COMP 382: Reasoning about Algorithms

Greedy Algorithms: Minimum Spanning Trees

Prof. Maryam Aliakbarpour

co-instructors: Prof. Anjum Chida & Prof. Konstantinos Mamouras

October 23, 2025

Today's Lecture

1. **Minimum Spanning Trees**

2. **Prim's Algorithm**

3. **Kruskal's Algorithm**

Reading:

- Chapter 15 of [Roughgarden, 2022]

Adapted from the same chapters.

Minimum Spanning Trees

The Core Problem: Cheap Connections

Imagine you need to connect a set of locations—like computer servers, cities, or houses—as cheaply as possible.

The Goal:

- Connect all locations into a single network.
- Do so with the minimum possible total cost (e.g., cable length, pipe cost, road miles).
- Don't create any redundant loops or cycles.

This problem appears everywhere, from designing computer networks to machine learning.

Formalizing the Problem

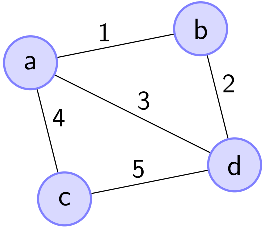
To solve this, we model the problem using a graph.

An **undirected graph** $G = (V, E)$ has:

- A set of **vertices** V (the locations).
- A set of **edges** E (the potential connections).
- Each edge e has a **cost** c_e.

A **Spanning Tree** is a subset of edges that:

1. Connects all vertices ("spanning").
2. Contains no cycles ("tree").



A Weighted Graph

Formalizing the Problem

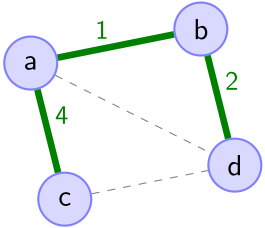
To solve this, we model the problem using a graph.

An **undirected graph** $G = (V, E)$ has:

- A set of **vertices** V (the locations).
- A set of **edges** E (the potential connections).
- Each edge e has a **cost** c_e.

A **Spanning Tree** is a subset of edges that:

1. Connects all vertices ("spanning").
2. Contains no cycles ("tree").



A Spanning Tree

Prim's Algorithm

A Greedy Algorithm for MST

Prim's Algorithm: The Mold Grower

Our first method, Prim's algorithm, builds the MST by growing a single tree, one edge at a time.

Prim's Greedy Strategy

Start at an arbitrary vertex. In each step, greedily add the **cheapest edge** that connects a vertex *inside* our growing tree to a vertex *outside* the tree.

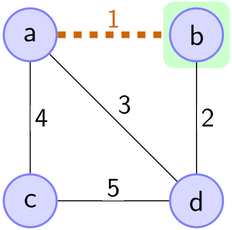
Think of it like a mold that starts at one point and expands along the cheapest paths until it covers everything.

Prim's Algorithm in Action

Let's run Prim's starting from vertex **b**. The green area shows the vertices spanned so far.

Start: At vertex b

- Candidates: (b,a) [cost 1], (b,d) [cost 2].
- Add cheapest: **(b,a)**.



Total Cost: 0

Prim's Algorithm in Action

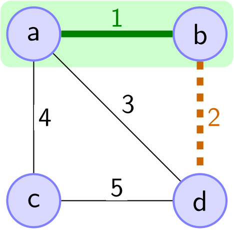
Let's run Prim's starting from vertex **b**. The green area shows the vertices spanned so far.

Start: At vertex b

- Candidates: (b,a) [cost 1], (b,d) [cost 2].
- Add cheapest: **(b,a)**.

Step 1: Add (b,a)

- Candidates: (a,c) [4], (a,d) [3], (b,d) [2].
- Add cheapest: **(b,d)**.



Total Cost: 1

Prim's Algorithm in Action

Let's run Prim's starting from vertex **b**. The green area shows the vertices spanned so far.

Start: At vertex b

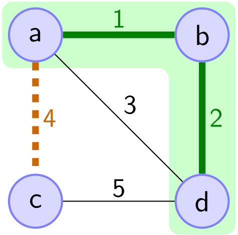
- Candidates: (b,a) [cost 1], (b,d) [cost 2].
- Add cheapest: **(b,a)**.

Step 1: Add (b,a)

- Candidates: (a,c) [4], (a,d) [3], (b,d) [2].
- Add cheapest: **(b,d)**.

Step 2: Add (b,d)

- Ignore (a,d) → creates cycle.
- Candidates: (a,c) [4], (c,d) [5].
- Add cheapest: **(a,c)**.



Total Cost: 1 + 2

Prim's Algorithm in Action

Let's run Prim's starting from vertex **b**. The green area shows the vertices spanned so far.

Start: At vertex b

- Candidates: (b,a) [cost 1], (b,d) [cost 2].
- Add cheapest: **(b,a)**.

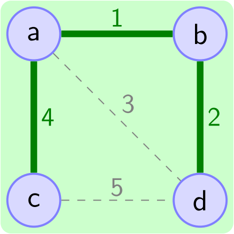
Step 1: Add (b,a)

- Candidates: (a,c) [4], (a,d) [3], (b,d) [2].
- Add cheapest: **(b,d)**.

