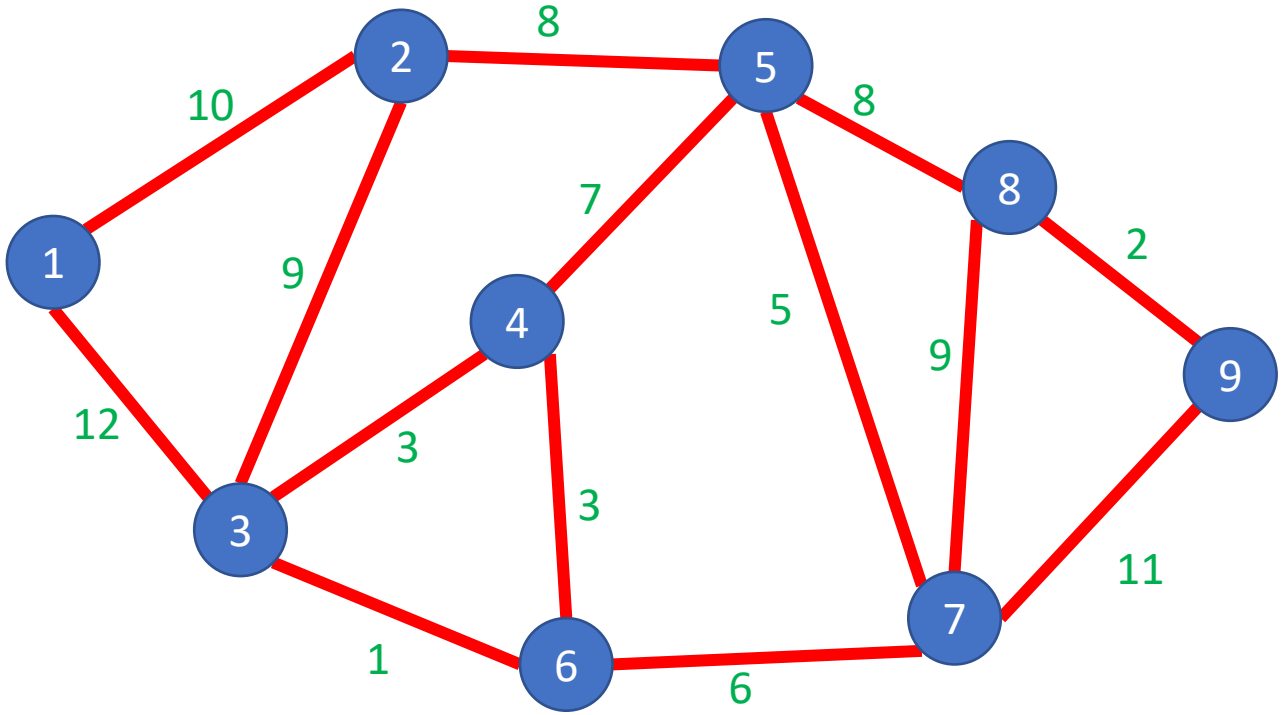
CSE 332 Spring 2026
Lecture 18: Graphs 4, MSTs

Nathan Brunelle

http://www.cs.uw.edu/332

Single-Source Shortest Path



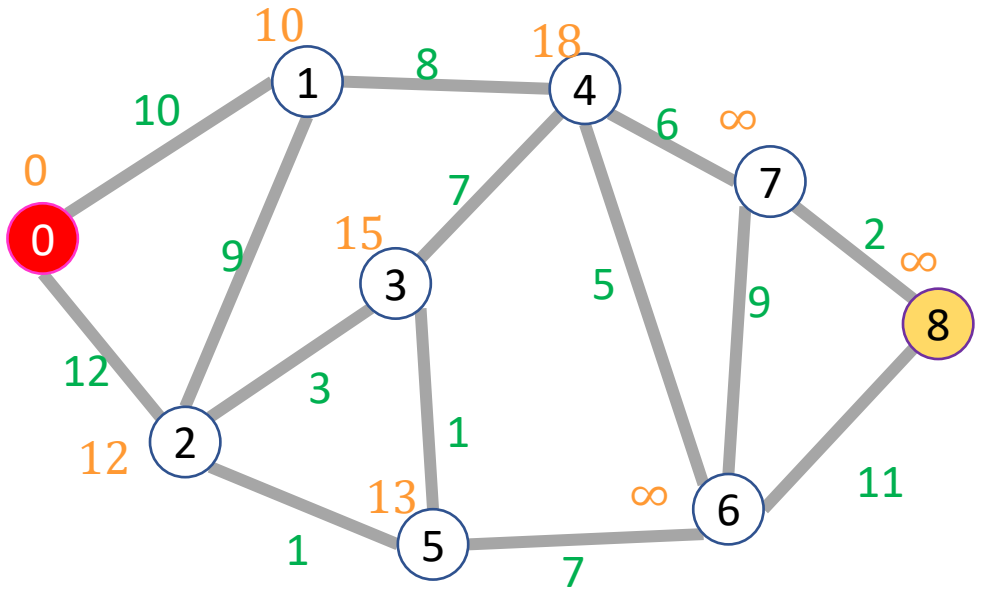
Find the quickest way to get from UW to each of these other places

Given a graph $G = (V, E)$ and a start node $s \in V$, for each $v \in V$ find the least-weight path from $s \to v$ (call this weight $\delta(s, v)$)

(assumption: all edge weights are positive)

Dijkstra's Algorithm

- Input: graph with **no negative edge weights**, start node s, optional end node t
- Behavior: Start with node s, repeatedly go to the incomplete node "nearest" to s
- Output:
  - Distance from start to end
  - Distance from start to every node



Dijkstra's

```
Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract();
    mark curr as "done"
    for each neighbor v of curr:
        d = distance to curr + weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            PQ.decreaseKey(v, d)
```

Loops $|E|$ times

Worst case $\Theta(\log|V|)$ each

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Seen = added to the priority queue
Done = removed from the priority queue
When it's done we've found the shortest path to that node

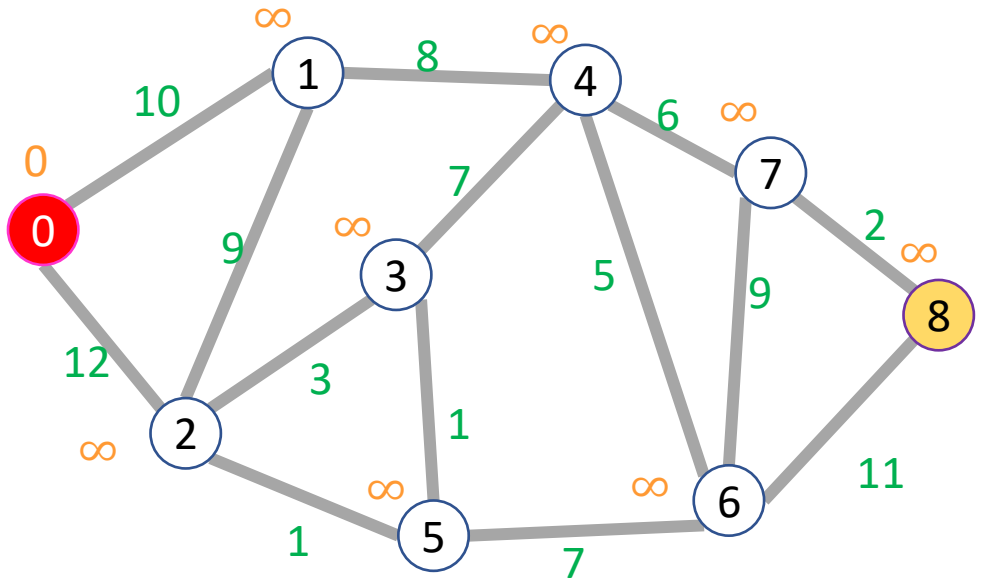
Running time: $\Theta(|E|\log|V|)$

Dijkstra's Algorithm (1/8)

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	F	0
1	F	F	∞
2	F	F	∞
3	F	F	∞
4	F	F	∞
5	F	F	∞
6	F	F	∞
7	F	F	∞
8	F	F	∞

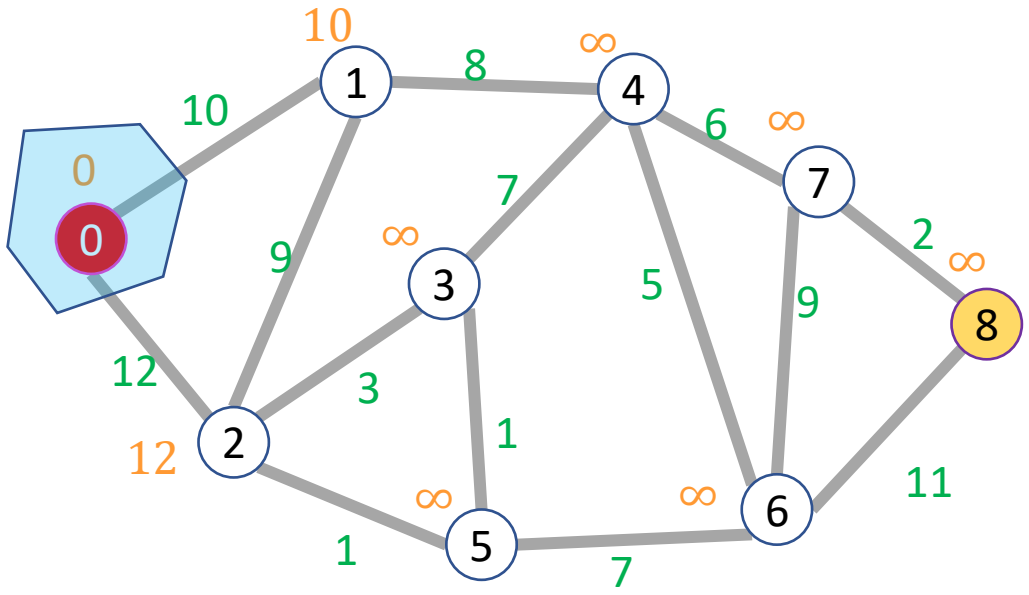


Dijkstra's Algorithm (2/8)

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	T	0
1	T	F	10
2	T	F	12
3	F	F	∞
4	F	F	∞
5	F	F	∞
6	F	F	∞
7	F	F	∞
8	F	F	∞

