Greedy: Scheduling and Minimum Spanning Trees

Admin

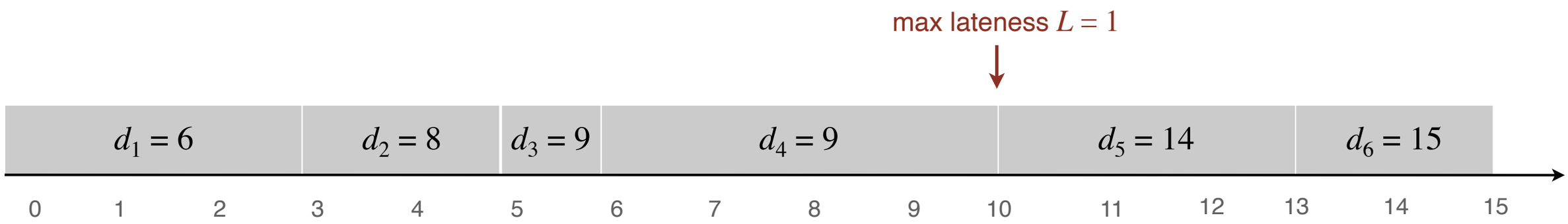
- Readings: we're a tad behind the schedule on the website—I'll revisit the schedule after mountain day

- Questions or comments?

Recall: Scheduling with Deadlines

Given interval length t_i and deadline d_i for $i \in \{1, \ldots, n\}$ jobs, schedule all tasks, that is, assign start and finish times $(t_i, d_i) \rightarrow (s_i, f_i)$, where $f_i = s_i + t_i$, so as to minimize the maximum lateness.

- Lateness of process i : $L_i = \max\{0, f_i - d_i\}$



Greedy: Earliest-Deadline First

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT jobs by due times and renumber so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

FOR $j = 1$ TO n

  Assign job j to interval $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

  $t \leftarrow t + t_j$.

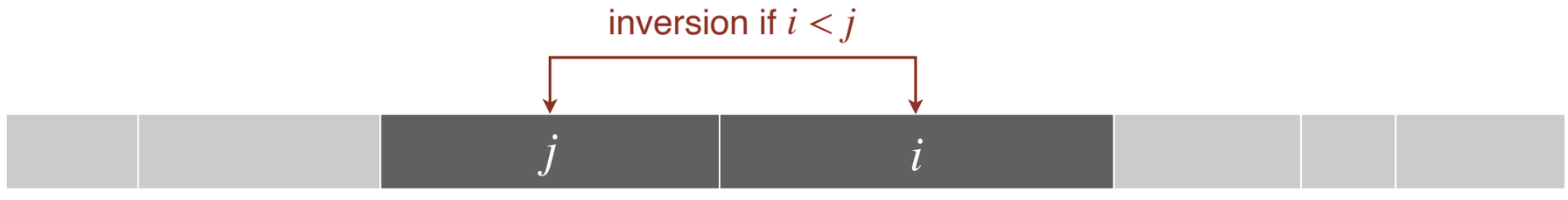
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.

Minimizing Lateness: Greedy

Observations about our greedy algorithm

- It produces a schedule with no idle time
- It produces a schedule with no inversions
  - i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

a schedule with an inversion



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

Structure of the Solution

- **Notice:** All schedules with no inversions and no idle time have the same maximum lateness
  - Distinct deadlines, unique schedule
  - Non-distinct deadlines: Consider two jobs with deadline d; the maximum lateness does not depend on the order in which they are scheduled
- Say the two jobs have duration t_i, t_j and same deadline d
- If i is scheduled first at time s, the max lateness is: $\max\{0,(s + t_i + t_j) - d\}$
- If j is scheduled first at time s, the max lateness is the same: $\max\{0,(s + t_i + t_j) - d\}$

Where We Are Going

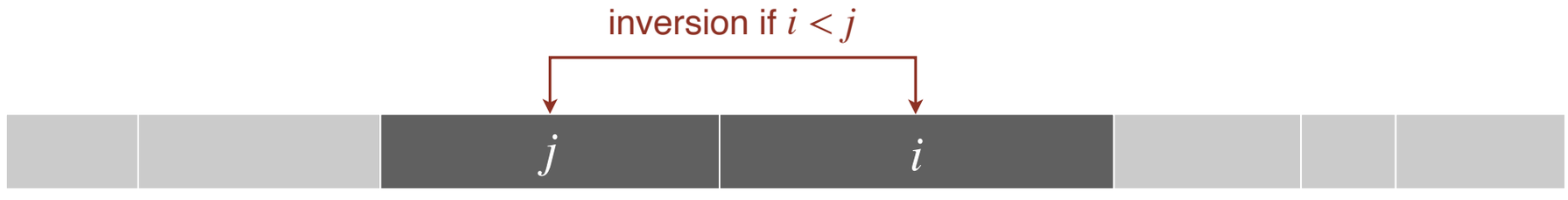
- **Notice:** All schedules with no inversions and no idle time have the same maximum lateness
  - Distinct deadlines, unique schedule
  - Non-distinct deadlines: Consider two jobs with deadline d; the maximum lateness does not depend on the order in which they are scheduled
- **Goal.** show there exists an optimal schedule with no inversions and no idle time
- Then, we have shown that the optimal schedule has the same maximum lateness as greedy!
- We will show this via **an exchange argument**
- Second proof technique to prove greedy is optimal

Minimizing Lateness: Greedy

Observations about our greedy algorithm

- It produces a schedule with no idle time
- It produces a schedule with no inversions
  - i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

a schedule with an inversion

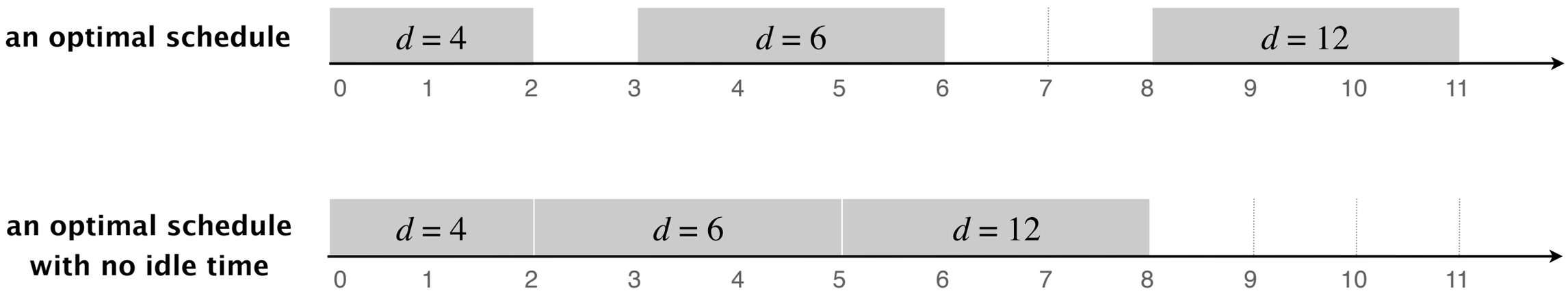


recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Structure of Optimal

Observation about optimal.

- There exists an optimal schedule with no idle time.
- (Can always schedule jobs earlier to prevent idleness!)



Structure of Optimal: 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 ($d_i < d_j$)

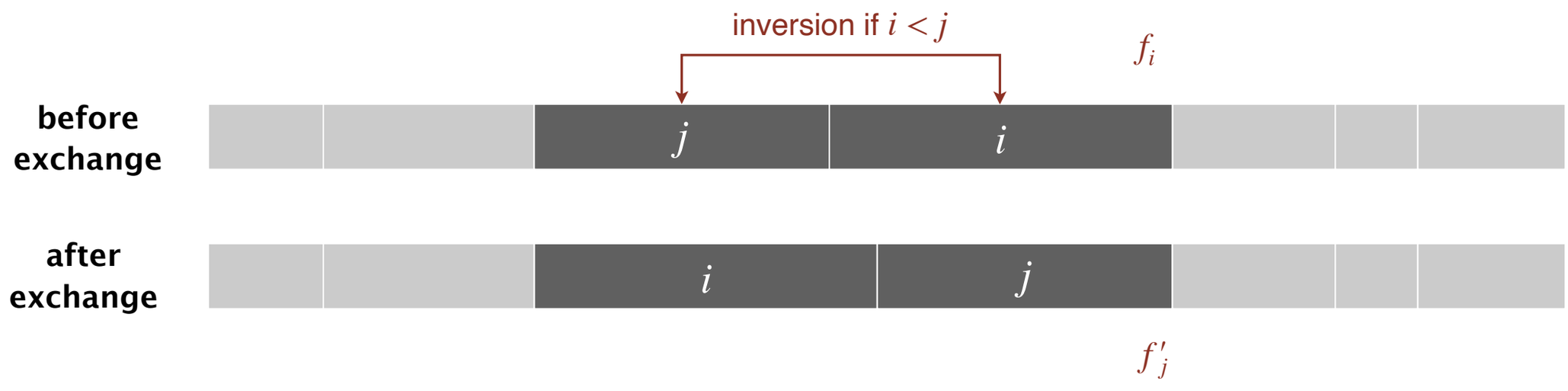
Proof. [Contradiction]

- Let i, j be any two non-adjacent inversions without another inversion between them
- 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 ($\Rightarrow\Leftarrow$) ∎

Structure of Optimal: Inversions

Claim. Given a schedule with k inversions, we can modify it to schedule with $k - 1$ inversions (without increasing the maximum lateness).