Step 2: Add (b,d)

- Ignore (a,d) → creates cycle.
- Candidates: (a,c) [4], (c,d) [5].
- Add cheapest: **(a,c)**.

Step 3: Add (a,c)



Total Cost: $1 + 2 + 4 = 7$

Prim's Algorithm: Pseudocode

This is the simple, high-level idea.

Prim's Algorithm (G, s)

- Initialize $X = \{s\}$ (our set of spanned vertices)
- Initialize $T = \emptyset$ (our set of MST edges)
- **while** $X \neq V$:
  - Let $e = (u, v)$ be the **cheapest** edge with:
  - $u \in X$
  - $v \notin X$
- Add e to T
- Add v to X
- **return** T

Question: How do we know this greedy strategy actually works?

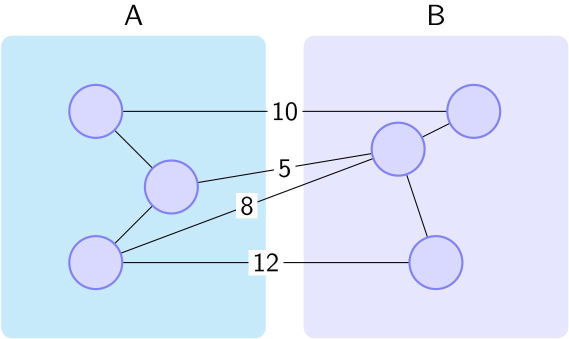
Correctness: The Cut Property

Why is this "Greedy" Choice Safe?

The answer is a beautiful idea called the **Cut Property**.

What is a "Cut"?

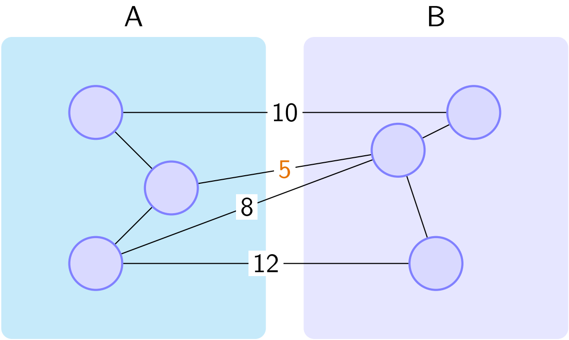
- A "cut" is just a partition of the vertices V into two non-empty sets, A and B.
- "Crossing edges" are edges with one endpoint in A and one in B.



The Cut Property

The Cut Property

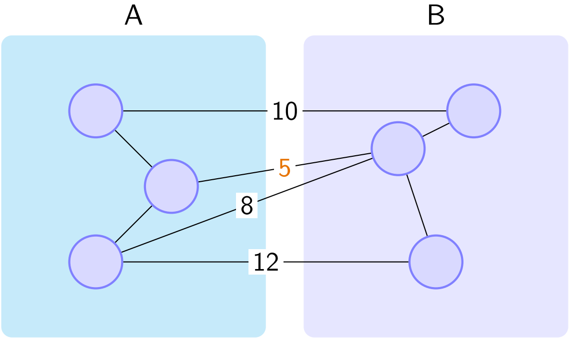
Assume all edge costs are distinct.
Let e be the **cheapest edge** crossing *any* cut (A, B).
Then e **must** belong to the Minimum Spanning Tree.



The Cut Property

Why is this true? If an MST **didn't** use *e*, it would have to use some other, more expensive edge *f* to cross that cut. We could swap *f* for *e* and get a **cheaper** tree!

This is a contradiction.

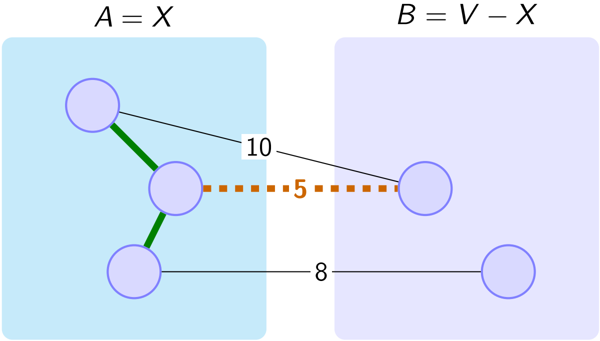


Prim's Algorithm IS The Cut Property

Prim's algorithm cleverly uses the Cut Property in every single step!