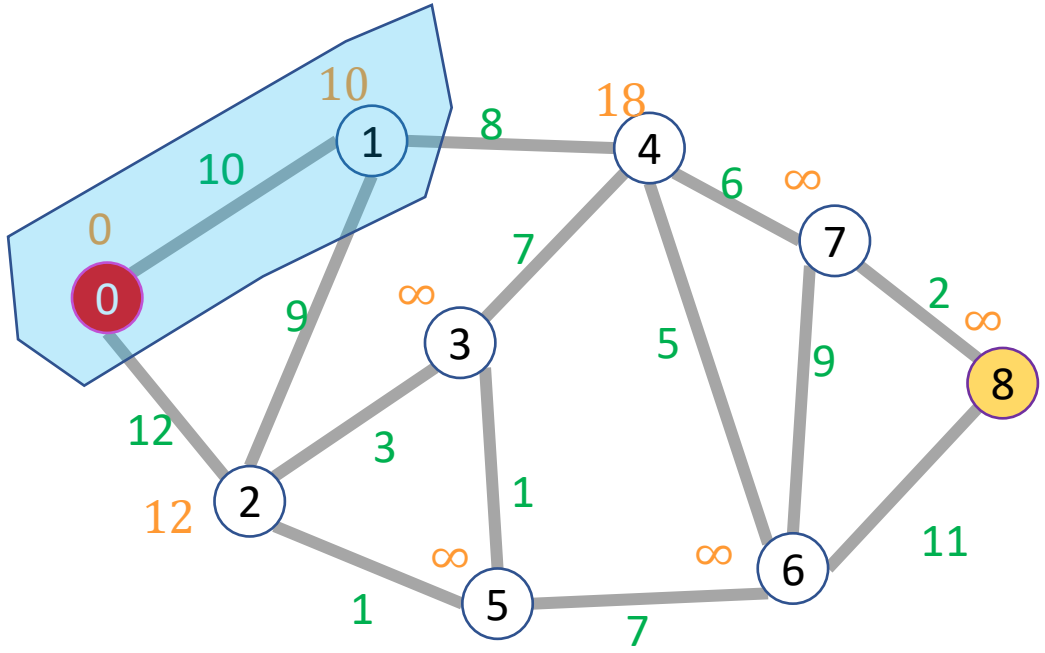


Dijkstra's Algorithm (3/8)

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	F	12
3	F	F	∞
4	T	F	18
5	F	F	∞
6	F	F	∞
7	F	F	∞
8	F	F	∞

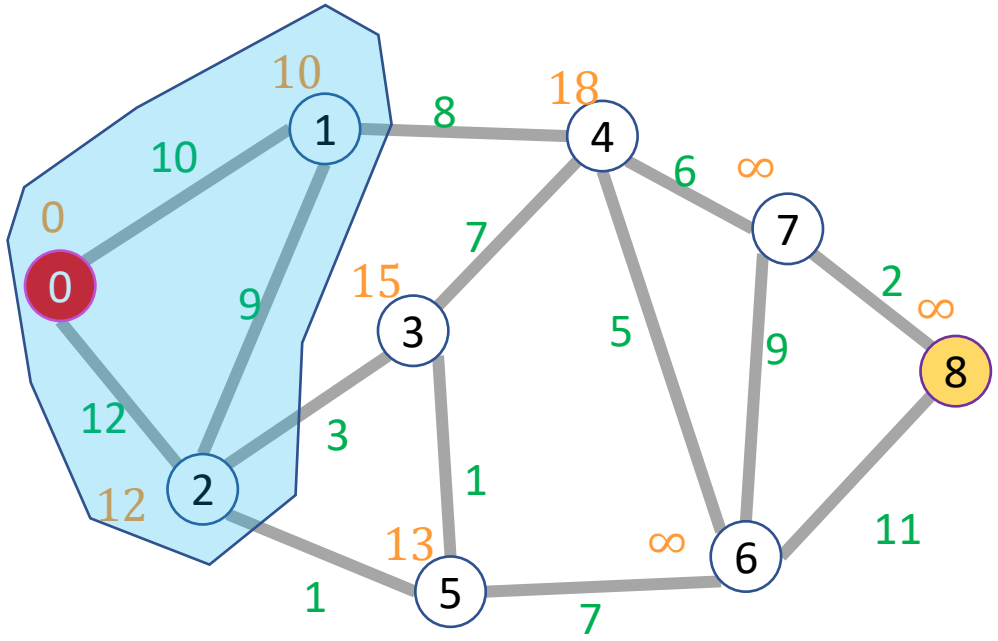


Dijkstra's Algorithm (4/8)

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	T	12
3	T	F	15
4	T	F	18
5	T	F	13
6	F	F	∞
7	F	F	∞
8	F	F	∞

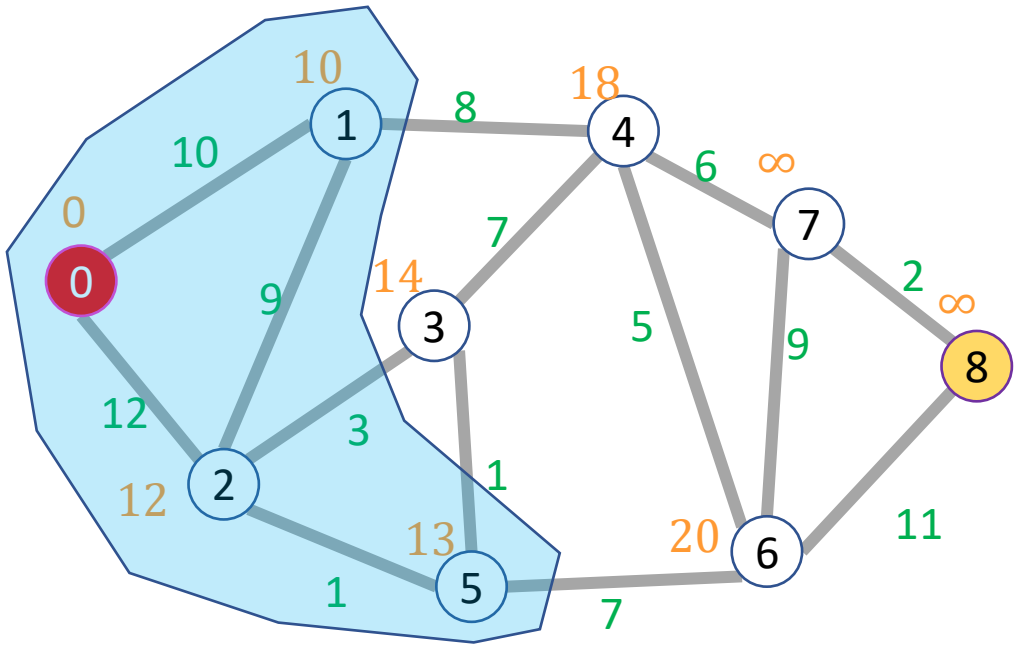


Dijkstra's Algorithm (5/8)

Idea: When a node is the closest not-done thing to the start, we have found its shortest path

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	T	12
3	T	F	14
4	T	F	18
5	T	T	13
6	T	F	20
7	F	F	∞
8	F	F	∞

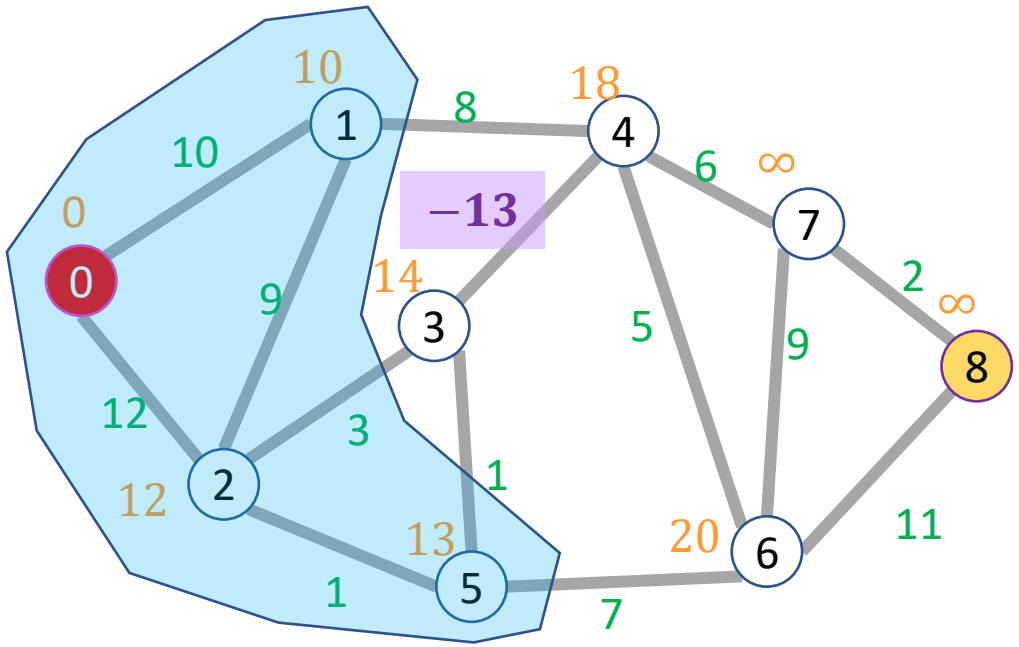


Dijkstra's Algorithm (6/8)

What if we had a negative-weight edge?

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	T	12
3	T	F	14
4	T	F	18
5	T	T	13
6	T	F	20
7	F	F	∞
8	F	F	∞

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

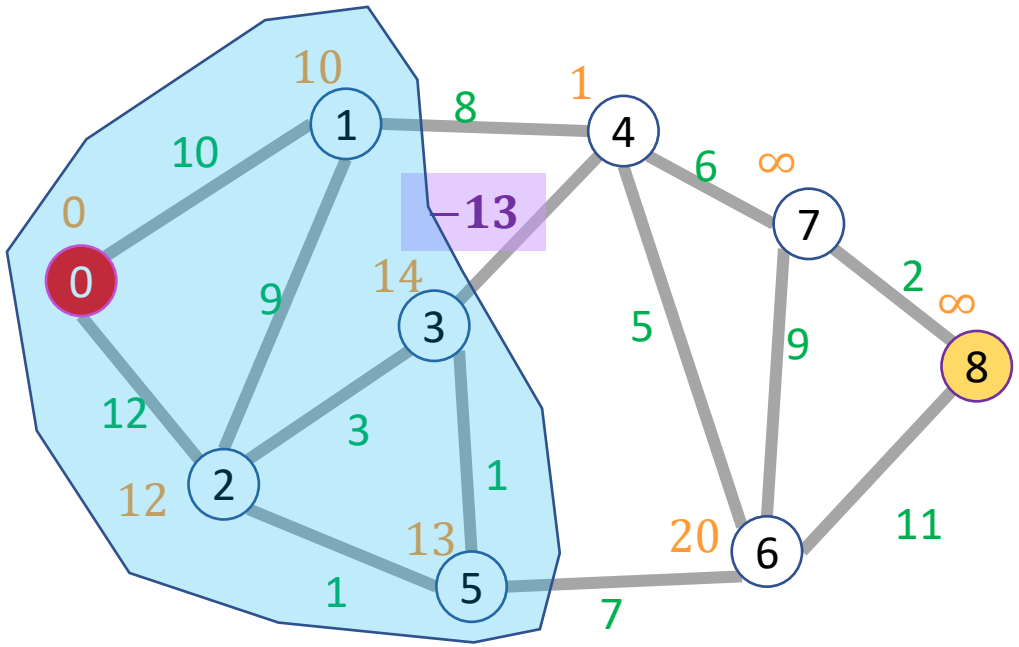


Dijkstra's Algorithm (7/8)

What if we had a negative-weight edge?