Proof. (Key Idea) Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the maximum lateness.

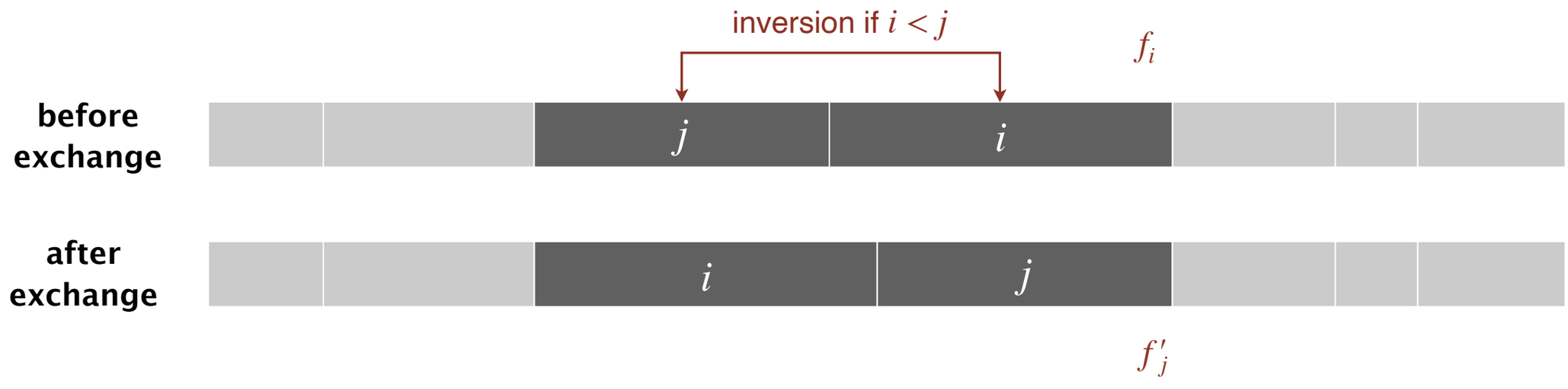


Structure of Optimal: Inversions

Claim. Given a schedule with k inversions, we can modify it to schedule with $k - 1$ inversions (without increasing the maximum lateness).

Proof. Let i, j be 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)
- $\ell'_i \leq \ell_i$ (lateness of i improves after swap)
- $\ell'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq \ell_i$ ∎



Optimality of Greedy

Summarizing the proof.

- All schedules with no inversions and no idle time have the same maximum lateness
- Greedy schedule has no inversions and no idle time
- Consider an optimal schedule $\mathcal{O}$, without loss of generality, we can assume that
  - $\mathcal{O}$ has no idle time
  - $\mathcal{O}$ has no inversions, why?
  - **[Iterate and exchange].** If there is an inversion, must be adjacent, exchanging them decreases # of inversions by 1 without increasing max lateness (repeat until no inversions)
- Greedy and $\mathcal{O}$ have same max lateness. ∎

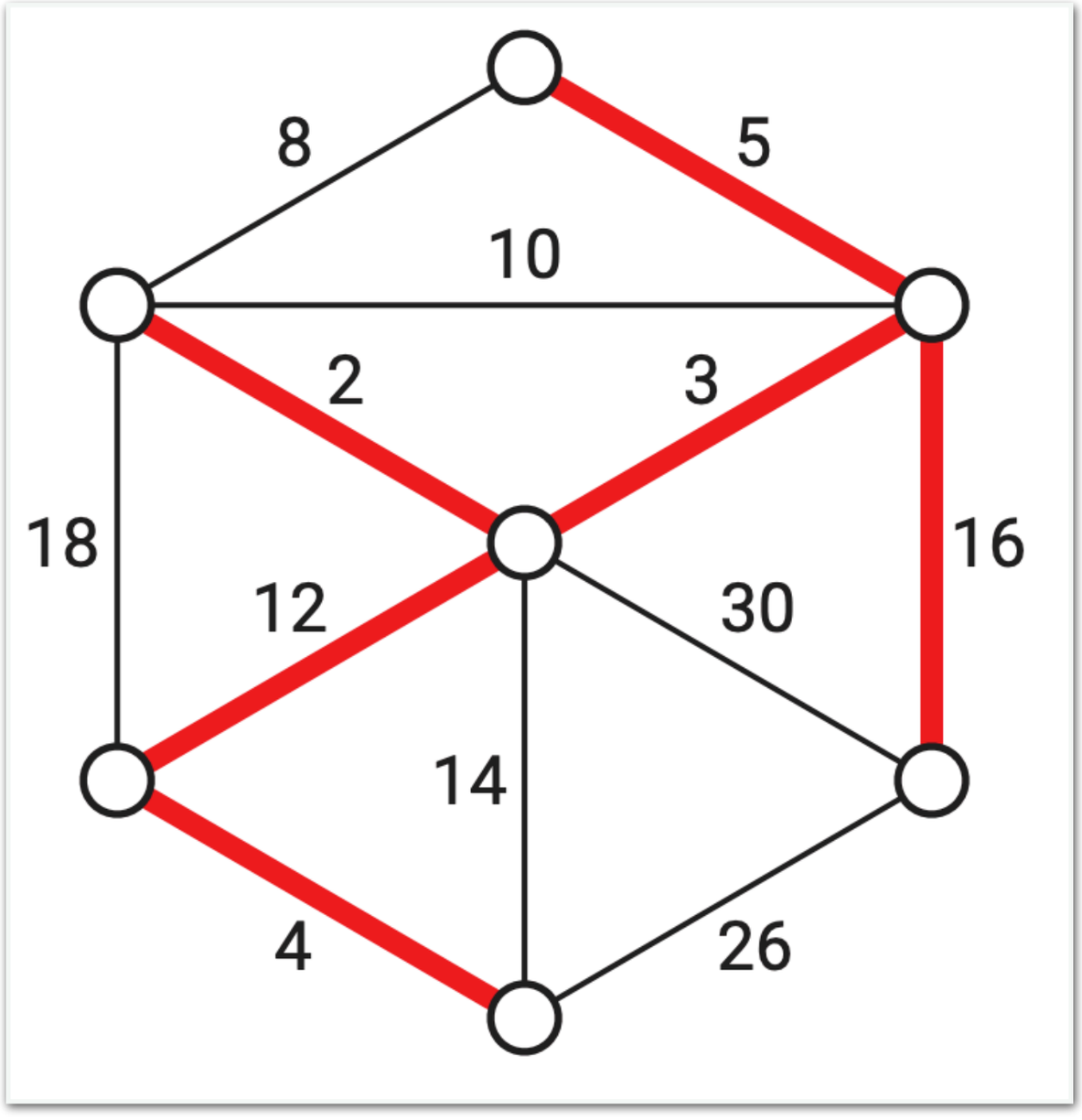
Exchange Argument

General Pattern.

- You start with an arbitrary optimal solution
- Prove that WLOG it must have certain nice properties
- Assume there is an optimal solution that is different from the greedy solution
- Find the “first” difference between the two solutions
- 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 Graph Algorithms: Minimum Spanning Trees

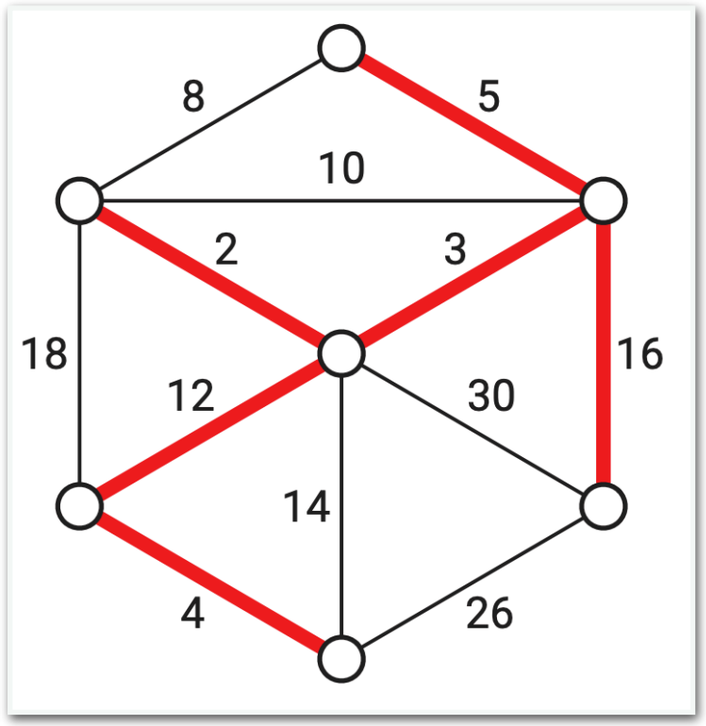
Minimum Cost Spanning Trees



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 $\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e$