At each step, Prim's defines a cut:

- $A = X$ (vertices already in our tree)
- $B = V - X$ (vertices not yet in)

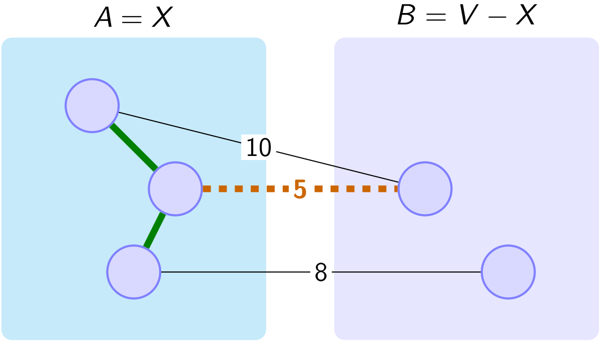


Prim's Algorithm IS The Cut Property

The algorithm then finds the **cheapest edge** crossing this *specific* cut...

...and adds it to the tree!

The Cut Property guarantees this is a "safe" and correct move.



Making Prim's Algorithm Fast

Via Priority Queue

How Fast is Prim's Algorithm?

Let $n = |V|$ (vertices) and $m = |E|$ (edges).

A "Straightforward" Implementation:

- The main loop runs $n - 1$ times (once for each vertex).
- In each loop, we have to search *all* m edges to find the cheapest one crossing the cut.

$$\text{Total Time: } O(n \times m) = O(mn)$$

How Fast is Prim's Algorithm?

Let $n = |V|$ (vertices) and $m = |E|$ (edges).

A "Straightforward" Implementation:

- The main loop runs $n - 1$ times (once for each vertex).
- In each loop, we have to search *all* m edges to find the cheapest one crossing the cut.

$$\text{Total Time: } O(n \times m) = O(mn)$$

We can do much better!

Prim's Algorithm: Running Time

This is the simple, high-level idea.

Prim's Algorithm (G, s)

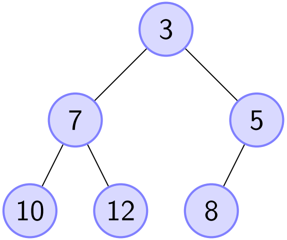
- Initialize $X = \{s\}$ (our set of spanned vertices)
- Initialize $T = \emptyset$ (our set of MST edges)
- **while** $X \neq V$: — O(n) times (once per vertex)
  - Let $e = (u, v)$ be the **cheapest** edge with: — O(m) search overall edges.
  - $u \in X$
  - $v \notin X$
  - Add e to T
  - Add v to X
- **return** T

Tool for the Job: The Heap (Priority Queue)

To find the cheapest crossing edge faster, we need a special tool.

What is a Heap?

- A data structure that maintains an evolving set of objects, each with a "key" or "cost".
- Its main job is to perform **minimum** computations very, very quickly.
- Think of it as a "queue" list where the task with the **smallest cost** is always at the top, ready to be pulled.



A **Min-Heap**

Tool for the Job: The Heap (Priority Queue)

Key Operations (for n items)

Operation	What it does	Time
`INSERT`	Adds a new object to the set.	$O(\log n)$
`EXTRACT-MIN`	Removes and returns the object with the *smallest* key.	$O(\log n)$
`DELETE`	Removes a specific object from the set.	$O(\log n)$

Tool for the Job: The Heap (Priority Queue)

Key Operations (for n items)

Operation	What it does	Time
`INSERT`	Adds a new object to the set.	$O(\log n)$
`EXTRACT-MIN`	Removes and returns the object with the *smallest* key.	$O(\log n)$
`DELETE`	Removes a specific object from the set.	$O(\log n)$

This is perfect for Prim's!

- `EXTRACT-MIN` gives us the next vertex to add to X.
- `DELETE` + `INSERT` lets us update the key of a vertex when a cheaper edge is found.

Speeding Up Prim's with a Heap

The bottleneck is re-scanning all edges just to find the cheapest one.

The Key Idea: Use a **heap** (Priority Queue) to keep track of the "cheapest crossing edge" for each vertex *outside* our tree.

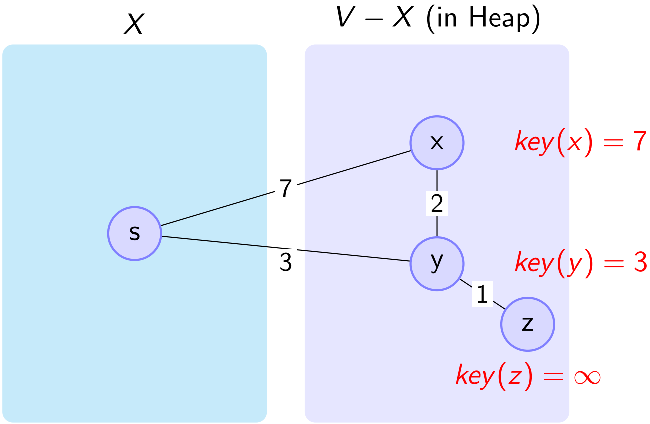
Heap Invariant

- The heap stores all vertices in $V - X$ (those not in the tree).
- The "key" for a vertex $v \in V - X$ is the cost of the **cheapest edge** connecting v to any vertex *inside* X.

Now, each step of Prim's is just an **Extract-Min** from the heap!