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	T	12
3	T	T	14
4	T	F	1
5	T	T	13
6	T	F	20
7	F	F	∞
8	F	F	∞

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path



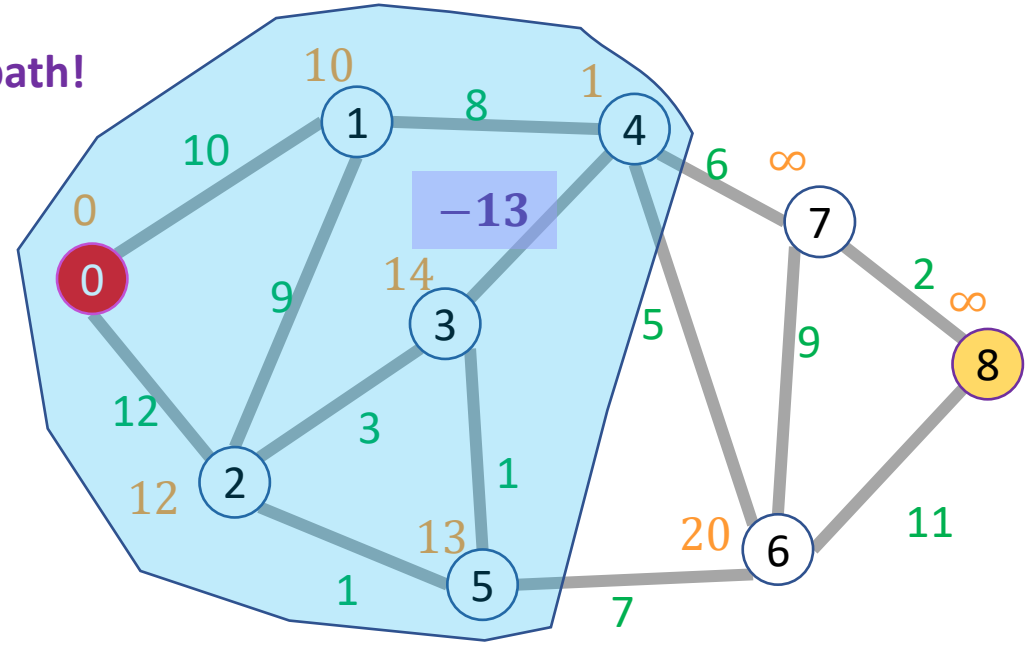
Dijkstra's Algorithm (8/8)

What if we had a negative-weight edge?

Extract a node from priority queue (making it "done")
Mark extracted node as seen
for each not-done neighbor:
    Update its distance if we found a better path

Node	Seen?	Done?	Distance
0	T	T	0
1	T	T	10
2	T	T	12
3	T	T	14
4	T	T	1
5	T	T	13
6	T	F	20
7	F	F	∞
8	F	F	∞

There's a better path!

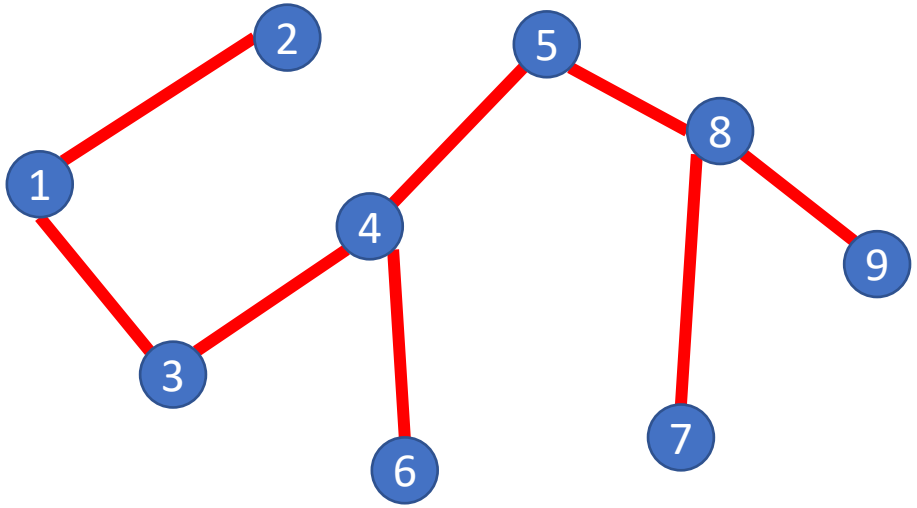


Dijkstra's Algorithm - Pseudocode

```
int dijkstras(graph, start, end){
        distances = [∞, ∞, ∞,…];  // one index per node
        seen = [False,False,False,…];  // one index per node
        done = [False,False,False,…];  // one index per node
        PQ = new MinHeap();
        PQ.insert(0, start);  // priority=0, value=start
        distances[start] = 0;
        while (!PQ.isEmpty){
                current = PQ.extract();
                done[current] = True;
                for (neighbor : current.neighbors){
                        new_dist = distances[current]+weight(current,neighbor);
                        if (! seen[neighbor]){
                                seen[neighbor] = True;
                                distances[neighbor] = new_dist;
                                PQ.insert(new_dist, neighbor);
                        }
                        else if (! done[neighbor] && new_dist < distances[neighbor]){
                                distances[neighbor] = new_dist;
                                PQ.decreaseKey(new_dist,neighbor);
                        }
                }
        }
        return distances[end]
}
```

Definition: Tree

A connected graph with no cycles

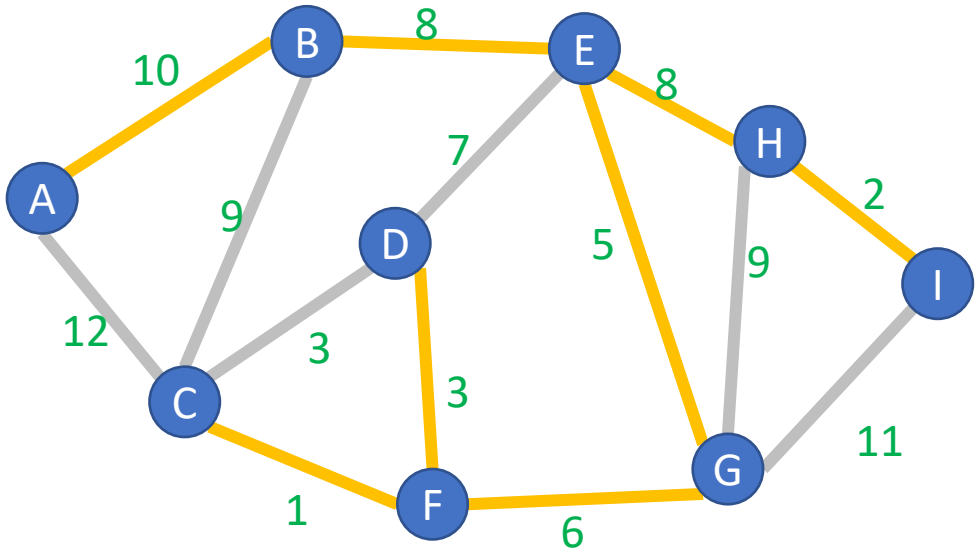


A Tree

Note: A tree does not need a root, but they often do!

Definition: Spanning Tree

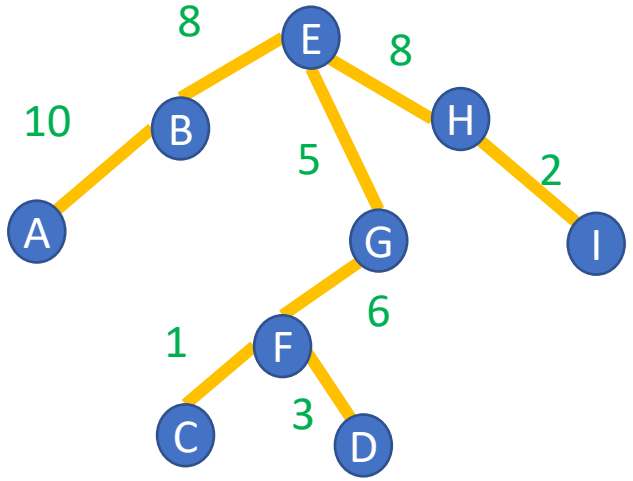
A Tree $\boldsymbol{T} = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$



How many edges does T have?

$V - 1$

Pick some arbitrary root node and rearrange tree

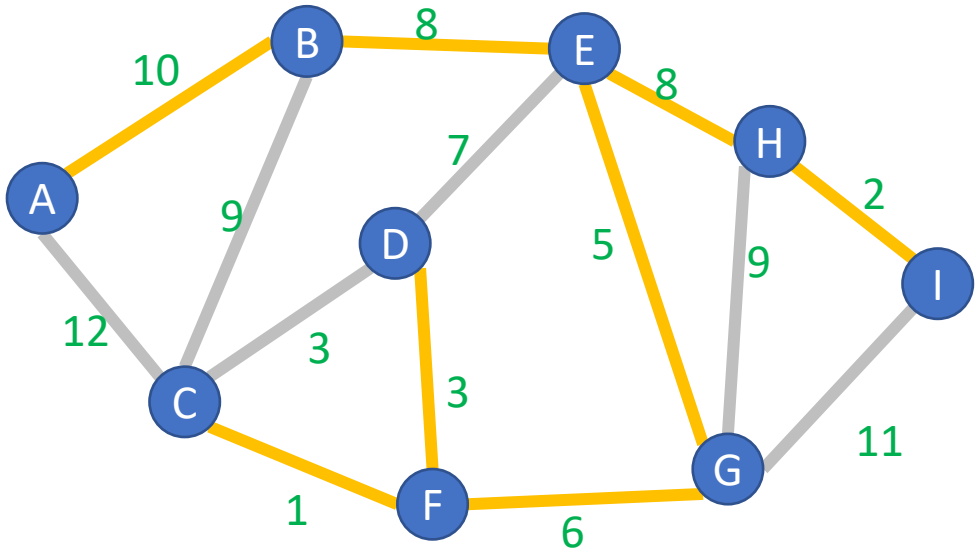


Any set of V-1 edges in the graph that doesn't have any cycles is guaranteed to be a spanning tree!

