CSE 332 Autumn 2023
Lecture 26: Topological Sort and Minimum Spanning Trees

Nathan Brunelle

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

Bank Account

```
Public static final Object BANK = new Object();
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        synchronized void transferTo(int amt, BankAccount a) {
                timer.start();
                lk.lock();
                other thread
                        }
}
```

The Deadlock

Expected Behavior:
Thread 2 items from a stack are popped in LIFO order

Thread 1:

x.transferTo(1,y);

Thread 2:

y.transferTo(1,x);

acquire lock for account x b/c transferTo is synchronized **acquire lock for account y** b/c deposit is synchronized **release lock for account y** after depost **release lock for account x** at end of transferTo	**acquire lock for account y** b/c transferTo is synchronized **acquire lock for account x** b/c deposit is synchronized **release lock for account x** after deposit **release lock for account y** at end of transferTo

The Deadlock

Expected Behavior:
Thread 2 items from a stack are popped in LIFO order

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

acquire lock for account x b/c transferTo is synchronized	
	acquire lock for account y b/c transferTo is synchronized
acquire lock for account y b/c deposit is synchronized	
	acquire lock for account x b/c deposit is synchronized
release lock for account y after depost	
	release lock for account x after deposit
release lock for account x at end of transferTo	
	release lock for account y at end of transferTo

Resolving Deadlocks

- Deadlocks occur when there are multiple locks necessary to complete a task and different threads may obtain them in a different order
- Option 1:
  - Have a coarser lock granularity
  - E.g. one lock for ALL bank accounts
- Option 2:
  - Have a finer critical section so that only one lock is needed at a time
  - E.g. instead of a synchronized transferTo, have the withdraw and deposit steps locked separately
- Option 3:
  - Force the threads to always acquire the locks in the same order
  - E.g. make transferTo acquire both locks before doing either the withdraw or deposit, make sure both threads agree on the order to aquire

Option 1: Coarser Locking

```
static final Object BANK = new Object();
class BankAccount {
    …
    synchronized void withdraw(int amt) {…}
    synchronized void deposit(int amt) {…}
    void transferTo(int amt, BankAccount a) {
        synchronized(BANK){
            this.withdraw(amt);
            a.deposit(amt);
        }
    }
}
```

Option 2: Finer Critical Section

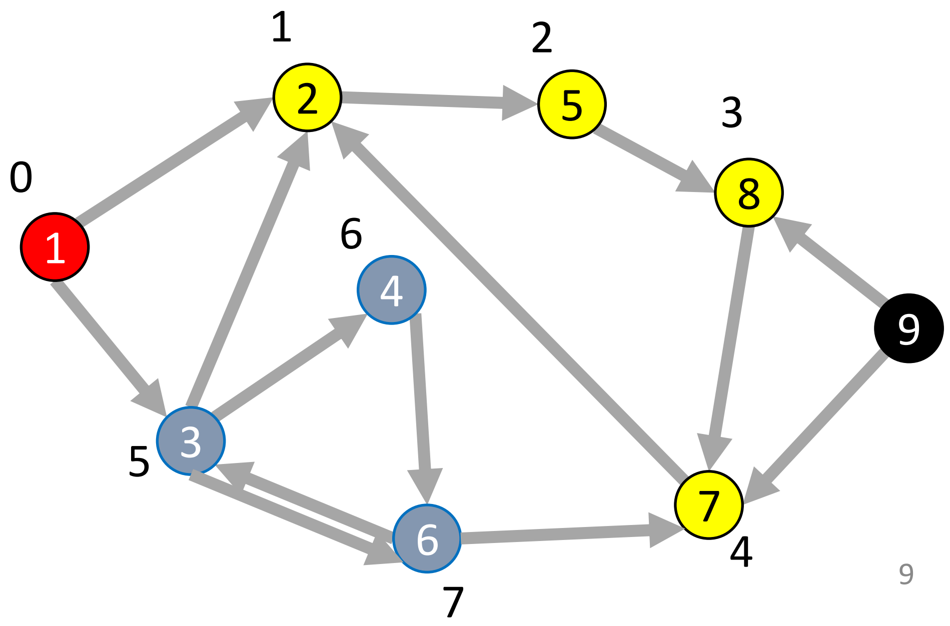
```
class BankAccount {
	…
	synchronized void withdraw(int amt) {…}
	synchronized void deposit(int amt) {…}
	void transferTo(int amt, BankAccount a) {
		synchronized(this){
			this.withdraw(amt);
		}
		synchronized(a){
			a.deposit(amt);
		}
	}
}
```