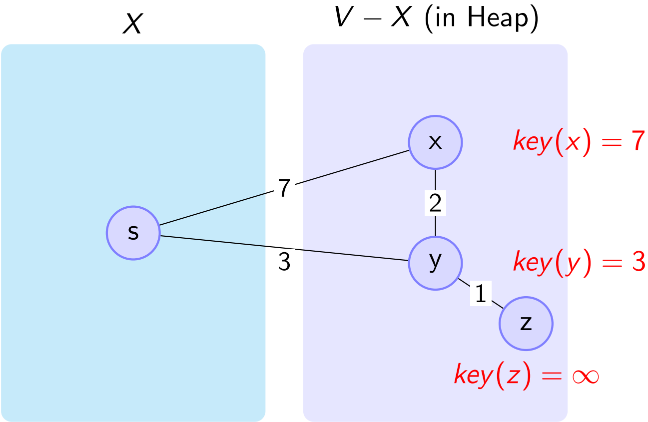
Prim's with a Heap

- **Heap contains:** $\{y, x, z\}$
- **Keys:**
  - $key(y) = 3$
  - $key(x) = 7$
  - $key(z) = \infty$ (no edge to X)



Prim's with a Heap

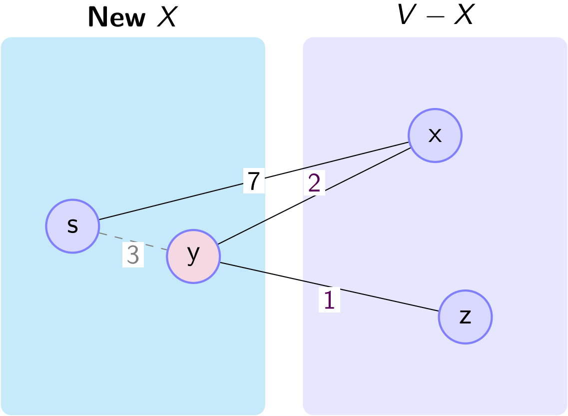
- **Heap contains:** $\{y, x, z\}$
- **Keys:**
  - $key(y) = 3$
  - $key(x) = 7$
  - $key(z) = \infty$ (no edge to X)
- **Step 1:** 'Extract-Min()'
- **Returns:** vertex **y** (cost 3).
- **Action:** Add y to X.



The "Catch": Updating Keys

When we add a vertex (like y) to X, we must update the keys of its neighbors!

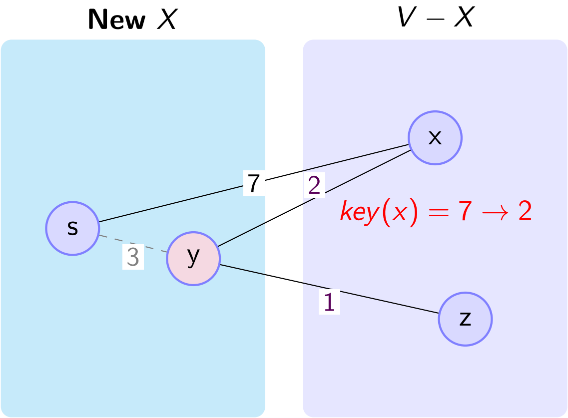
- y is now in X.
- Look at y's neighbors in $V - X$:



The "Catch": Updating Keys

When we add a vertex (like y) to X, we must update the keys of its neighbors!

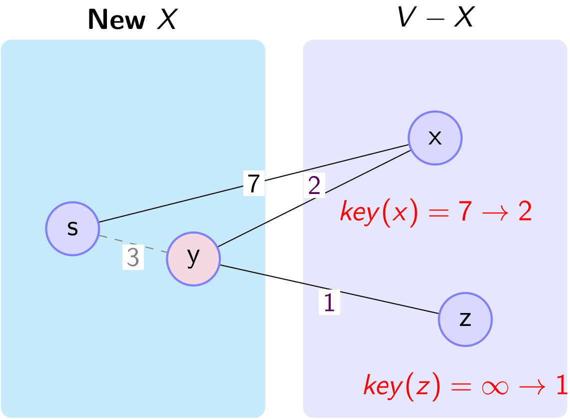
- y is now in X.
- Look at y's neighbors in $V - X$:
- **Neighbor x:**
  - Old key: 7 (from s)
  - New edge (y, x): cost 2
  - Update key(x) to 2.



The "Catch": Updating Keys

When we add a vertex (like y) to X, we must update the keys of its neighbors!

- y is now in X.
- Look at y's neighbors in $V - X$:
- **Neighbor x:**
  - Old key: 7 (from s)
  - New edge (y, x): cost 2
  - Update key(x) to 2.
- **Neighbor z:**
  - Old key: ∞
  - New edge (y, z): cost 1
  - Update key(z) to 1.



This is a **Decrease-Key** operation in the heap.