Any set of V-1 edges that connects all the nodes in the graph is guaranteed to be a spanning tree!

Definition: Minimum Spanning Tree

A Tree $\boldsymbol{T} = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$, that has minimal cost



$$Cost(T) = \sum_{e \in E_T} w(e)$$

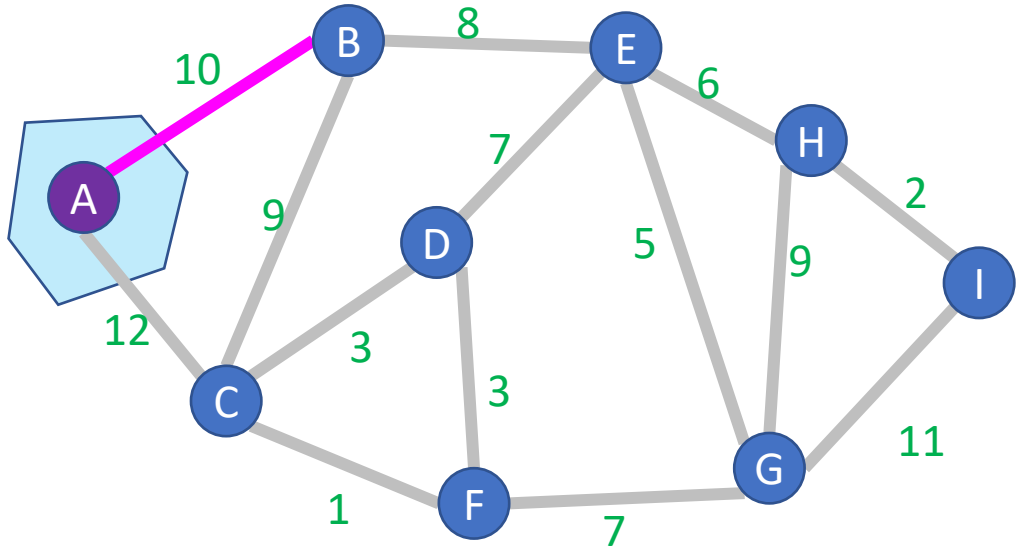
Prim's Algorithm (1/4)

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



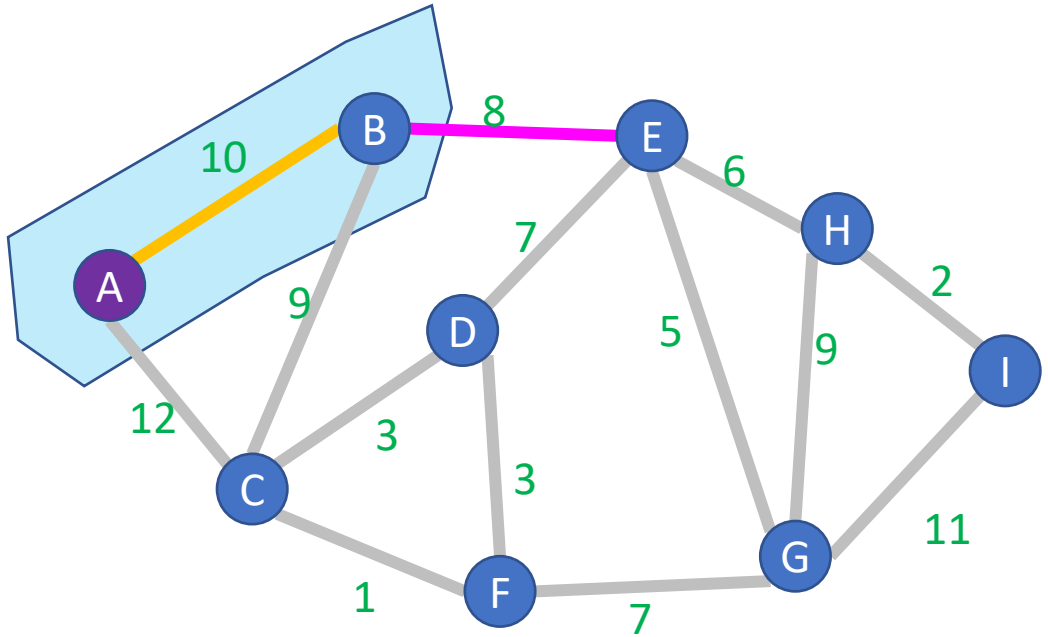
Prim's Algorithm (2/4)

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



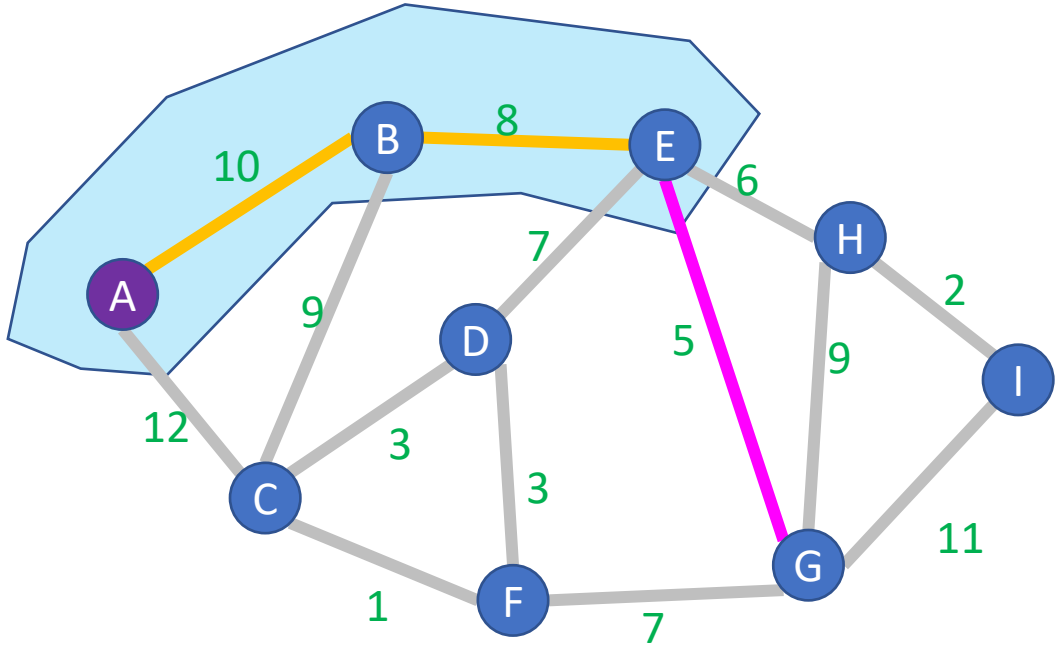
Prim's Algorithm (3/4)

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



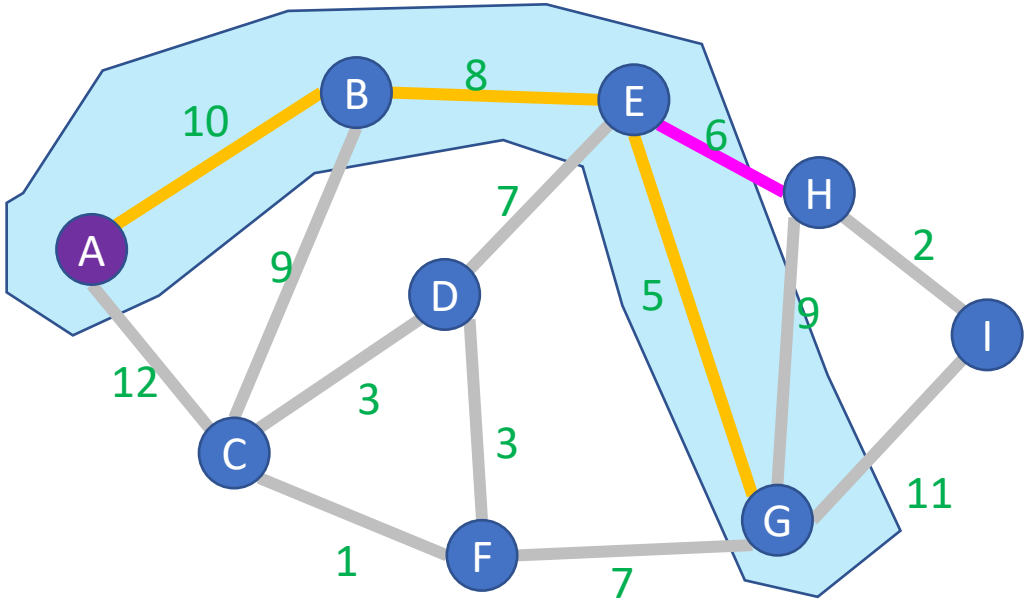
Prim's Algorithm (4/4)

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

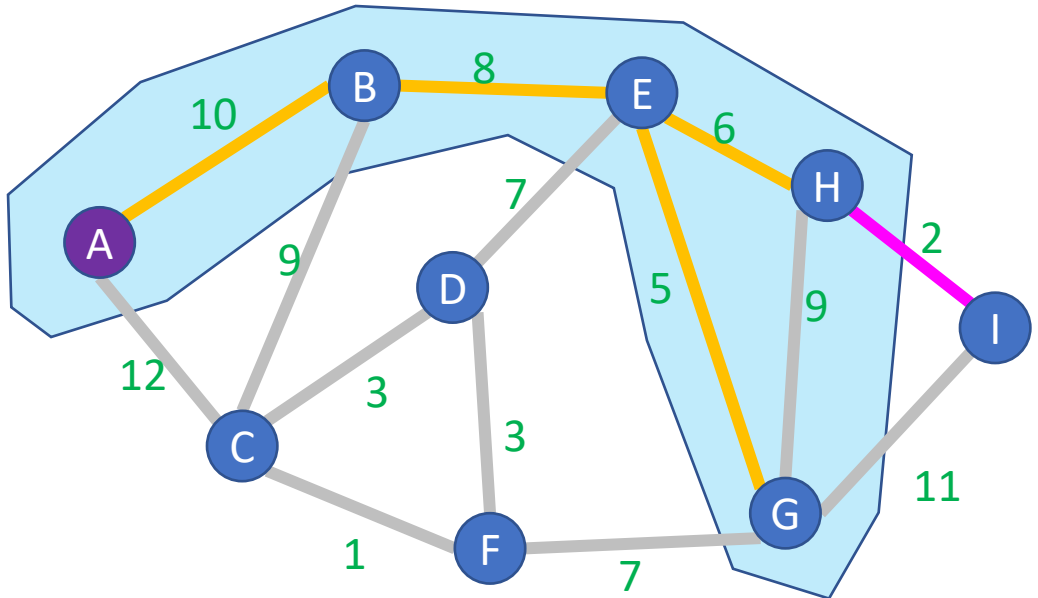
Add the min-weight edge which connects to node in A with a node not in A