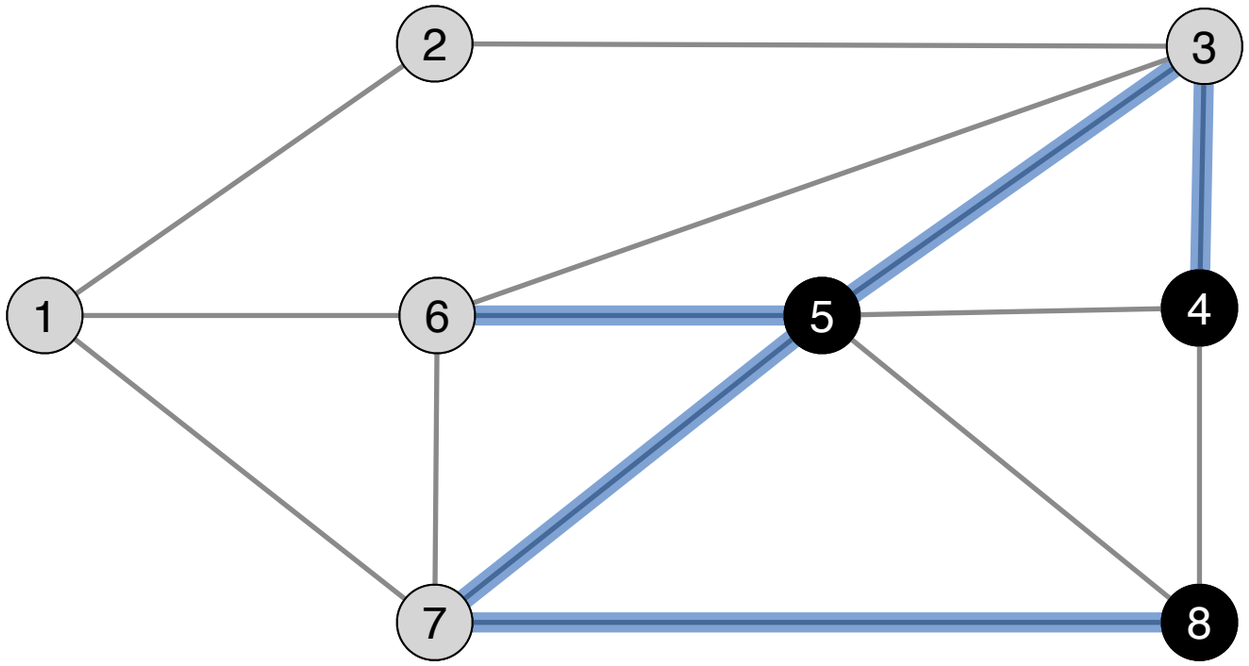


Minimum Spanning Trees

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

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) }

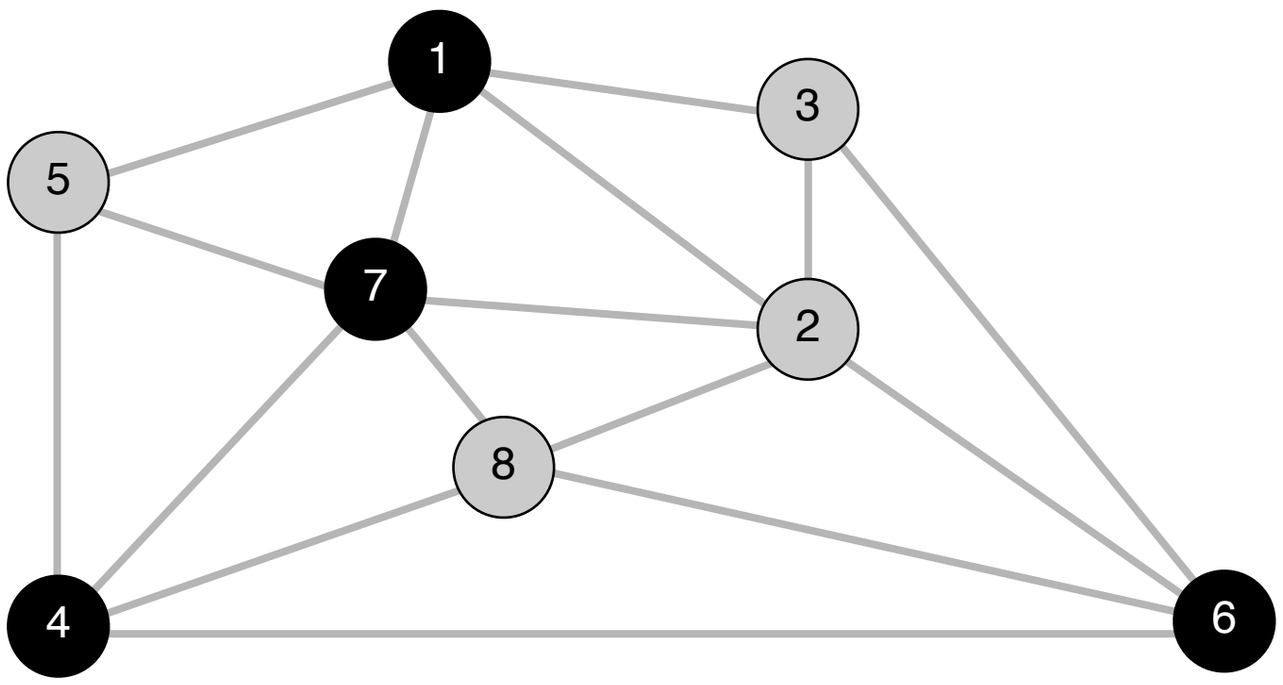
Spanning Trees and Cuts

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)

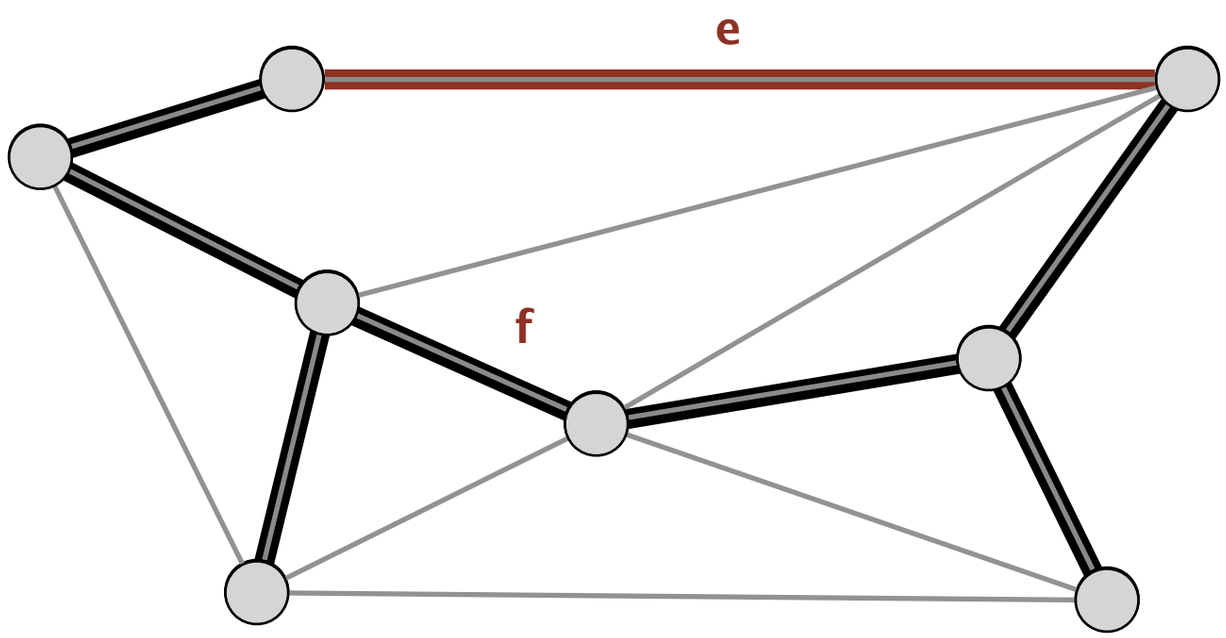
C. (2, 3)



Fundamental Cycle

Let T be a spanning tree of G.