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n - 1$ iterations):

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n - 1$ iterations):
- Extract-Min: $n - 1$ times. Total: $O(n \log n)$

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n - 1$ iterations):
- Extract-Min: $n - 1$ times. Total: $O(n \log n)$
- Decrease-Key (Updates): Total: $O(m \log n)$

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n-1$ iterations):
- Extract-Min: $n-1$ times. Total: $O(n \log n)$
- Decrease-Key (Updates): Total: $O(m \log n)$
  - This is the tricky part. We check each edge (v, w) *once*, when v is first added to X.

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n - 1$ iterations):
- Extract-Min: $n - 1$ times. Total: $O(n \log n)$
- Decrease-Key (Updates): Total: $O(m \log n)$
  - This is the tricky part. We check each edge (v, w) *once*, when v is first added to X.
  - If we have an adjacency list, we can do this in $O(d_v)$ time (where d_v is the degree of v).

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n - 1$ iterations):
- Extract-Min: $n - 1$ times. Total: $O(n \log n)$
- Decrease-Key (Updates): Total: $O(m \log n)$
  - This is the tricky part. We check each edge (v, w) *once*, when v is first added to X.
  - If we have an adjacency list, we can do this in $O(d_v)$ time (where d_v is the degree of v).
  - The total time is $O(\sum_v d_v) = O(m)$.

Heap-Based Running Time

Let's count the total work.

- Initialization: Build the heap $O(n \log n)$
- Main Loop (total over $n-1$ iterations):
- Extract-Min: $n-1$ times. Total: $O(n \log n)$
- Decrease-Key (Updates): Total: $O(m \log n)$
  - This is the tricky part. We check each edge (v, w) *once*, when v is first added to X.
  - If we have an adjacency list, we can do this in $O(d_v)$ time (where d_v is the degree of v).
  - The total time is $O(\sum_v d_v) = O(m)$.

$$\text{Grand Total: } O(n \log n + m \log n) = O(m \log n)$$

(Assuming $m \geq n-1$, which is true for connected graphs)

Kruskal's Algorithm

Another Greedy Algorithm for MST

Kruskal's Algorithm: The Forest Loner

A completely different (but equally brilliant) greedy strategy.

Kruskal's Greedy Strategy

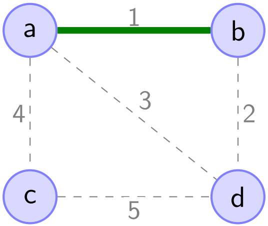
1. **Sort** all m edges in the graph from cheapest to most expensive.
2. **Iterate** through the sorted edges:
3. Add an edge to your tree T **if and only if** it does **not** create a cycle.

Instead of growing one "mold," Kruskal's builds up a "forest" of small trees that eventually merge into one.

Kruskal's Algorithm in Action

Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]

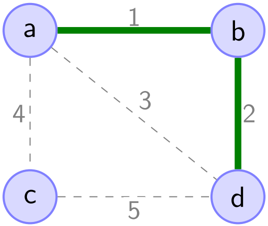
1. Edge (a,b) [cost 1]:
   - No cycle. Add.



Kruskal's Algorithm in Action

Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]

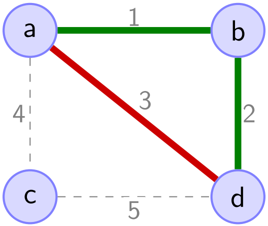
1. Edge (a,b) [cost 1]:
   - No cycle. Add.
2. Edge (b,d) [cost 2]:
   - No cycle. Add.



Kruskal's Algorithm in Action

Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]

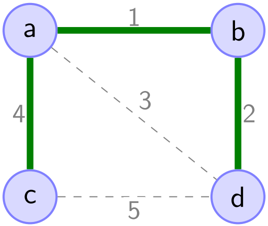
1. Edge (a,b) [cost 1]:
   - No cycle. Add.
2. Edge (b,d) [cost 2]:
   - No cycle. Add.
3. Edge (a,d) [cost 3]:
   - Creates a cycle (a-b-d-a). Skip!



Kruskal's Algorithm in Action

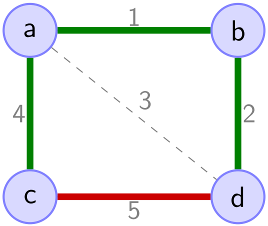
Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]

1. Edge (a,b) [cost 1]:
   - No cycle. Add.
2. Edge (b,d) [cost 2]:
   - No cycle. Add.
3. Edge (a,d) [cost 3]:
   - Creates a cycle (a-b-d-a). Skip!
4. Edge (a,c) [cost 4]:
   - No cycle. Add.



Kruskal's Algorithm in Action

Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]



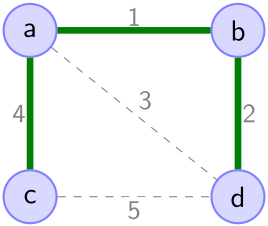
1. Edge (a,b) [cost 1]:
   - No cycle. Add.
2. Edge (b,d) [cost 2]:
   - No cycle. Add.
3. Edge (a,d) [cost 3]:
   - Creates a cycle (a-b-d-a). Skip!
4. Edge (a,c) [cost 4]:
   - No cycle. Add.
5. Edge (c,d) [cost 5]:
   - Creates a cycle. Skip!