Option 3: First Get All Locks In A Fixed Order

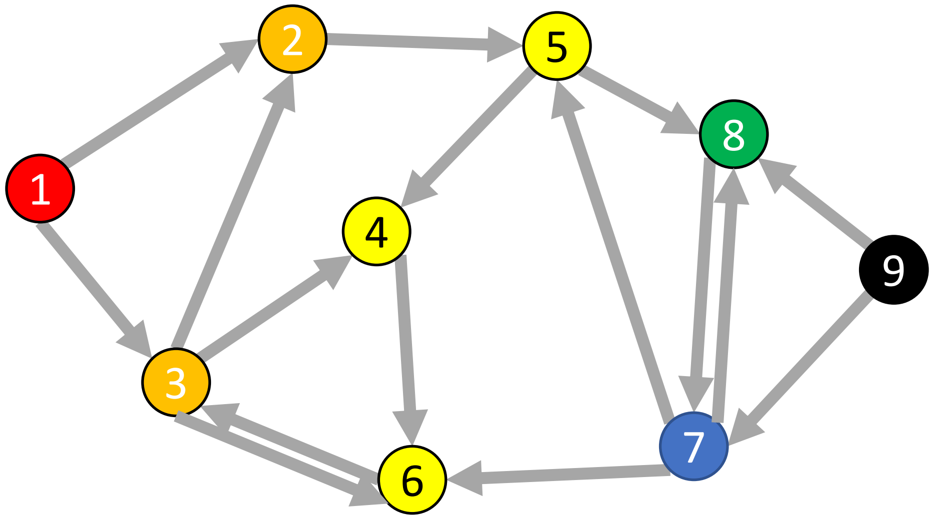
```
class BankAccount {
	…
	synchronized void withdraw(int amt) {…}
	synchronized void deposit(int amt) {…}
	void transferTo(int amt, BankAccount a) {
		if (this.acctNum < a.acctNum){
			synchronized(this){
				synchronized(a){
					this.withdraw(amt);
					a.deposit(amt);
		} } }
		else {
			synchronized(a){
				synchronized(this){
					this.withdraw(amt);
					a.deposit(amt);
		} } }
	}
}
```

Depth-First Search

- Input: a node s
- Behavior: Start with node s, visit one neighbor of s, then all nodes reachable from that neighbor of s, then another neighbor of s,…
- Output:
  - Does the graph have a cycle?
  - A **topological sort** of the graph.



DFS (non-recursive)