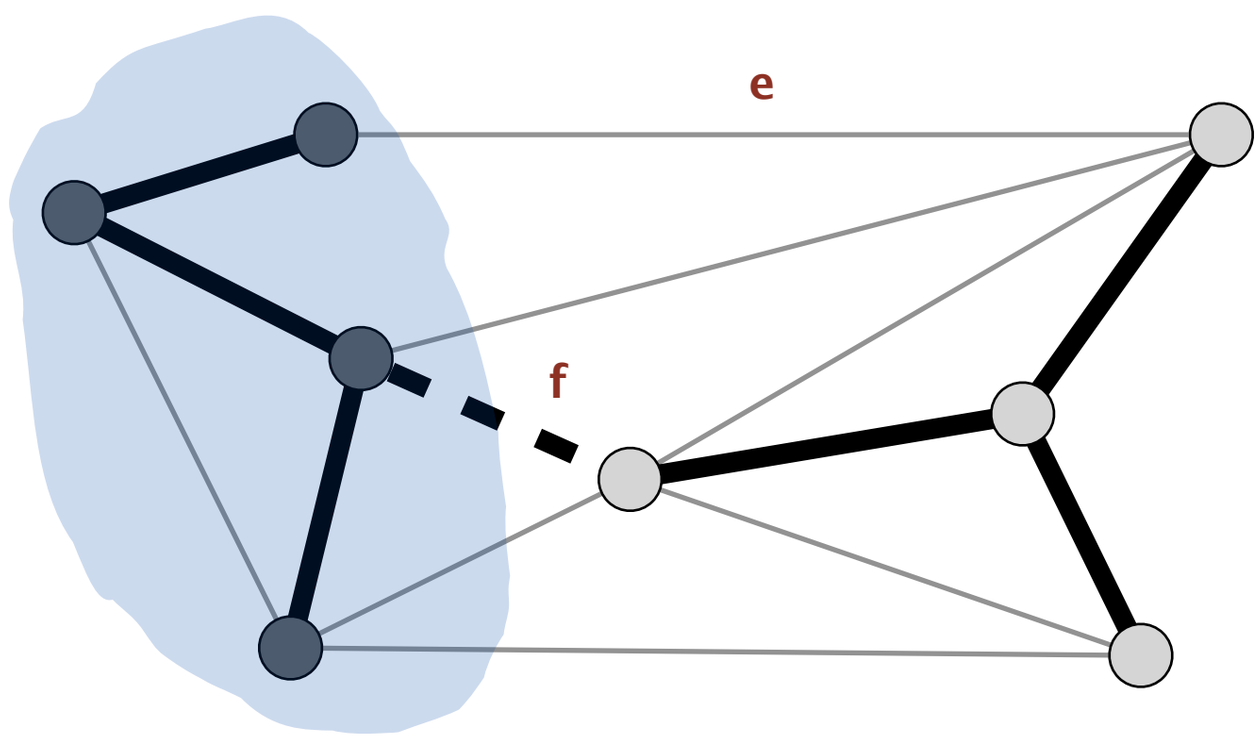
- For any edge $e \notin T$, $T \cup \{e\}$ creates a unique cycle C
- For any edge $f \in C$: $T \cup \{e\} - \{f\}$ is a spanning tree



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 $S \subset V$, if $e = (u, v)$ is the strictly smallest edge connecting any vertex in S to a vertex in $V - S$, then every minimum spanning tree must include e.

Proof. (By contradiction via an exchange argument)

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'?

Exchange argument

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 (why?)
- 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?)
- $w(T') < w(T)$

$(\Rightarrow\Leftarrow)$ ∎

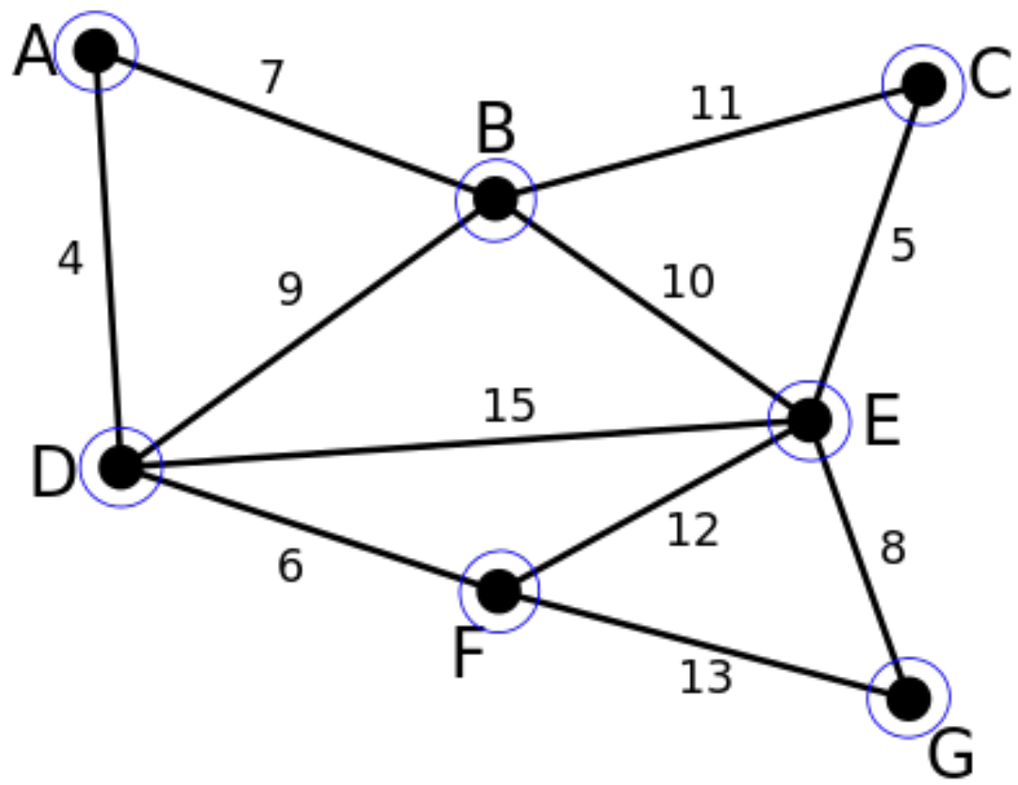
Spanning Trees and Cuts

- What if there's no unique minimum?

Lemma (Cut Property). For any cut $S \subset V$, if $e_1, e_2, \ldots, e_k$ are the smallest edges connecting any vertex in S to a vertex in $V - S$, then every minimum spanning tree must include some e_i.

Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
    - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
    - $T \leftarrow T \cup \{e\}$
    - $S \leftarrow S \cup \{v\}$



Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
    - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
    - $T \leftarrow T \cup \{e\}$
    - $S \leftarrow S \cup \{v\}$
- **Implementation crux.** Find and add min-cost edge for the cut $(S, V - S)$ and add it to the tree in each iteration, update cut edges
- How can we prove that this finds the MST?
    - Cut property! (On board.)

Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
    - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
    - $T \leftarrow T \cup \{e\}$
    - $S \leftarrow S \cup \{v\}$
- **Implementation crux.** Find and add min-cost edge for the cut $(S, V - S)$ and add it to the tree in each iteration, update cut edges
- Running time?
    - Naive implementation may take $O(nm)$
    - Need to maintain set of edges adjacent to nodes in T and extract min-cost cut edge from it each time
    - Which data structure from CS 136 can we use?

CS136 Review: Priority Queue

Managing such a set typically involves the following operations on S

- **Insert.** Insert a new element into S
- **Delete.** Delete an element from S
- **ExtractMin.** Retrieve highest priority element in S