Kruskal's Algorithm in Action

Sorted Edges: (a,b) [1], (b,d) [2], (a,d) [3], (a,c) [4], (c,d) [5]

1. Edge (a,b) [cost 1]:
   - No cycle. Add.
2. Edge (b,d) [cost 2]:
   - No cycle. Add.
3. Edge (a,d) [cost 3]:
   - Creates a cycle (a-b-d-a). Skip!
4. Edge (a,c) [cost 4]:
   - No cycle. Add.
5. Edge (c,d) [cost 5]:
   - Creates a cycle. Skip!

Done! We have $n - 1 = 3$ edges.



Final Cost: $1 + 2 + 4 = 7$

Kruskal's Algorithm: Pseudocode (high level)

Kruskal's Algorithm (G, s)

- $T = \emptyset$ (our set of MST edges)
- Sort all m edges in E by increasing cost.
- **for** each edge $e = (u, v)$ in the sorted list:
  - **if** $T \cup \{e\}$ has no cycles:
  - Add e to T
- **return** T

Correctness: The Cut Property (Again!)

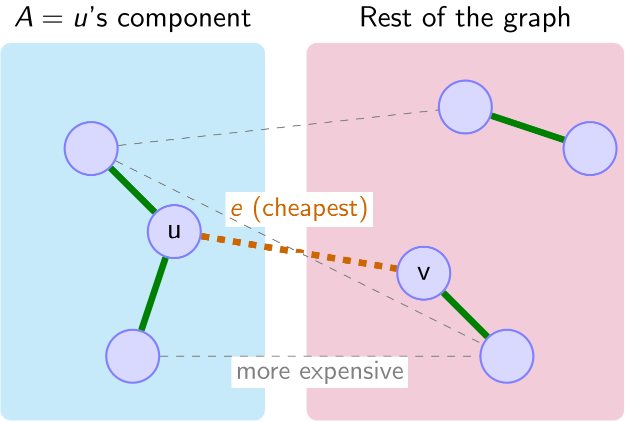
Why Does Kruskal's Work?

It also relies on the Cut Property, but in a sneakier way.

Proof Overview:

- Consider the moment Kruskal's adds edge $e = (u, v)$.
- At this point, u and v are in *different* components (or e would form a cycle).
- Let $A = u$'s component, $B = V - A$. This is a cut!
- Since edges are sorted, e *must* be the cheapest edge crossing this cut. (Any cheaper crossing edge would have been considered earlier).
- Adding e is a "safe" move by the Cut Property!

Why Does Kruskal's Work?



Kruskal's Running Time

How Fast is Kruskal's?

The algorithm has two main parts:

1. **Sorting the Edges**
   - We have m edges.
   - Using MergeSort: $\mathbf{O}(\mathbf{m} \log \mathbf{n})$.

2. **Checking for Cycles**
   - We loop m times.
   - Inside the loop: 'if ($T \cup e$ has no cycle)'... How?

How Fast is Kruskal's?

The algorithm has two main parts:

1. **Sorting the Edges**
   - We have m edges.
   - Using MergeSort: $\mathbf{O(m \log n)}$.

2. **Checking for Cycles**
   - We loop m times.
   - Inside the loop: 'if ($T \cup e$ has no cycle)'... How?

The "Straightforward" Way:

- A simple BFS/DFS check for a path between u and v takes $O(n)$ time.
- Total "straightforward" time: $O(m \log n) + O(m \times n) = \mathbf{O(mn)}$.
- This is no better than simple Prim's! We **must** make the cycle check faster.

Making Kruskal's Algorithm Fast

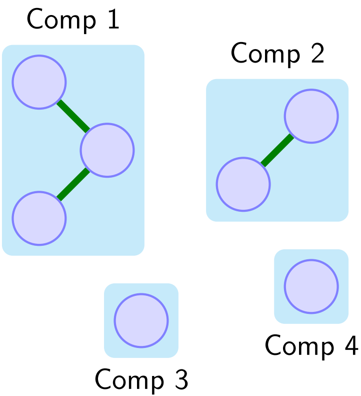
The Union-Find Data Structure

Speeding Up Kruskal's: The Union-Find Data Structure

This tool is designed specifically for tracking connected components.

The Core Idea

- Maintain the connected components formed by the edges added to T so far.
- "Objects" $=$ Vertices V.
- "Groups" $=$ Connected Components.



Speeding Up Kruskal's: The Union-Find Data Structure

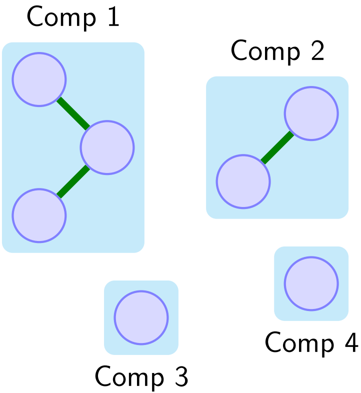
This tool is designed specifically for tracking connected components.

