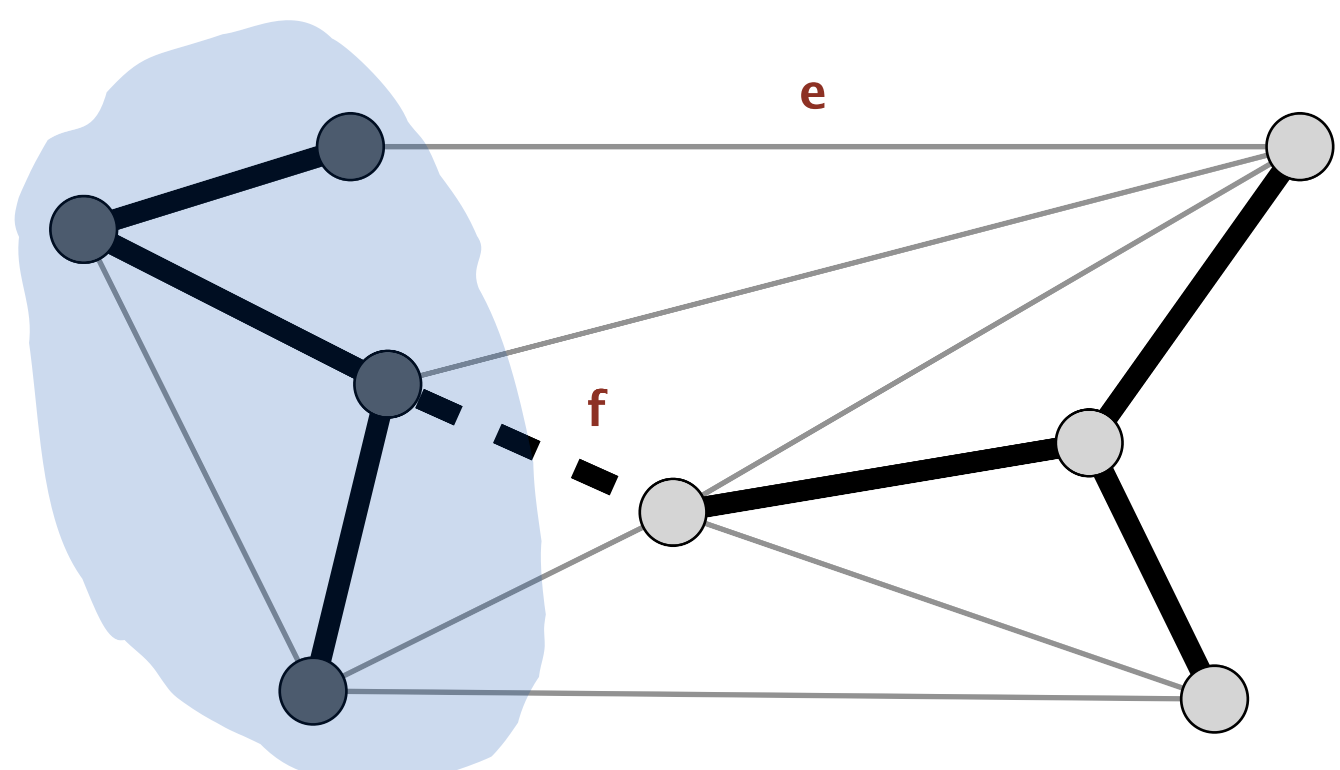
- For any edge $e \notin T$, $T \cup \{e\}$ creates a unique cycle C
- For any edge $f \in C : T \cup \{e\} - \{f\}$ is a spanning tree



Fundamental Cut

Let T be a spanning tree of G.

- For any edge $f \in T$, $T - \{f\}$ breaks the graph into two connected components, let D be the set of cut edges with end points in each component
- For any edge $e \in D : T - \{f\} \cup \{e\}$ is a spanning tree



Spanning Trees and Cuts

Lemma (Cut Property). For any cut $S \subset V$, let $e = (u, v)$ be the minimum weight edge connecting any vertex in S to a vertex in $V - S$, then every minimum spanning tree must include e.

Proof. (By contradiction)

Suppose T is a spanning tree that does not contain $e = (u, v)$.

Main Idea: We will construct another spanning tree $T' = T \cup e - e'$ with weight less than T ($\Rightarrow\!\Leftarrow$)

How to find such an edge e'?

Spanning Trees and Cuts

Proof (Cut Property).

Suppose T is a spanning tree that does not contain $e = (u, v)$.

- Adding e to T results in a unique cycle C
- C must "enter" and "leave" cut S, that is, $\exists e' = (u', v') \in C$ s.t. $u' \in S, v' \in V - S$
- $w(e') > w(e)$ (why?)
- $T' = T \cup e - e'$ is a spanning tree (why?)
- $w(T') < w(T)$

$(\Rightarrow\Leftarrow)$ ∎

Acknowledgments

- The pictures in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)