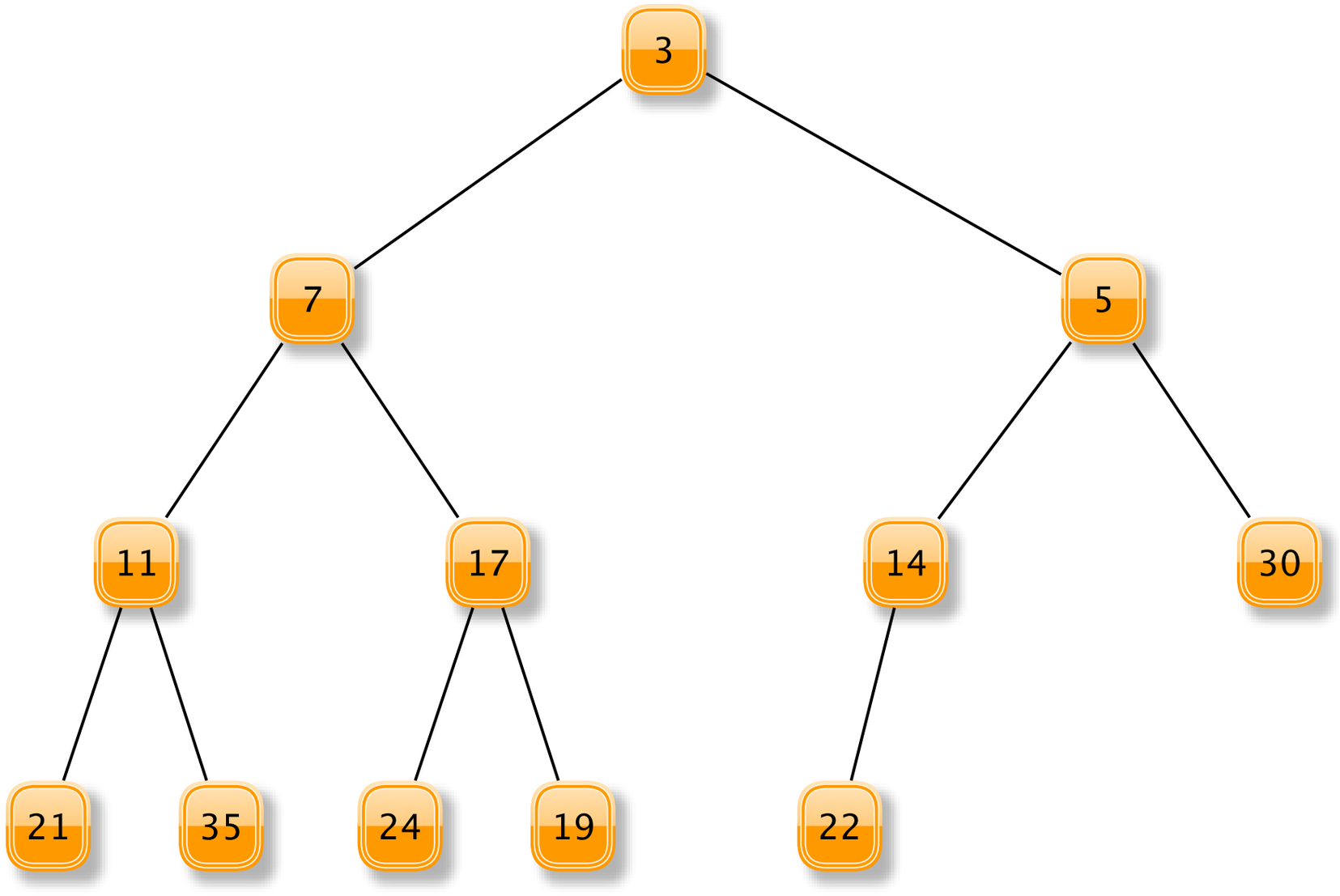
Priorities are encoded as a ‘key’ value

Typically: higher priority <—> lower key value

Heap as Priority Queue. Combines tree structure with array access

- Insert and delete: $O(\log n)$ time (‘tree’ traversal & moves)
- **Extract min.** Delete item with minimum key value: $O(\log n)$

Heap Example



H

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	3	7	5	11	17	14	30	21	35	24	19	22	-	-	-

“Prims” Implementation

- Use **Binary heaps**
  - Create a priority queue initially holding all edges incident to u.
  - At each step, dequeue edges from the priority queue until we find an edge (x, y) where $x \in S$ and$y \notin S$.
  - Add (x, y) to T.
  - Add to the queue all edges incident to y whose endpoints aren't in S.
  - Each edge is enqueued and dequeued at most once
  - Total runtime: $O(m \log m)$
    - In any graph, $m = O(n^2)$
    - So $O(m \log m) = O(m \log n)$

“Prims” Implementation

- Implementation using **Binary heaps**
  - Total runtime: $O(m \log n)$

- If a **Fibonacci heap** is used instead of binary heap:
  - Runs in $O(m + n \log n)$ **“amortized time”**
  - Support amortized $O(1)$-time inserts, $O(\log n)$ time extract min

Definition. If k operations take total time $O(t \cdot k)$, then the amortized time per operation is $O(t)$.

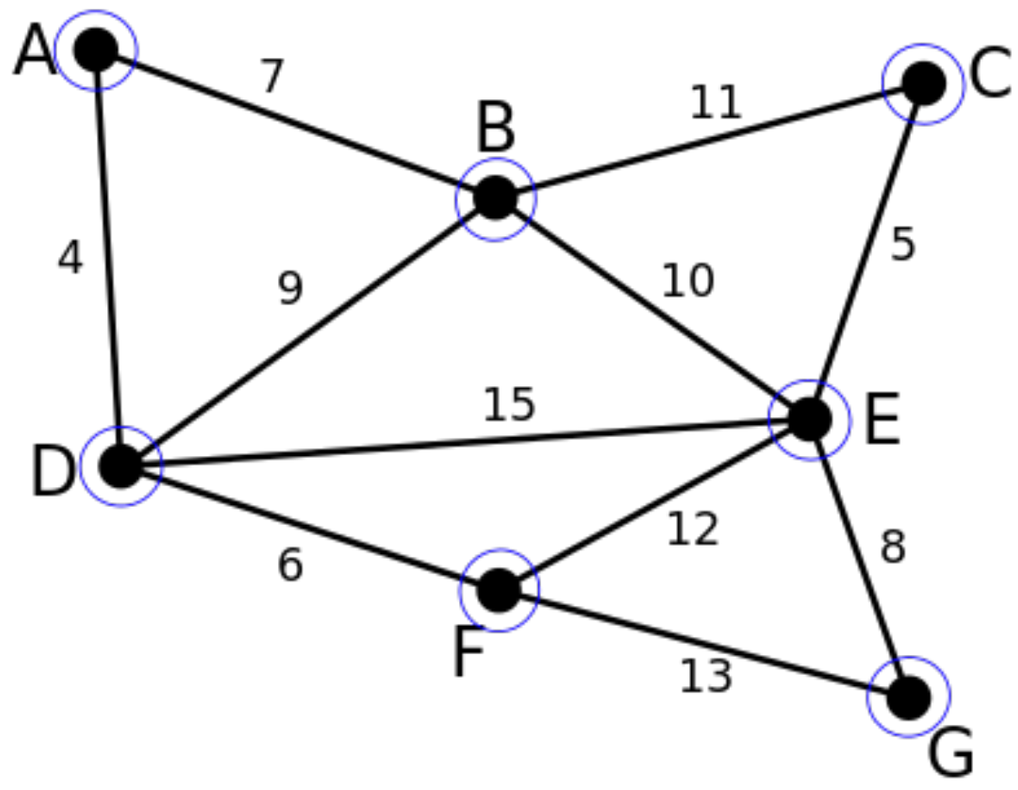
Kruskal's Algorithm

- Another MST algorithm
- Why do you think we're looking at a second one?

Kruskal's Algorithm

Idea: Add the cheapest remaining edge that does not create a cycle.

- Initialize $T = \emptyset$, $H \leftarrow E$
- While $|T| < n - 1$:
    - Remove cheapest edge e from H
    - If adding e to T does not create a cycle
    - $T \leftarrow T \cup \{e\}$



Kruskal's Algorithm

- Does it give us the correct MST?
  - Proof?
  - How quickly can we find the minimum remaining edge?
  - How quickly can we determine if an edge creates a cycle?

Kruskal's Implementation

- Sort edges by weight or extra edges stored in a min-heap (weight as priorities): $O(m \log m)$
    - Turns out this is the dominant cost
- Determine whether $T \cup \{e\}$ contains a cycle
    - Maintain a partition of V: components of T
    - Let $[u]$ denote component of u
    - Adding edge $e = (v, w)$ creates a cycle if and only if $[v] = [w]$
- Add an edge to T: update components

Union-Find Data Structure

Manages a **dynamic partition** of a set S

- Provides the following methods:
    - `MakeUnionFind()`: Initialize
    - `Find(x)`: Return name of set containing x
    - `Union(X, Y)`: Replace sets `X`, `Y` with $X \cup Y$

Kruskal's Algorithm can then use

- `Find` for cycle checking
- `Union` to update after adding an edge to T
- Sorting is still dominant step but imagine if edges were presented in sorted order!

Union-Find: First Attempt

Let $S = \{1, 2, \ldots, n\}$ be the set.

Idea: Each element stores the label of its partition

- `MakeUnionFind()`: Set $L[x] = x$ for each $x \in S$: $O(n)$
- `Find(x)`: Return $L[x]$: $O(1)$
- `Union(X,Y)`:
  - For each $z \in X \cup Y$, set $L[z]$ to the label of the **larger** set
    - $O(n)$
  - Doing this changes fewer names; how many times can an element change labels?
    - Each time an element changes labels, size of its set increases by at least 2: $O(\log n)$ times

Aside: Amortized Complexity

- Way to account for the average cost of a sequence of operations, where some operations may be expensive but happen infrequently
- Here we do at most n unions and total time spent on unions is $O(n \log n)$
    - What is the amortized complexity of a union operation?
    - $O(\log n)$

Definition. If k operations take total time $O(t \cdot k)$, then the amortized time per operation is $O(t)$.