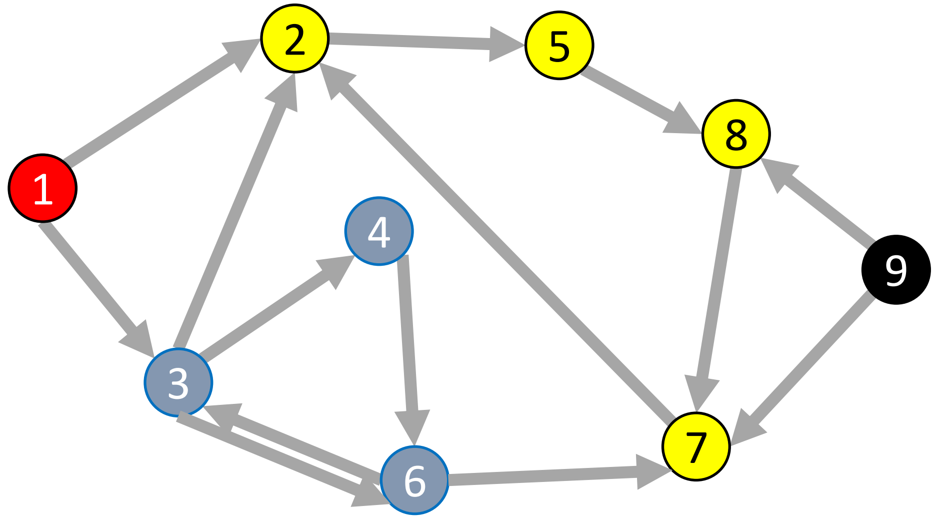


Running time: $\Theta(|V| + |E|)$

```
void dfs(graph, s){
        found = new Stack();
        found.pop(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.pop();
                for (v : neighbors(current)){
                        if (! v marked "visited"){
                                mark v as "visited";
                                found.push(v);
                        }
                }
        }
}
```

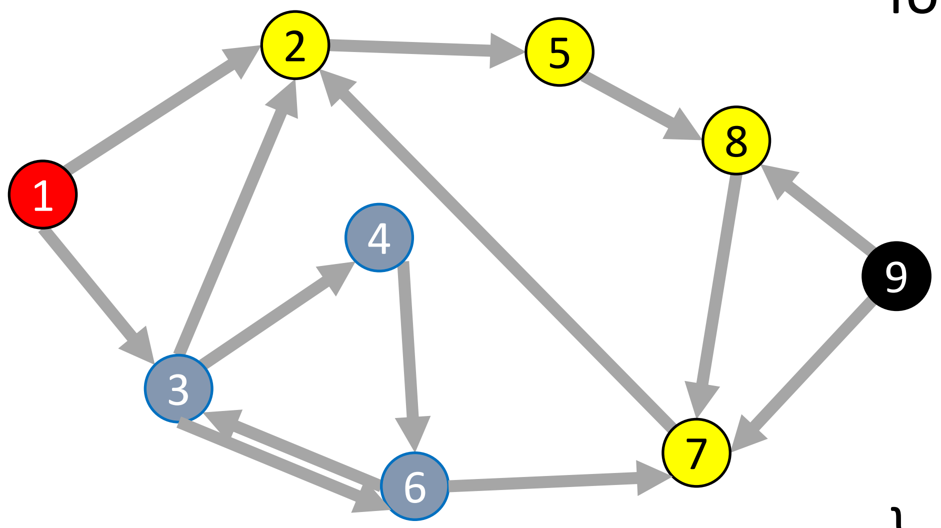
DFS Recursively (more common)

```
void dfs(graph, curr){
        mark curr as “visited”;
        for (v : neighbors(current)){
                if (! v marked “visited”){
                        dfs(graph, v);
                }
        }
        mark curr as “done”;
}
```



Cycle Detection

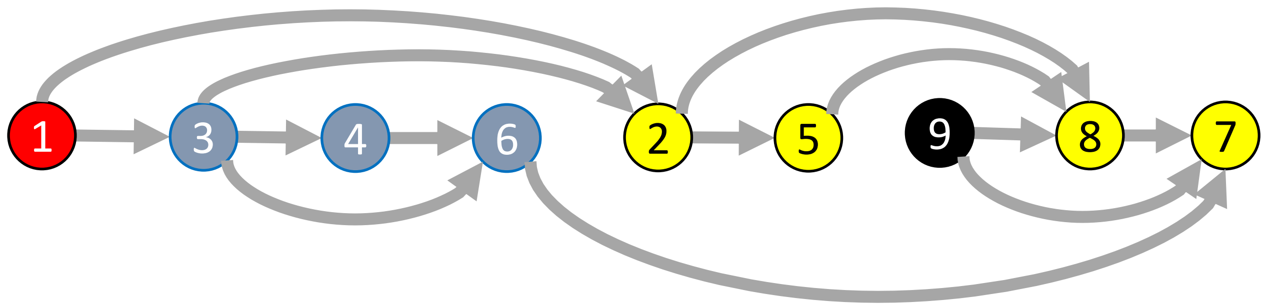
Idea: Look for a back edge!