The Core Idea

- Maintain the connected components formed by the edges added to T so far.
- "Objects" $=$ Vertices V.
- "Groups" $=$ Connected Components.

Key Operations

- `FIND(u)`: Get name/leader of u's component.
- `UNION(u, v)`: Merge u's and v's components.



Kruskal's Algorithm: Fast Pseudocode

Using Union-Find makes cycle checking incredibly efficient.

Kruskal's Algorithm (Fast Implementation)

- $T = \emptyset$
- Sort all m edges in E by increasing cost.
- Initialize a Union-Find structure U (each vertex in its own set).
- **for** each edge $e = (u, v)$ in the sorted list:
  - *Cycle Check:* **if** $\texttt{FIND}(U, u) \neq \texttt{FIND}(U, v)$:
  - Add e to T
  - $\texttt{UNION}(U, u, v)$ // Merge components
- **return** T

Making Kruskal’s Algorithm Fast

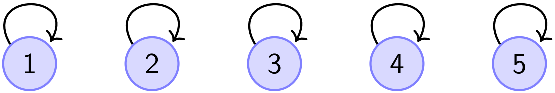
The Union-Find Data Structure

Union-Find: Initialization

Internally, Union-Find uses trees with parent pointers.

Initialization Step

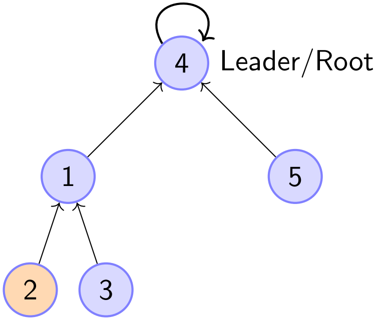
- Each vertex begins as an isolated component and its own root/leader.
- Each vertex points to itself to represent this.
- Setup time: $O(n)$ for n vertices.



Union-Find: FIND Operation

FIND(v) Operation: Finds the group leader

- Start at vertex v.
- Follow parent pointers upward until root
  - root = a vertex points to itself.
- Return that vertex (the component's leader).

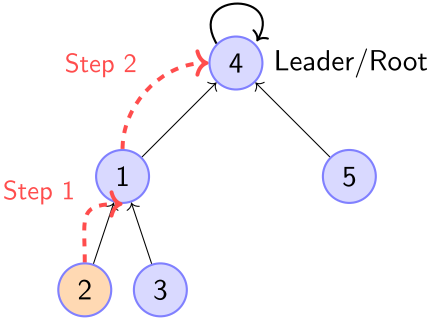


Union-Find: FIND Operation

FIND(v) Operation: Finds the group leader

- Start at vertex v.
- Follow parent pointers upward until root
  - root = a vertex points to itself.
- Return that vertex (the component's leader).

`FIND(2)` follows pointers:
$2 \to 1 \to 4$. Returns 4.



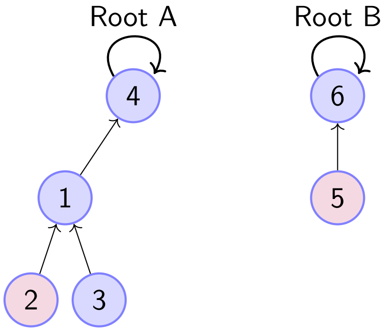
Union-Find: Simple UNION Operation

How do we merge two components (trees) A and B?

Simple UNION(A, B) Idea

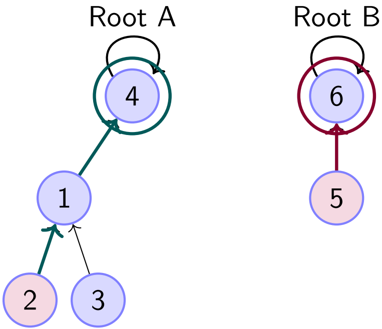
- Find the root of A (let's call it rootA).
- Find the root of B (let's call it rootB).
- Make one root point to the other (e.g., make rootA point to rootB).

Union-Find: Simple UNION Operation



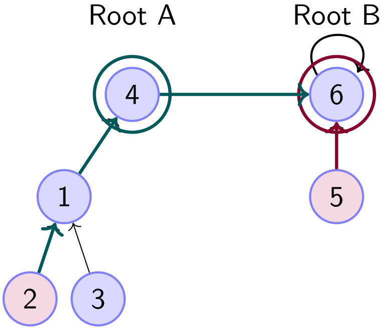
Perform `UNION(2, 5)`.

Union-Find: Simple UNION Operation



`find(2)` returns 4; `find(5)` returns 6.

Union-Find: Simple UNION Operation



Link roots 4 $\rightarrow$ 6; remove 4's self-loop (4 is no longer a leader).