Prim's Algorithm Running Time

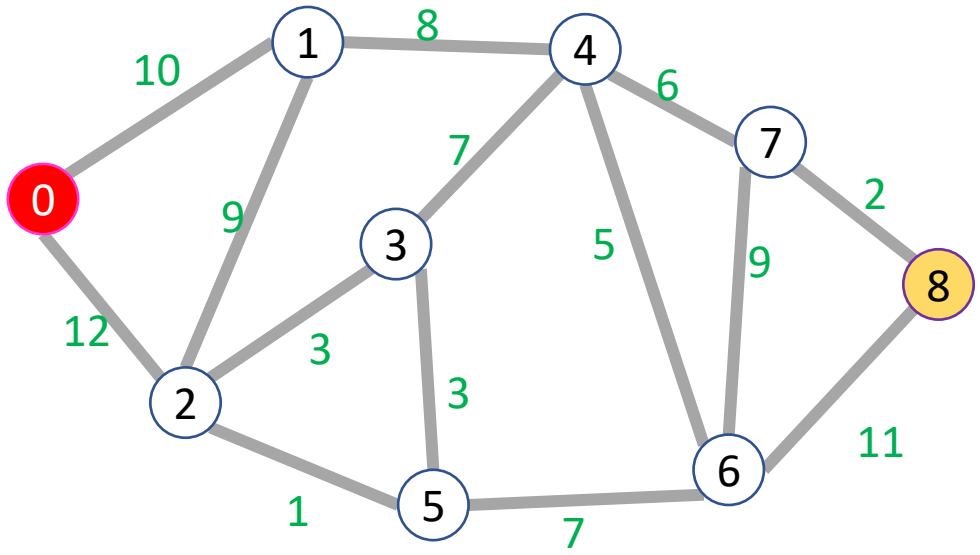
Keep edges in a Heap
$O(E \log V)$

Start with an empty tree A
Pick a start node
Repeat $V - 1$ times:
Add the min-weight edge which connects to node in A with a node not in A



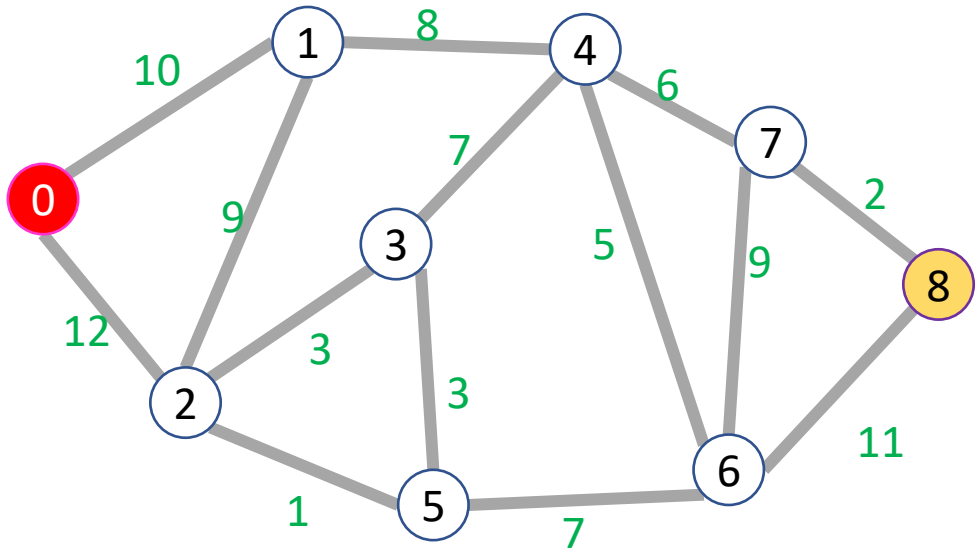
Recall Dijkstra's Algorithm

Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract()
    mark curr as "done"
    add the edge (previous of curr, curr) to the spanning tree
    for each neighbor v of curr:
        d = distance to curr + weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            previous of v = curr
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            previous of v = curr
            PQ.decreaseKey(v, d)
Return the spanning tree



Prim's Algorithm

Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract()
    mark curr as "done"
    add the edge (previous of curr, curr) to the spanning tree
    for each neighbor v of curr:
        d = weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            previous of v = curr
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            previous of v = curr
            PQ.decreaseKey(v, d)
Return the spanning tree



Avoiding Decrease Key

- Java's `PriorityQueue` interface does not have `decreaseKey`
- Two strategies to avoid this:
  - Allow copies of nodes to be on the PQ:
    - Every time we would want to decrease the key, instead add the node again, then make sure we skip "done" nodes when extracting.
  - Store edges on the priority queue instead of nodes.
    - If the destination of the edge has already been extracted, ignore that edge. Also avoids keeping track of previous nodes.
- Both change the running time from $\Theta(E \log V)$ to $\Theta(E \log E)$
  - Since the maximum size of the priority queue is E
  - Because $E \leq V^2$, the worst case asymptotic running time does not change

Prim's Algorithm – Original Recipe

```
Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract()
    mark curr as "done"
    add the edge (previous of curr, curr) to the spanning tree
    for each neighbor v of curr:
        d = weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            previous of v = curr
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            previous of v = curr
            PQ.decreaseKey(v, d)
Return the spanning tree
```

Prim's Algorithm – Duplicate Nodes on the PQ

```
Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    curr = PQ.extract()
    if(curr is "done") then continue
    mark curr as "done"
    add the edge (previous of curr, curr) to the spanning tree
    for each neighbor v of curr:
        d = weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            previous of v = curr
            PQ.add(v, d);
        if v is not "done" && d < distance to v:
            distance to v = d
            previous of v = curr
            PQ.add(v, d)
Return the spanning tree
```

Prim's Algorithm – Edges on the PQ

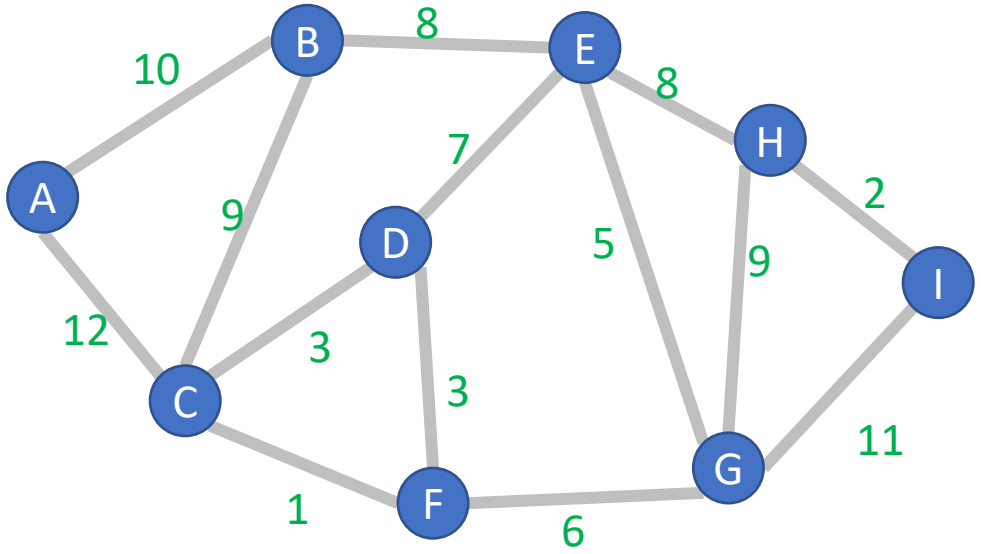
Distance to start = 0
Add the start node to PQ with priority 0
Mark start as "seen"
While the PQ is not empty:
    currEdge = PQ.extract()
    curr = currEdge.destination
    if(curr is "done") then continue
    mark curr as "done"
    add **currEdge** to the spanning tree
    for each neighbor v of curr:
        d = weight of (curr,v)
        if v is not "seen":
            mark v as "seen"
            distance to v = d
            PQ.add(**(curr,v)**,d);
        if v is not "done" && d < distance to v:
            distance to v = d
            PQ.**add**(**(curr,v)**, d)
Return the spanning tree