Union-Find: First Attempt

Let $S = \{1, 2, \ldots, n\}$ be the set.

Idea: Each element stores the label of its set

- `MakeUnionFind()`: Set $L[x] = x$ for each $x \in S$: $O(n)$
- `Find(x)`: Return $L[x]$: $O(1)$
- `Union(X,Y)`:
  - For each $z \in X \cup Y$, set $L[z]$ to the label of the **larger** set
    - $O(n)$
  - Can happen at most $O(\log n)$ times
    - A sequence of n union operations take $O(n \log n)$ time
    - **Amortized cost of union** $O(\log n)$

Kruskal's with Union-Find

Note: If sorting is the bottleneck, the first attempt at union-find is good enough for Kruskals:

- Each time label of a vertex changes, its component size grows by at least 2: happens $O(\log_2 n)$ times
- Total time spent updating vertex labels: $O(n \log n)$
- Running time: $O(m \log m + n \log n) = O(m \log n)$; space $O(n + m)$

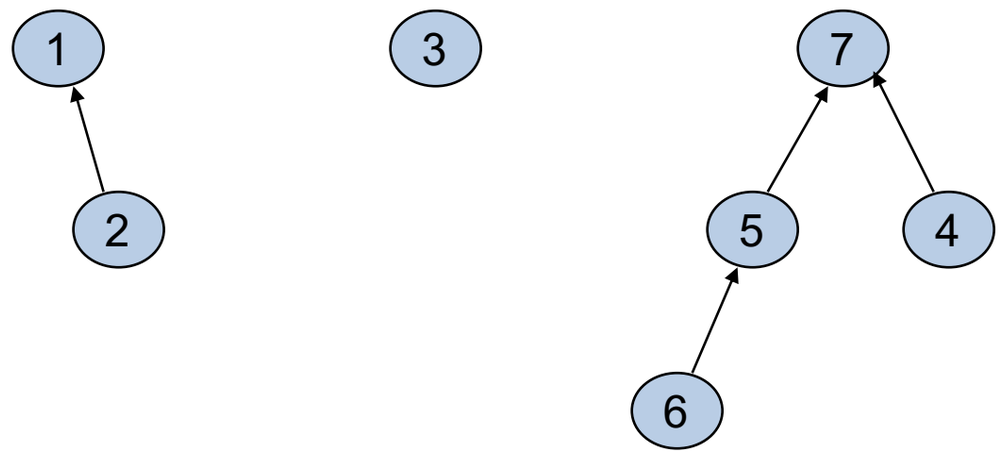
Not done with union-find. But as for union-find itself, we can still do a lot better…

Union-Find with Trees

- Trees let us find many elements given one root (i.e. representative); use one tree for each subset
- **Up tree.** If we reverse the pointers (child to parent), we can find a single root from many elements
- Union now will just require pointing one root to another

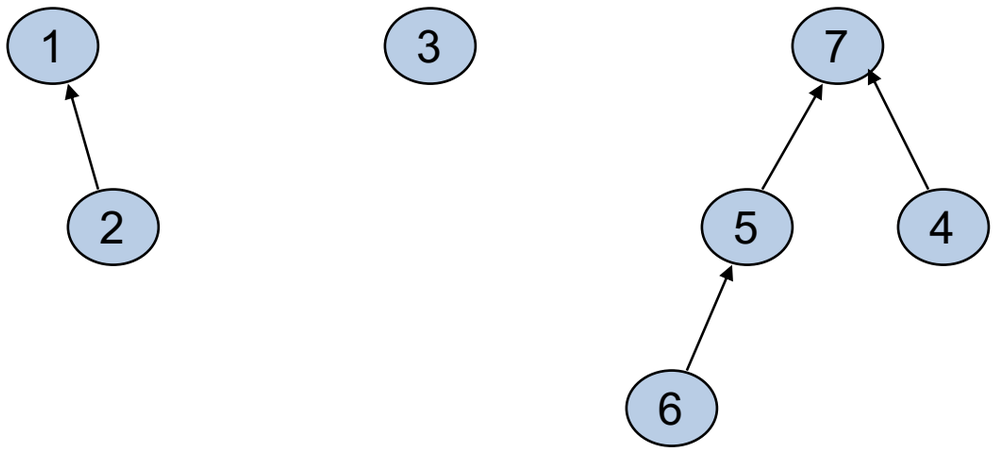
Initial state:

Intermediate state:

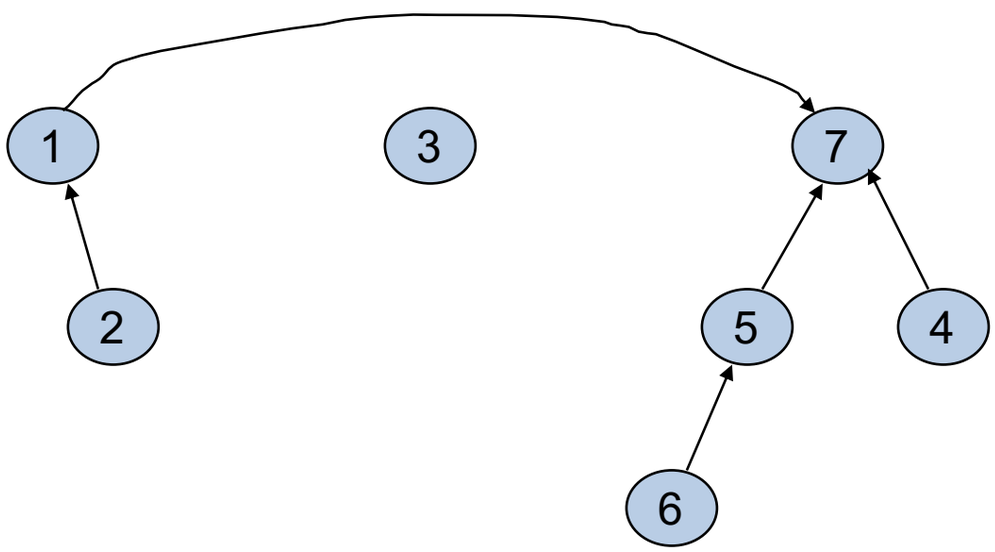


Union-Find with Trees

Intermediate state:

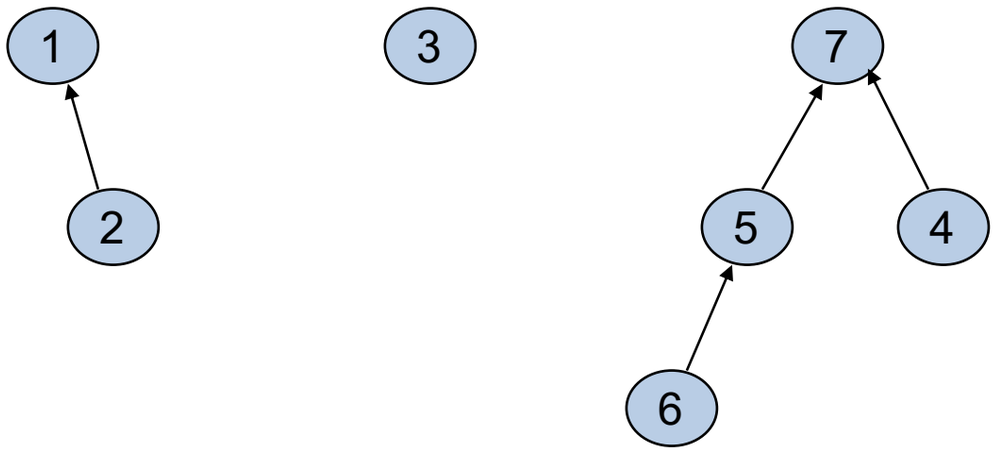


Union(1, 7)

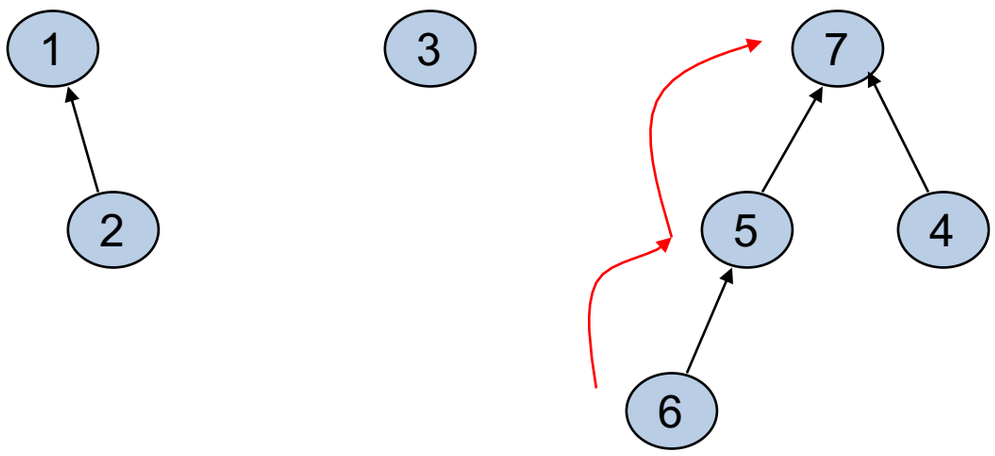


Union-Find with Trees

Intermediate state:

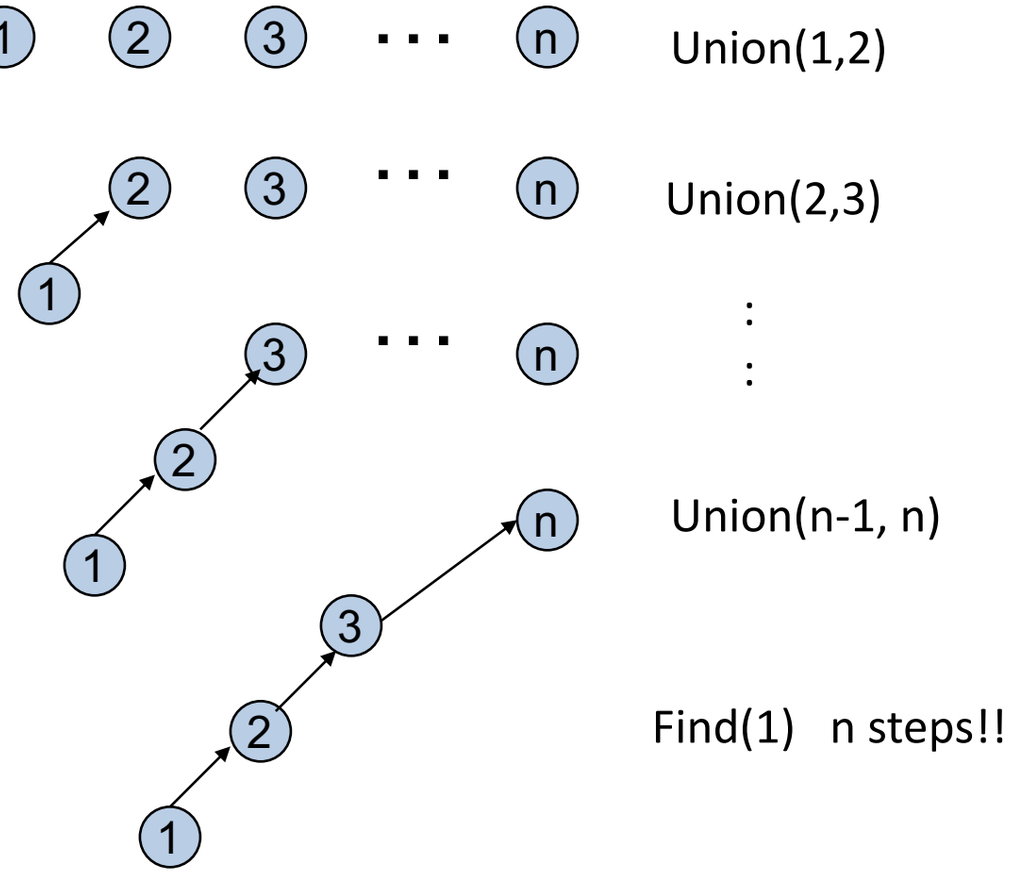


Find(6)



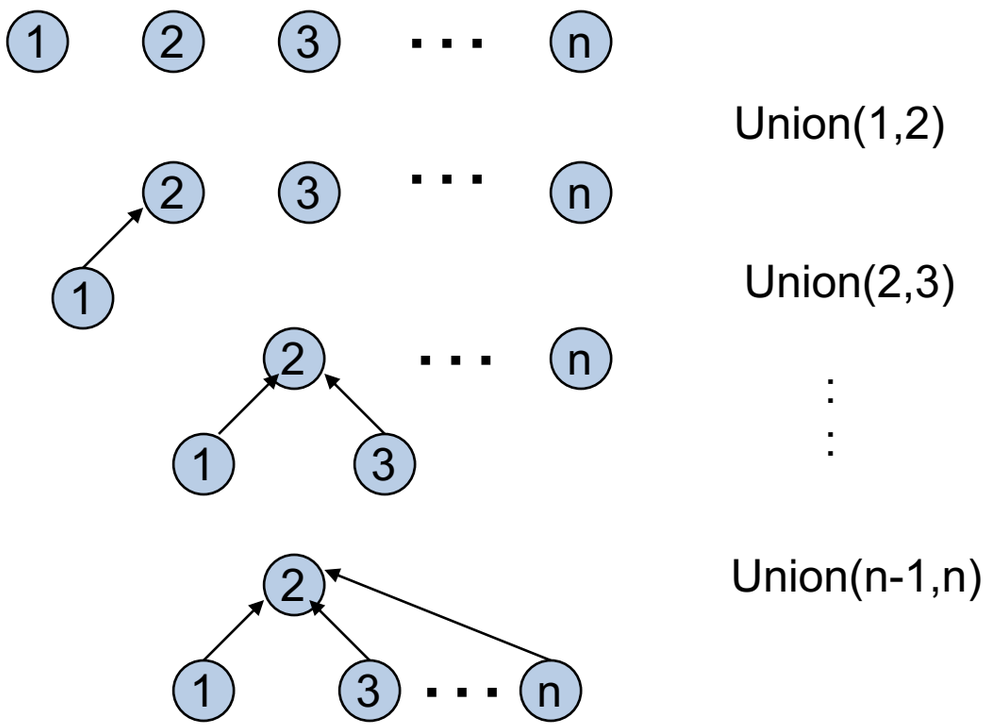
Union-Find with Trees

- Bad case!
- Find(1) takes $O(n)$ time



Union-Find with Trees

- Improvement (Union): point smaller to bigger tree



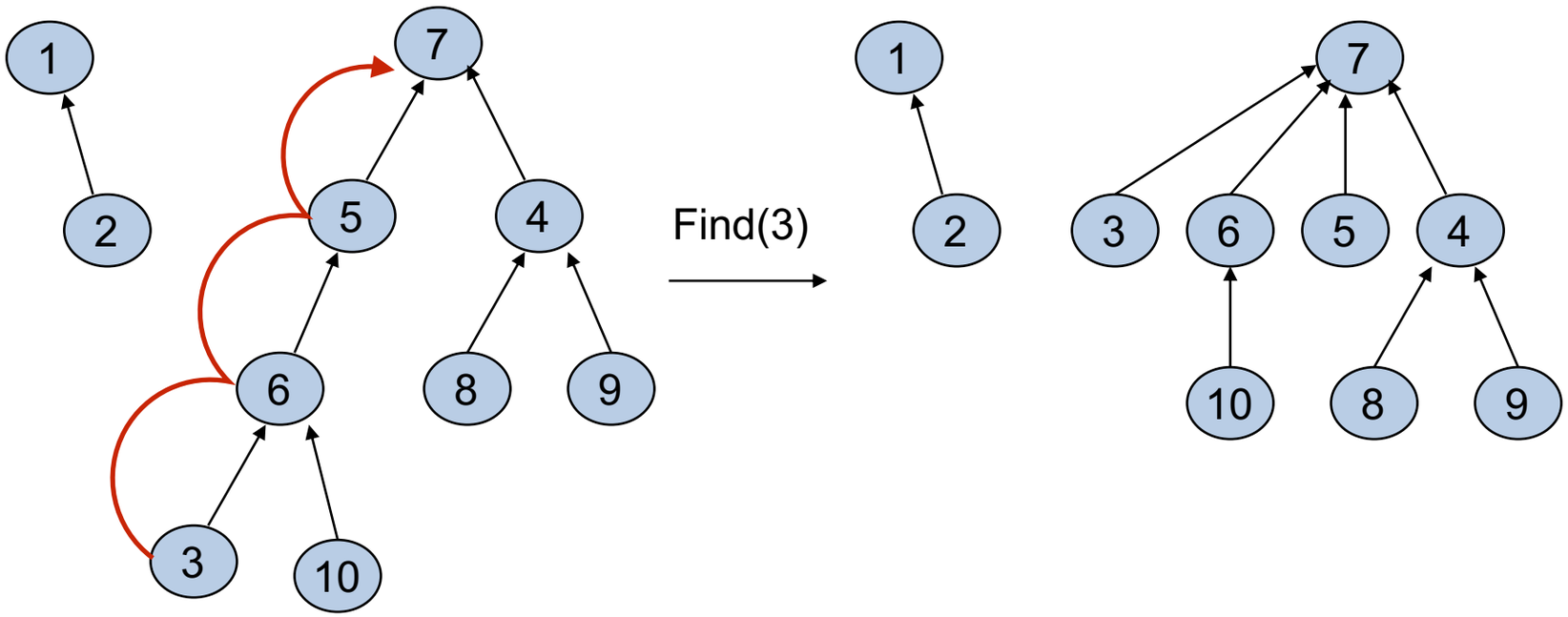
- Overall: Find $O(\log n)$ time (why?)
- Union: $O(1)$

Union-Find with Trees

- First attempt **(arrays)**
  - Union: amortized $O(\log n)$, worst-case $O(n)$
  - Find: $O(1)$
- Second attempt **(up trees)**
  - Union: $O(1)$
  - Find: $O(\log n)$
- **Question.** Do we always have to pay $\Omega(\log n)$ cost for one of union or find?
  - Surprisingly no, we can do better!
  - Possible to get cost of both essentially constant!