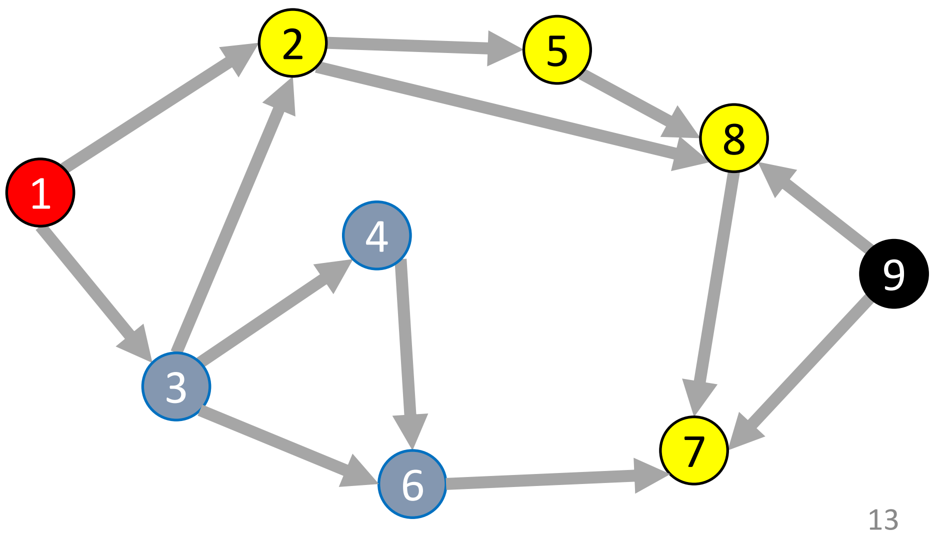



```
boolean hasCycle(graph, curr){
        mark curr as "visited";
        cycleFound = false;
        for (v : neighbors(current)){
                if (v marked "visited" && ! v marked "done"){
                        cycleFound=true;
                }
                if (! v marked "visited" && !cycleFound){
                        cycleFound = hasCycle(graph, v);
                }
        }
        mark curr as "done";
        return cycleFound;
}
```

Topological Sort

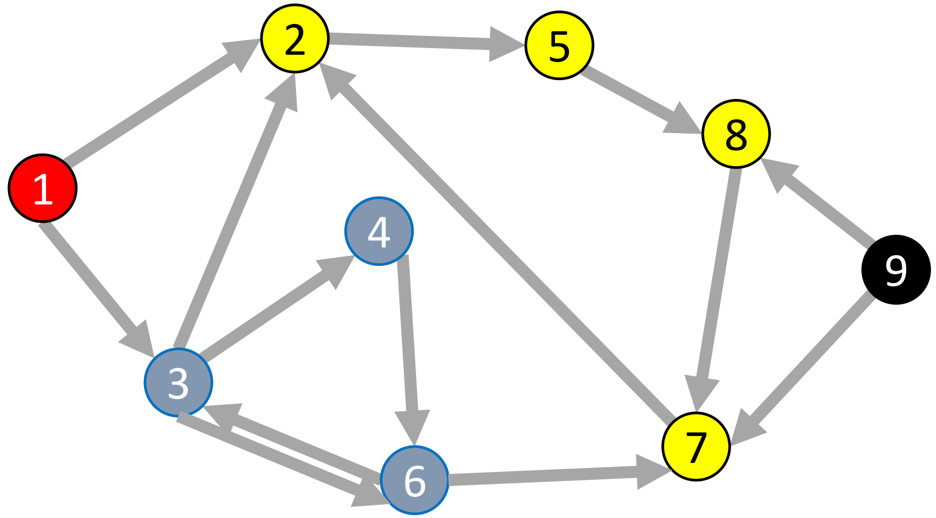
- A Topological Sort of a **directed acyclic graph** $\boldsymbol{G = (V, E)}$ is a permutation of V such that if $(u, v) \in E$ then u is before v in the permutation