Kruskal's Algorithm (1/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

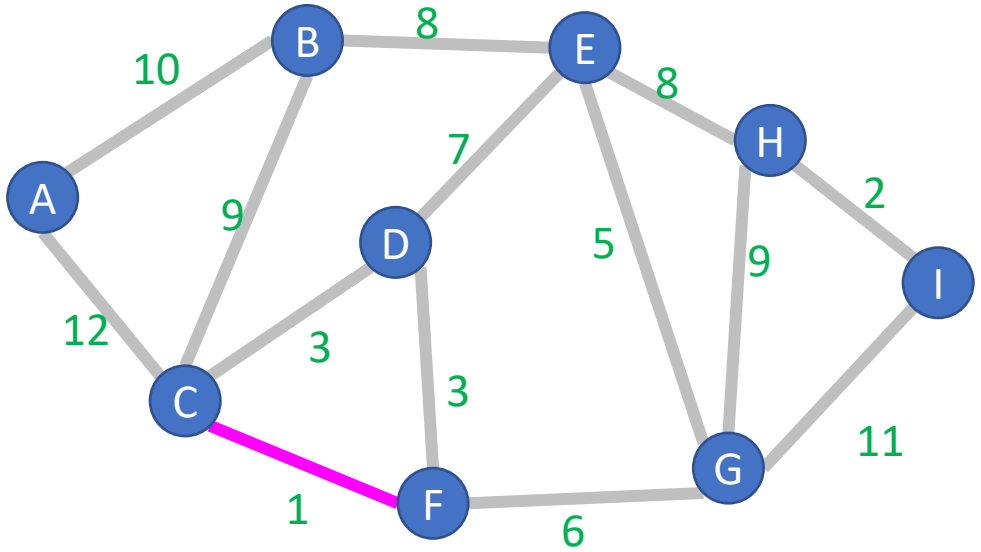


Kruskal's Algorithm (2/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

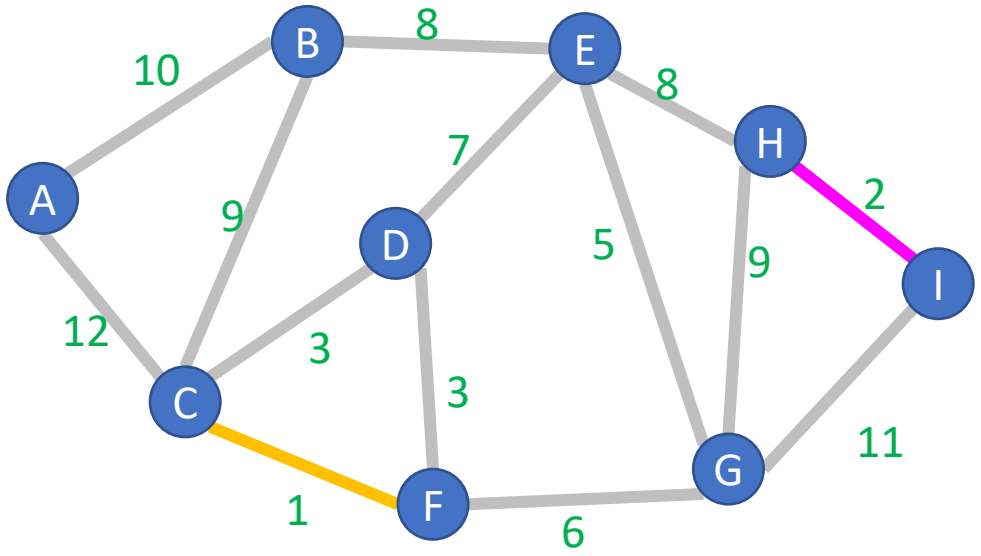


Kruskal's Algorithm (3/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

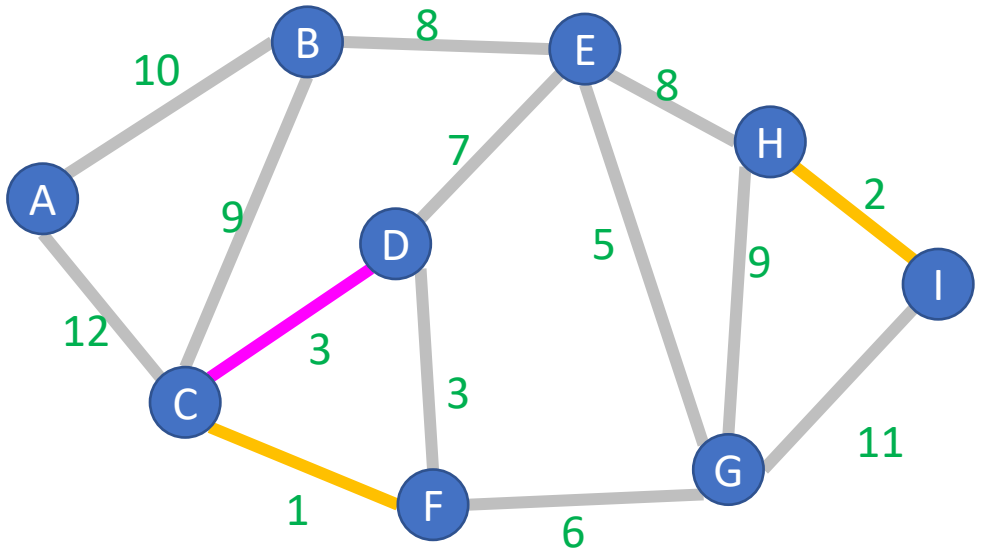


Kruskal's Algorithm (4/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

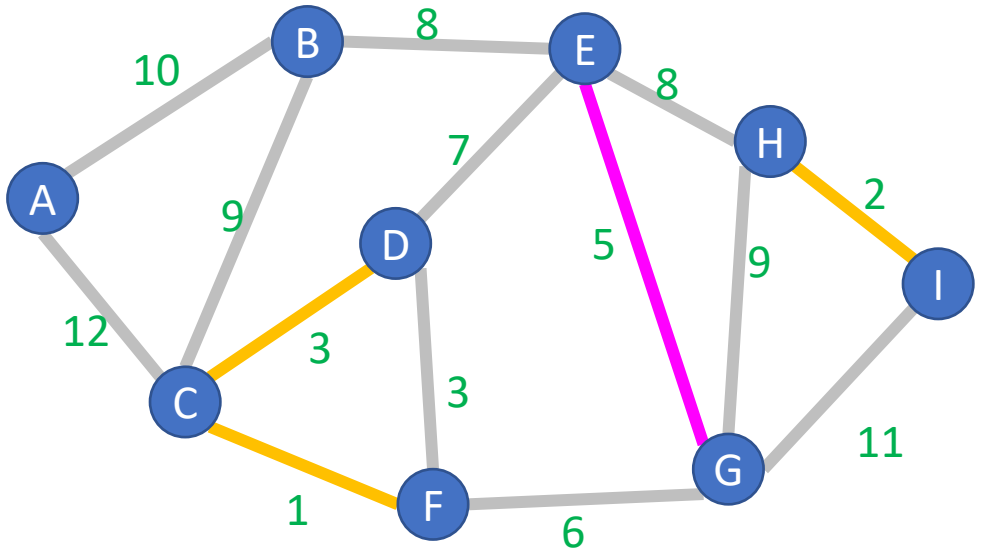


Kruskal's Algorithm (5/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

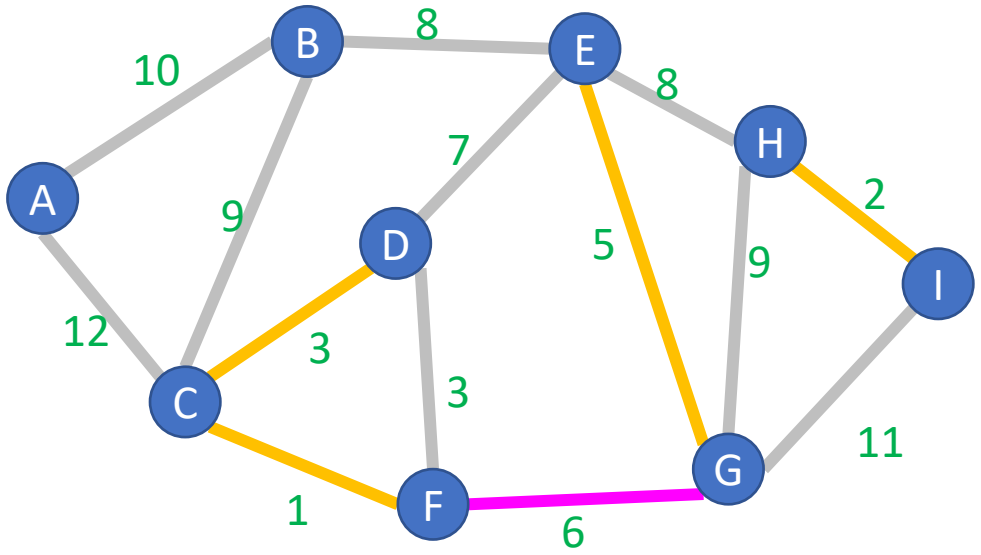


Kruskal's Algorithm (6/6)

Start with an empty tree A

Do $|V| - 1$ times:

Add to A the lowest-weight edge that does not create a cycle

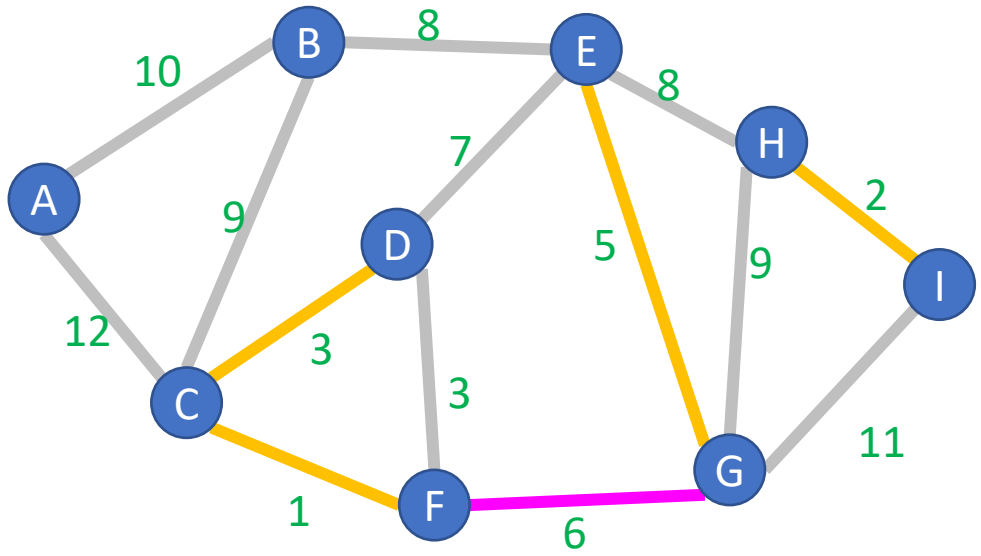


Kruskal's Algorithm Runtime

Start with an empty tree A

Repeat $V - 1$ times:

Add the min-weight edge that doesn't cause a cycle



Keep edges in a Disjoint-set data structure (very fancy)

$O(E \log V)$

Why do these work?