Union-Find: Path Compression

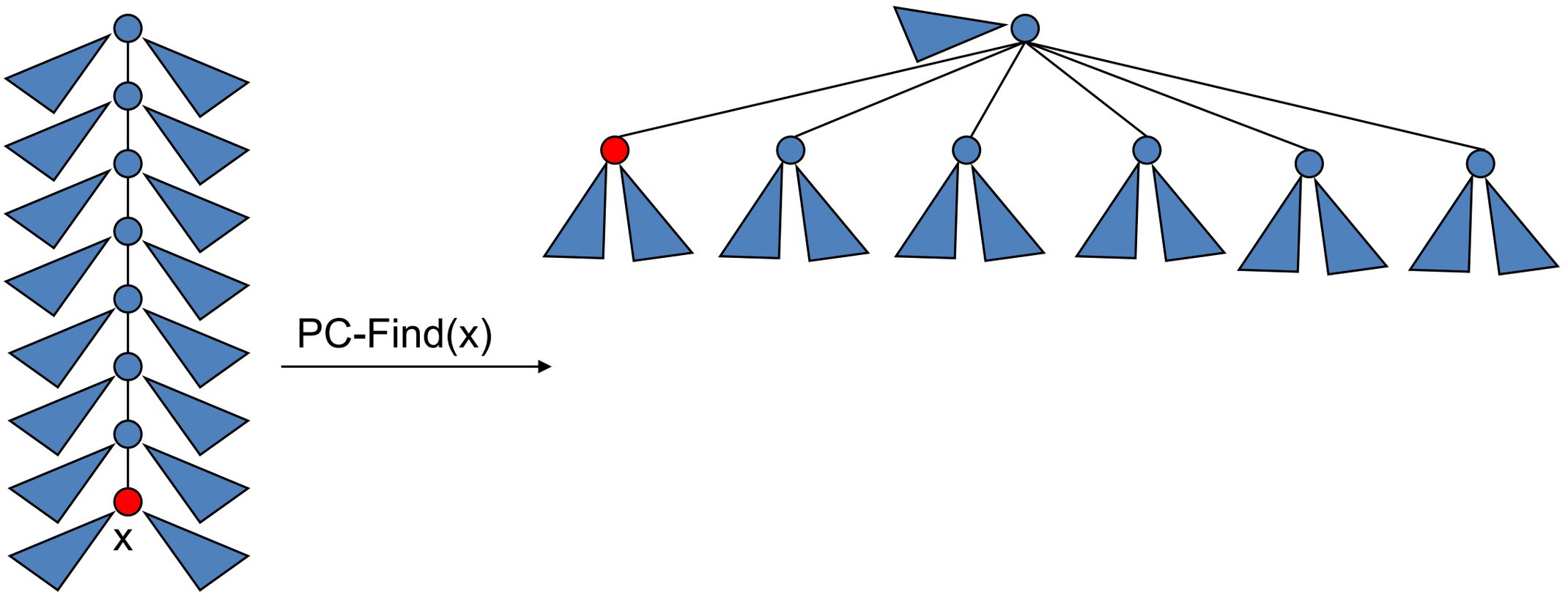
- **Heuristic.** On a Find operation point all the nodes on the search path directly to the root



- This does not change the worst case time complexity, which is still $O(\log n)$ and $O(1)$ for Union

Union-Find: Path Compression

- Self adjustment improves amortized complexity!



- Amortized complexity of find becomes very close to constant!

Surprising Result: Hopcroft Ulman'73

- Amortized complexity of union find with path compression improves significantly!
- Time complexity for n union and find operations on n elements is $O(n \log^* n)$
- $\log^* n$ is the number of times you need to apply the log function before you get to a number <= 1
- **Very small! Less than 5 for all reasonable values**

$$\log^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

n	1	2	$4 = 2^2$	$16 = 2^4$	$65,536 = 2^{16}$	$2^{65,536}$
$\log^*(n)$	0	1	2	3	4	5

Surprising Result: Tarjan ‘75

- Improved bound on amortized complexity of union-find with path compression
- Time complexity for n union and find operations on n elements is $O(n\alpha(n))$, where
    - $\alpha(n)$ is extremely slow-growing, **inverse-Ackermann function**
        - Essentially a constant
- Grows much **muuchch morrree** slowly than $\log^*$
- $\alpha(n) \leq 4$ for all values in practice
- **Result.** Union and Find become (essentially) amortized constant time in practice (just short of $O(1)$ in theory) !

Ackermann & Inverse Ackermann

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

“I am not smart enough to understand this” — Seidel

Inverse Ackermann

- **Inverse Ackerman:** The function $\alpha(n)$ grows much more slowly than $\log^{*c} n$ for **any fixed c**

- With $\log^*$, you count how many times does applying $\log$ over and over gets the result to become small
- With the inverse Ackermann, essentially you count how many times does applying $\log^*$ (not log!) over and over gets the result to become small

- $\alpha(n) = \min\{k \mid \log^{\overbrace{****\cdots*}^{k}}(n) \leq 2\}$

- $\alpha(n) = 4$ for $n = 2^{2^{2^{2^{16}}}}$

Many Applications of Union-Find

- Good for applications in need of clustering
    - cities connected by roads
    - cities belonging to the same country
    - connected components of a graph
- Maintaining equivalence classes
- Maze creation!

MST Algorithms History

- **Borůvka's Algorithm** (1926)
    - The Borvka / Choquet / Florek-ukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh algorithm
    - Oldest, most-ignored MST algorithm, but actually very good
- **Jarník's Algorithm** ("Prims Algorithm", 1929)
    - Published by Jarník, independently discovered by Kruskal in 1956, by Prims in 1957
- **Kruskal's Algorithm** (1956)
    - Kruskal designed this because he found Borůvka's algorithm "unnecessarily complicated"

Can we do better?

Best known algorithm by Chazelle (1999)

A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity*

BERNARD CHAZELLE†

NECI Research Tech Report 99-099 (July 1999)
Journal of the ACM, 47(6), 2000, pp. 1028–1047.

Abstract

A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.

1 Introduction

The history of the minimum spanning tree (MST) problem is long and rich, going as far back as Borůvka's work in 1926 [1, 9, 13]. In fact, MST is perhaps the oldest open problem in computer science. According to Nešetřil [13], "this is a cornerstone problem of combinatorial optimization and in a sense its cradle." Textbook algorithms run in $O(m \log n)$ time, where n

Can we do better?

Using randomness, can get $O(n + m)$ time!

A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees

DAVID R. KARGER

Stanford University, Stanford, California

PHILIP N. KLEIN

Brown University, Providence, Rhode Island

AND

ROBERT E. TARJAN

Princeton University and NEC Research Institute, Princeton, New Jersey

Abstract. We present a randomized linear-time algorithm to find a minimum spanning tree in a connected graph with edge weights. The algorithm uses random sampling in combination with a recently discovered linear-time algorithm for verifying a minimum spanning tree. Our computational model is a unit-cost random-access machine with the restriction that the only operations allowed on edge weights are binary comparisons.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete**

Optimal MST Algorithm?

Has been discovered but don't know its running time!

An Optimal Minimum Spanning Tree Algorithm

SETH PETTIE AND VIJAYA RAMACHANDRAN

The University of Texas at Austin, Austin, Texas

Abstract. We establish that the algorithmic complexity of the minimum spanning tree problem is equal to its decision-tree complexity. Specifically, we present a deterministic algorithm to find a minimum spanning tree of a graph with n vertices and m edges that runs in time $O(\mathcal{T}^*(m, n))$ where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons needed to determine the solution. The algorithm is quite simple and can be implemented on a pointer machine.

Although our time bound is optimal, the exact function describing it is not known at present. The current best bounds known for $\mathcal{T}^*$ are $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where α is a certain natural inverse of Ackermann's function.

Even under the assumption that $\mathcal{T}^*$ is superlinear, we show that if the input graph is selected from $G_{n,m}$, our algorithm runs in linear time with high probability, regardless of n, m, or the permutation of edge weights. The analysis uses a new martingale for $G_{n,m}$ similar to the edge-exposure martingale for G