DFS Recursively

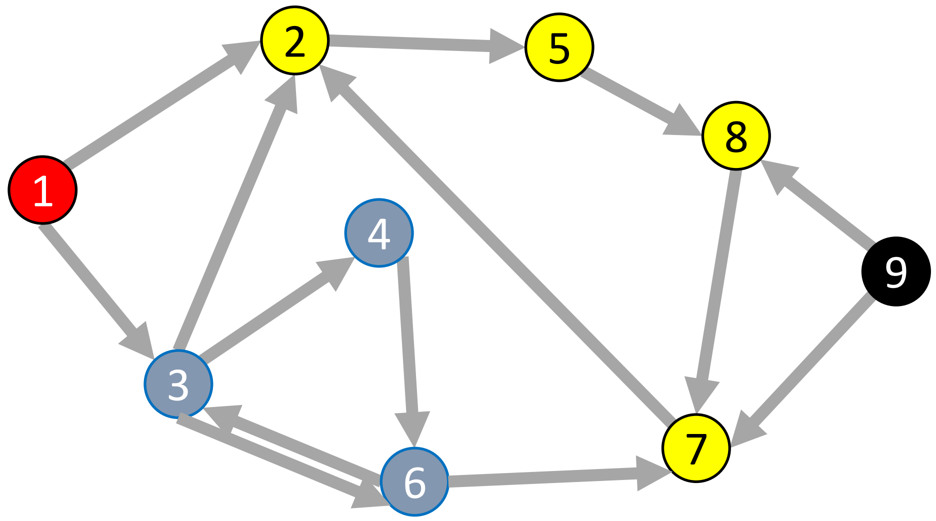
```
void dfs(graph, curr){
        mark curr as “visited”;
        for (v : neighbors(current)){
                if (! v marked “visited”){
                        dfs(graph, v);
                }
        }
        mark curr as “done”;
}
```



DFS Recursively

```
void dfs(graph, curr){
        mark curr as "visited";
        for (v : neighbors(current)){
                if (! v marked "visited"){
                        dfs(graph, v);
                }
        }
        mark curr as "done";
}
```

Idea: List in reverse order by "done" time



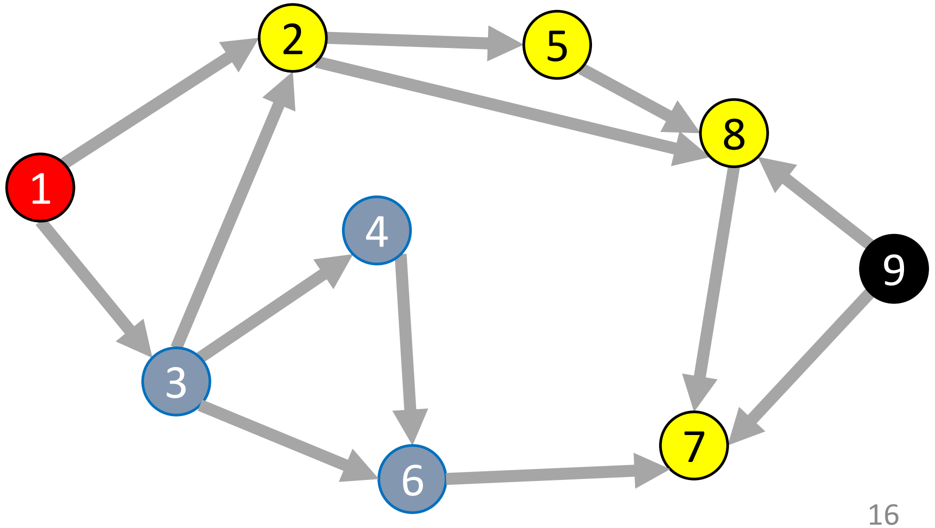
DFS: Topological sort

```
List topSort(graph){
        List<Nodes> done = new List<>();
        for (Node v : graph.vertices){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.reverse();
        return done;
}

void finishTime(graph, curr, finished){
        curr.visited = true;
        for (Node v : curr.neighbors){
                if (!v.visited){
                        finishTime(graph, v, finished);
                }
        }
        done.add(curr)
}
```