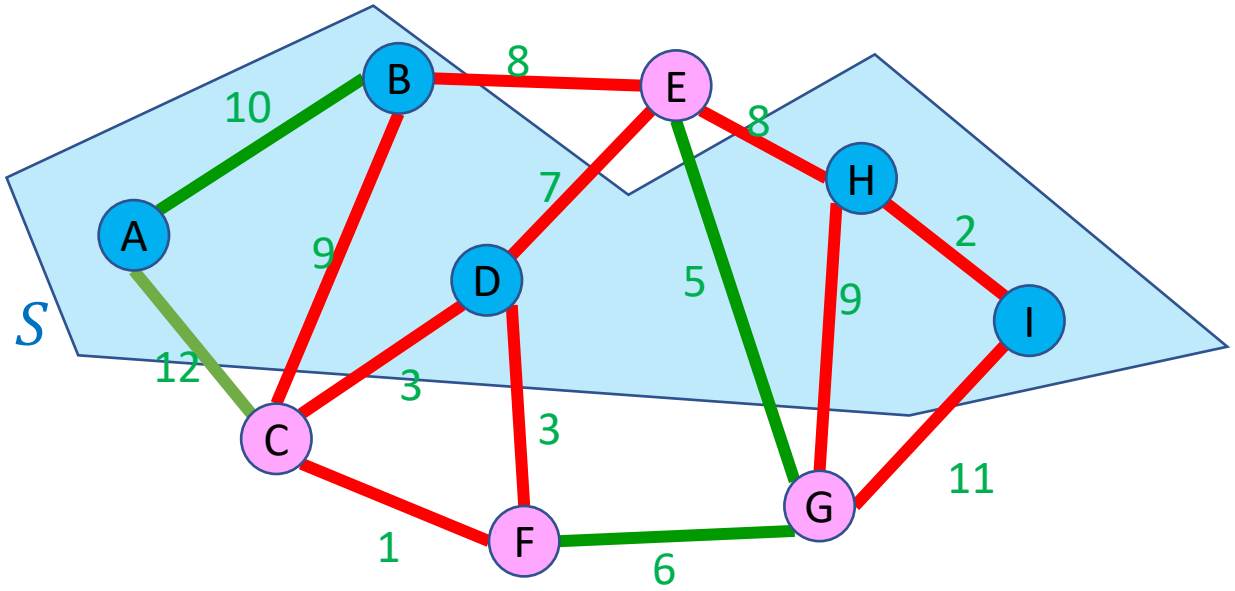
- To argue that Prim's, Kruskal's produce a minimum spanning tree:
  - First we show that the algorithm produces a spanning tree
    - Show two of:
      - Connected
      - Acyclic
      - $V - 1$ edges
  - Then we show that it is a minimum spanning tree
    - Show all edges chosen are MST edges
      - Using the "Cut Theorem"

Prim's, Kruskal's Produce a Spanning Tree

- Prim's:
  - The graph is connected:
    - We will eventually add each node to the PQ, we find a path to the node when that happens
  - The graph has no cycles:
    - We only ever select edges that lead to a new node that's not yet part of the tree
- Kruskal's
  - The graph has $|V| - 1$ edges
    - The algorithm is designed to select exactly this many edges
  - The graph is acyclic
    - The algorithm specifically checks that it never creates a cycle

Definition: Cut

A Cut of graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



Edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$ (or opposite), e.g. (A, C)

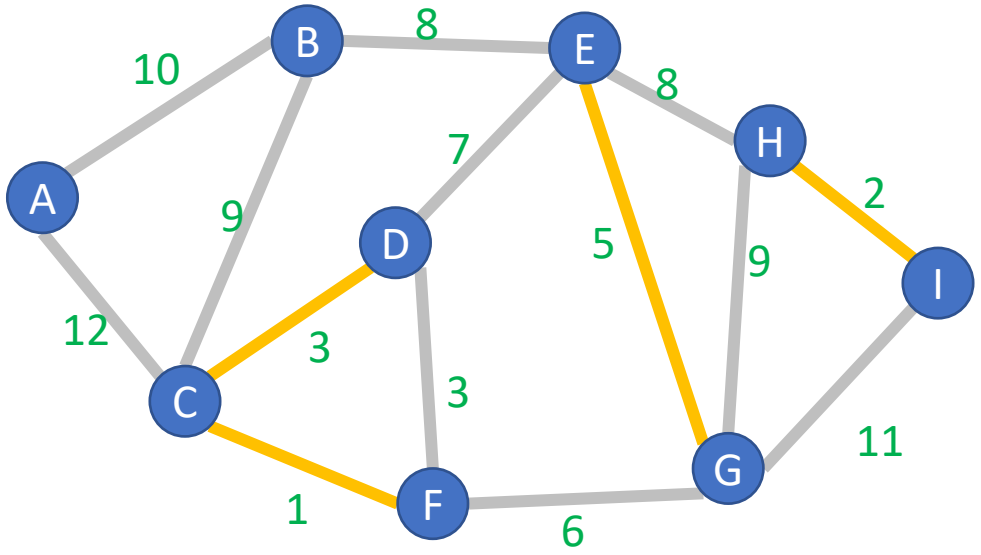
A set of edges R Respects a cut if no edges cross the cut
e.g. $R = \{(A, B), (E, G), (F, G)\}$

Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.

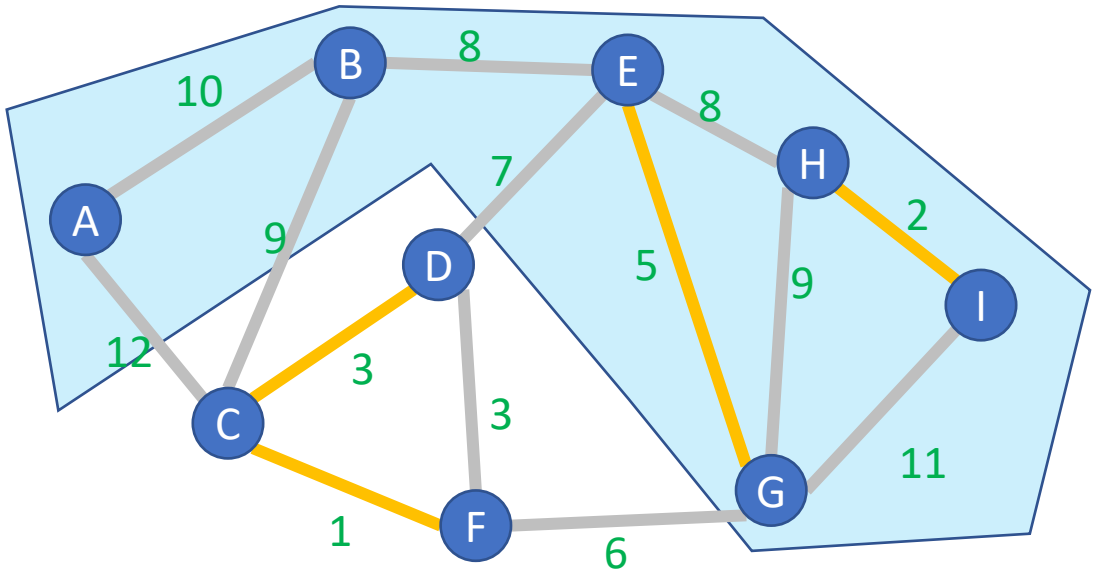
Cut Theorem (1/4)

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



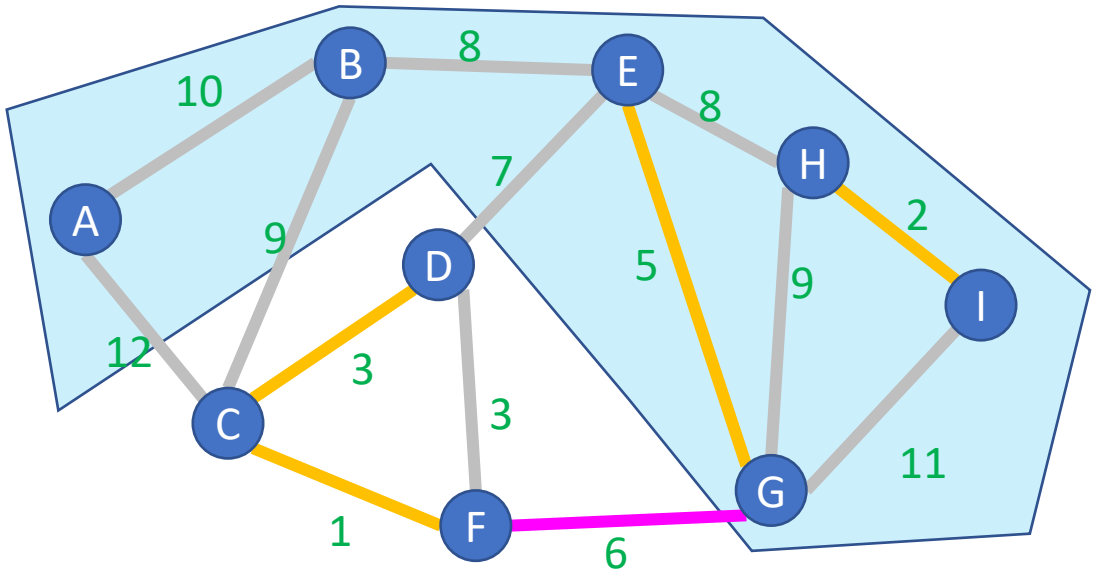
Cut Theorem (2/4)

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



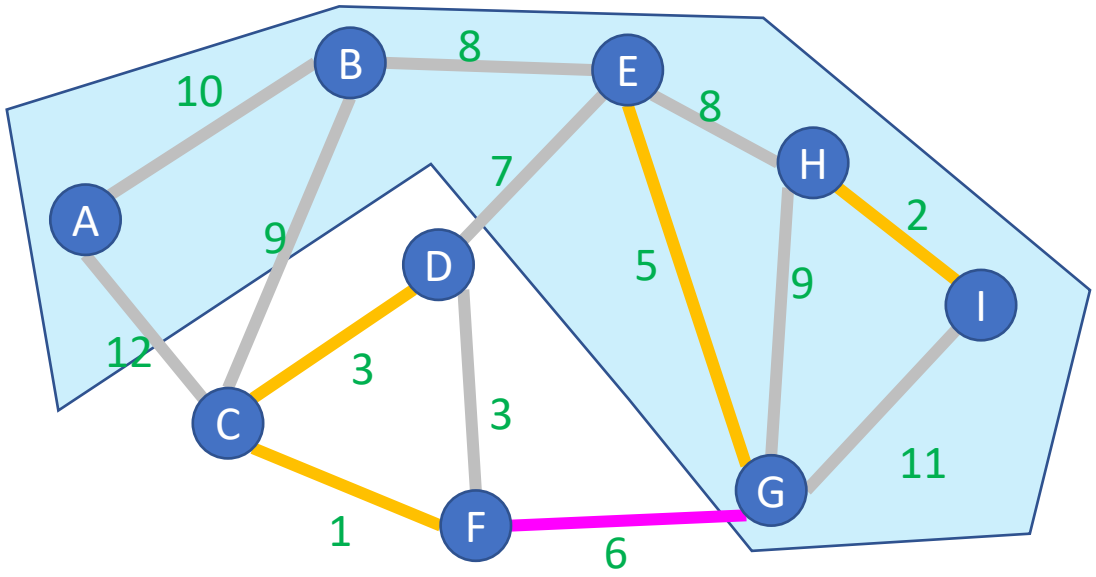
Cut Theorem (3/4)

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



Cut Theorem (4/4)

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.