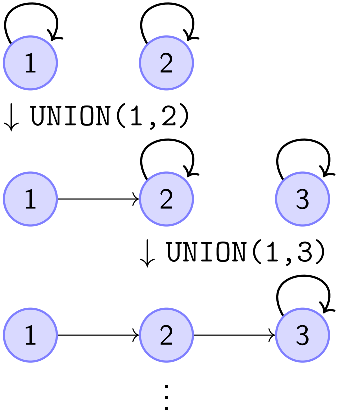
The Problem with Simple UNION

Issue: Arbitrary unions can create inefficient trees.

Worst Case:

- Repeated merges form a long chain.
- Tree height grows to $O(n)$.

Finding the root could take $O(n)$ steps.
slow!



Making Union-Find Fast: Union-by-Size

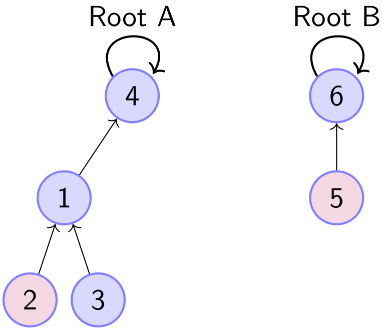
We can avoid creating tall trees with a simple rule.

The Trick: Union-by-Size (or Rank)

When doing `UNION(A, B)`, always attach the root of the **smaller** tree under the root of the **larger** tree. (Break ties arbitrarily).

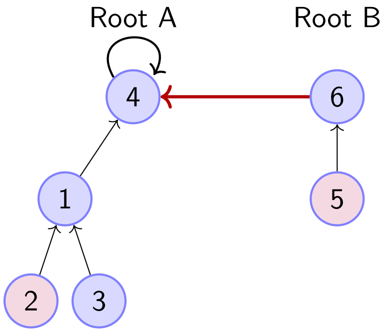
- Requires storing the size (number of nodes) at the root of each tree.
- Update size when merging.

Union-Find: UNION-by-Size



Perform `UNION(2, 5)`.

Union-Find: UNION-by-Size



Link roots 4←6; remove 4's self-loop (4 is no longer a leader).

Why is Union-by-Size Fast?

This simple heuristic dramatically improves performance!

Key Insight:

- Consider any vertex v.
- When does the depth of v (distance to root) increase?
- Only when v's tree is attached under *another* root during a UNION.
- By Union-by-Size, this happens only if the *other* tree was $\geq$ the size of v's current tree.
- $\implies$ Every time v's depth increases, the size of its *new* component **at least doubles**.

Why is Union-by-Size Fast?

This simple heuristic dramatically improves performance!

Key Insight:

- Consider any vertex v.
- When does the depth of v (distance to root) increase?
- Only when v's tree is attached under *another* root during a UNION.
- By Union-by-Size, this happens only if the *other* tree was $\geq$ the size of v's current tree.
- $\implies$ Every time v's depth increases, the size of its *new* component **at least doubles**.

- Max component size is n. Size can double $\leq \log_2 n$ times.
- Therefore, the depth of any node is always $O(\log n)$.
- FIND operations take $O(\log n)$ time! UNION takes $O(\log n)$ (due to FINDs).
- With "path compression," it's even faster - nearly constant time!

Kruskal's Final Running Time (Revisited)

Let's re-evaluate the total work using our faster Union-Find.

- **1. Sort edges:** $O(m \log n)$.
- **2. Initialize Union-Find:** $O(n)$.
- **3. Main Loop (m iterations):**
  - $2 \times m$ FIND operations: Total $O(m \log n)$.
  - $n - 1$ UNION operations: Total $O(n \log n)$.

Grand Total:

$$O(m \log n) + O(n) + O(m \log n) + O(n \log n) = \mathbf{O(m \log n)}$$

(Sorting is usually the bottleneck!)

Can We Do Better? (State of the Art in MST Research)

Can we beat $O(m \log n)$? **Yes — in theory!**

- *Randomized:* $O(m)$ expected time (Karger–Klein–Tarjan, 1995).
- *Deterministic:* $O(m\,\alpha(n))$ (Chazelle, 2000). $\alpha(n) =$ inverse Ackermann function (< 5 for all practical n).
- Pettie–Ramachandran (2002): asymptotically optimal but unknown exact runtime.

Open Questions

- Still no **simple, deterministic** $O(m)$ MST algorithm.

Summary: Two Algorithms, One Goal

We learned two “incredibility fast” greedy algorithms for the MST problem.

Prim’s Algorithm

- “Grows a single tree”
- Greedy Choice: Add cheapest edge from X to $V - X$.
- Data Structure: Heap
- Runtime: $\mathbf{O}(\mathbf{m} \log \mathbf{n})$

Kruskal’s Algorithm

- “Merges a forest”
- Greedy Choice: Add cheapest edge that *doesn’t* form a cycle.
- Data Structure: Union-Find
- Runtime: $\mathbf{O}(\mathbf{m} \log \mathbf{n})$

Both are correct because they cleverly exploit *The Cut Property*.

References

Roughgarden, T. (2022).
Algorithms Illuminated: Omnibus Edition.
Soundlikeyourself Publishing, LLC.