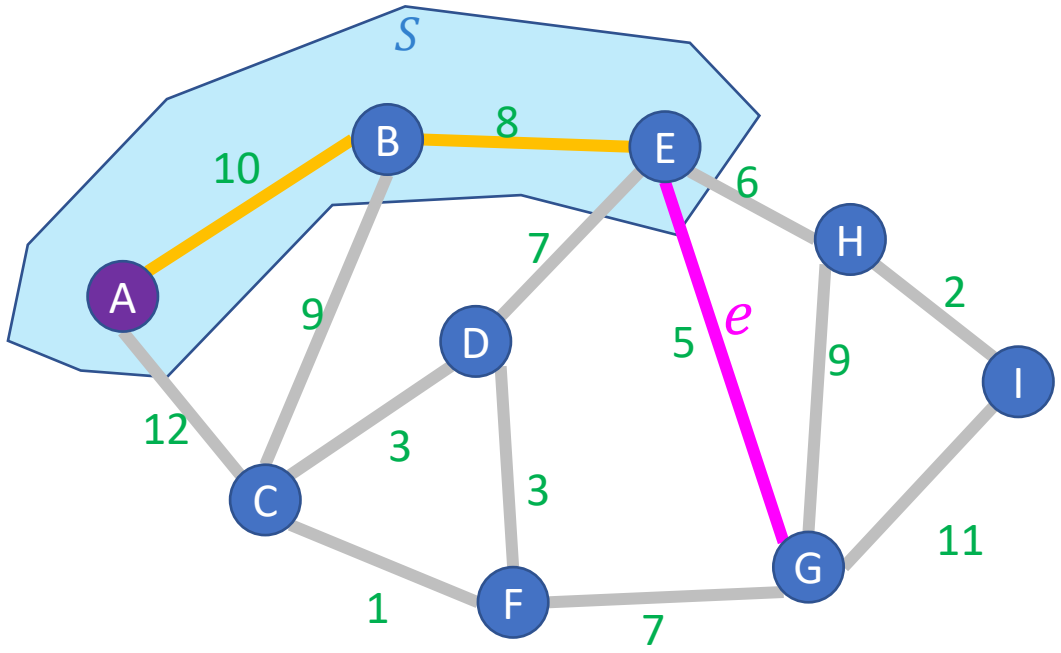


Proof of Prim's Algorithm

Start with an empty tree A

Repeat $V - 1$ times:

Add the min-weight edge that connects to a node not currently in the tree



Proof: By Structural Induction

Suppose we have some arbitrary set of edges A that Prims's has already selected to include in the MST. $e = (E, G)$ is the edge Prims's selects to add next

We know that there cannot exist a path from E to G using only edges in A because G has not been removed from the priority queue

We can cut the graph therefore into 2 disjoint sets:

- Nodes that have been removed from the priority queue
- All other nodes

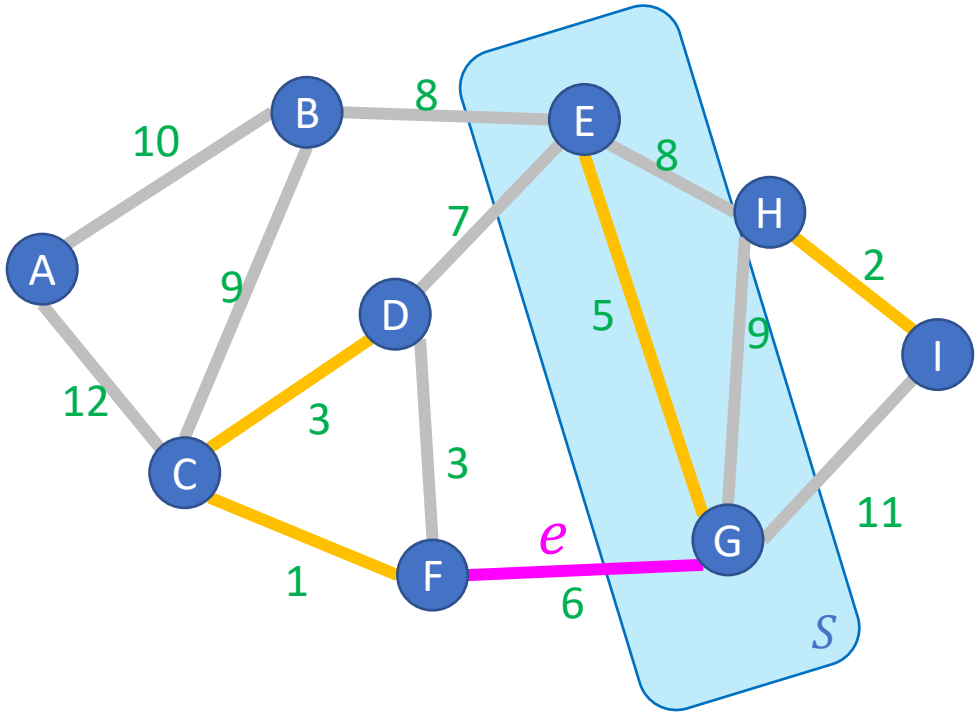
e is the minimum cost edge that crosses this cut, so by the Cut Theorem, Prim's only selects MST edges!

Proof of Kruskal's Algorithm

Start with an empty tree A

Repeat $V - 1$ times:

Add the min-weight edge that doesn't cause a cycle



Proof: Suppose we have some arbitrary set of edges A that Kruskal's has already selected to include in the MST. $e = (F, G)$ is the edge Kruskal's selects to add next

We know that there cannot exist a path from F to G using only edges in A because e does not cause a cycle

We can cut the graph therefore into 2 disjoint sets:

- nodes reachable from G using edges in A
- All other nodes

e is the minimum cost edge that crosses this cut, so by the Cut Theorem, Kruskal's is optimal!

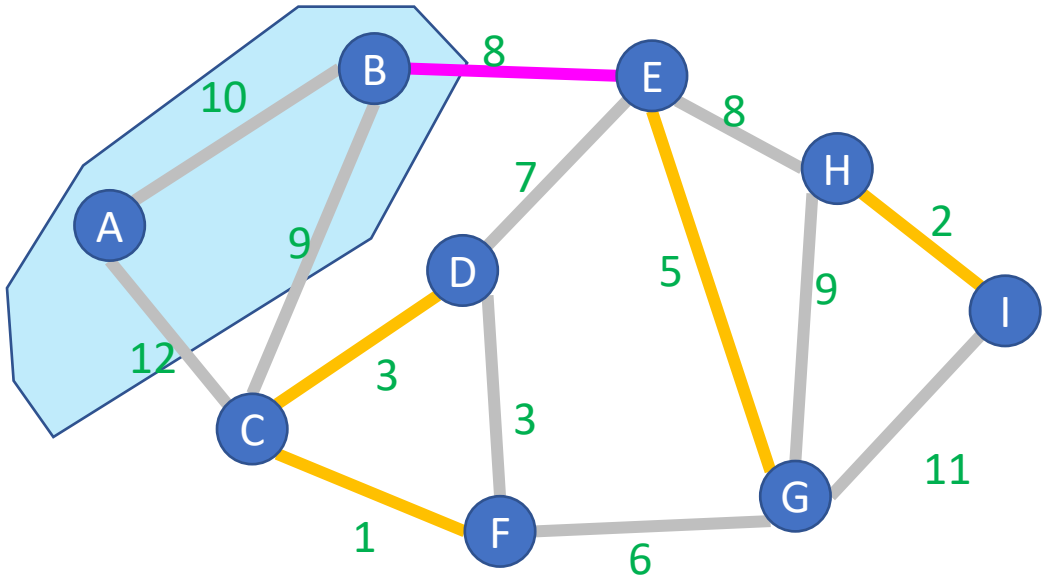
General MST Algorithm

Start with an empty tree A

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects (typically implicitly)

Add the min-weight edge which crosses $(S, V - S)$



Generic MST Algorithm as Prim's

Start with an empty tree A

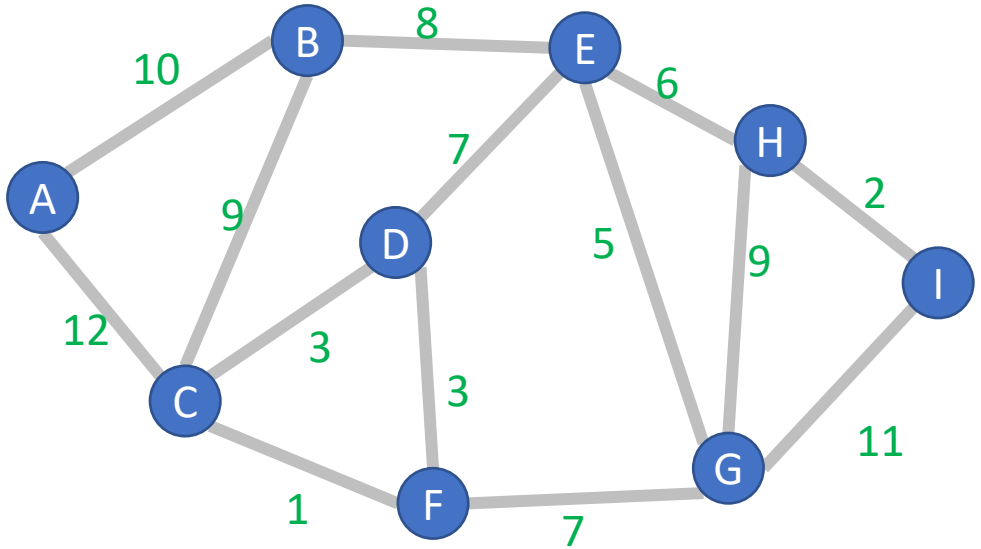
Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects

Add the min-weight edge which crosses $(S, V - S)$

S is all endpoints of edges in A

e is the min-weight edge that grows the tree



Generic MST Algorithm as Kruskal's

Start with an empty tree A

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects

Add the min-weight edge which crosses $(S, V - S)$

S is all nodes reaching from 1 endpoint of e

e is the least-weight edge that does not create a cycle