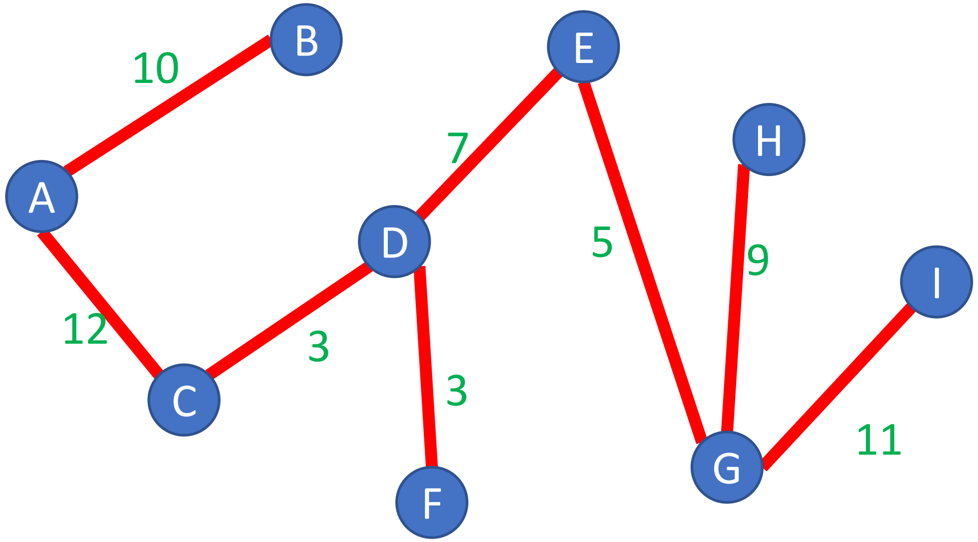
Idea: List in reverse order by "done" time

finished:



Definition: Tree

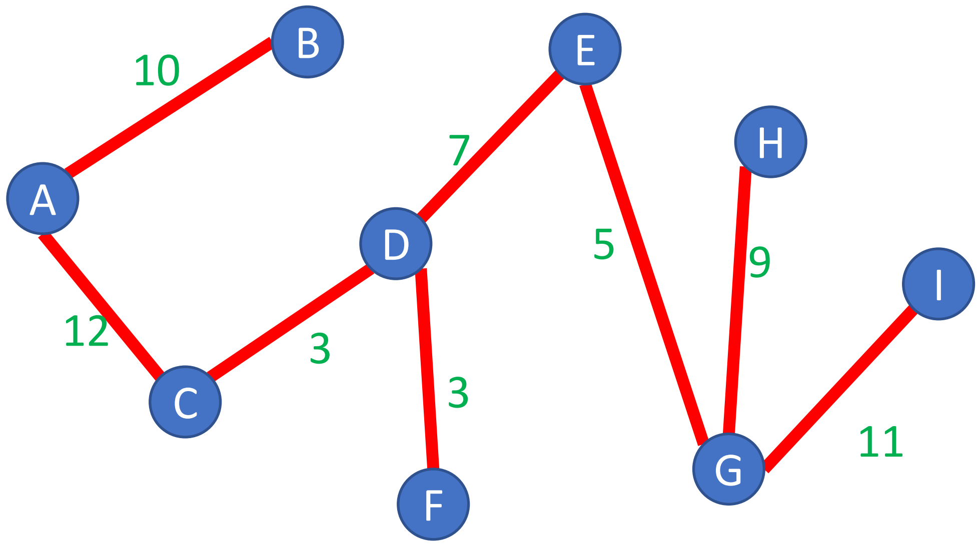
A connected graph with no cycles



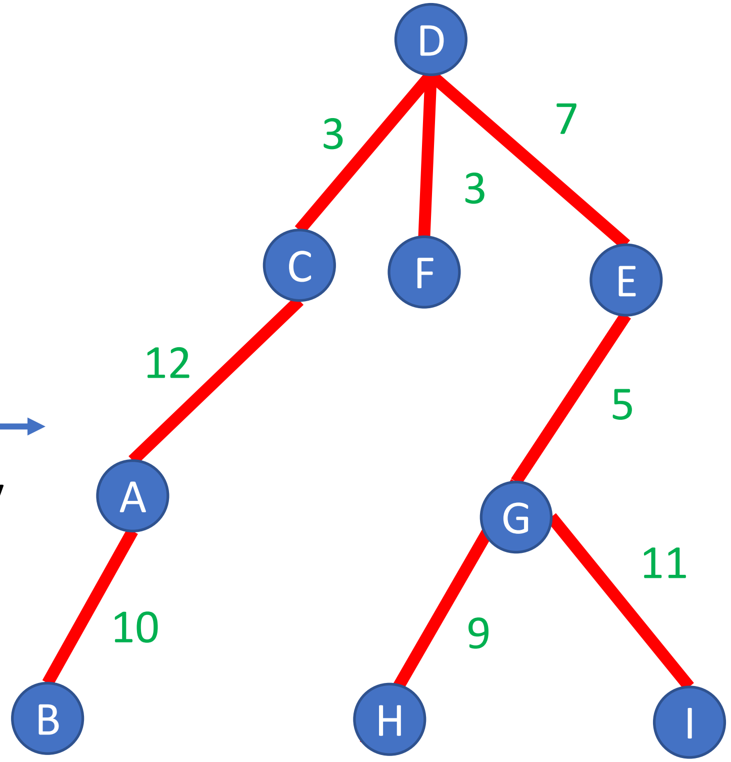
Note: A tree does not need a root, but they often do!

Definition: Tree

A connected graph with no cycles

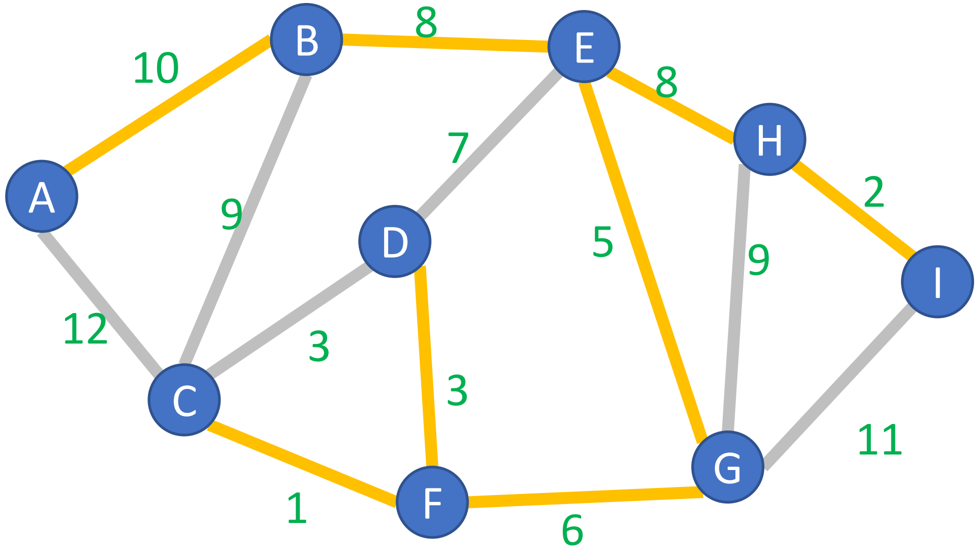


Pick some arbitrary root node and rearrange tree



Definition: Spanning Tree

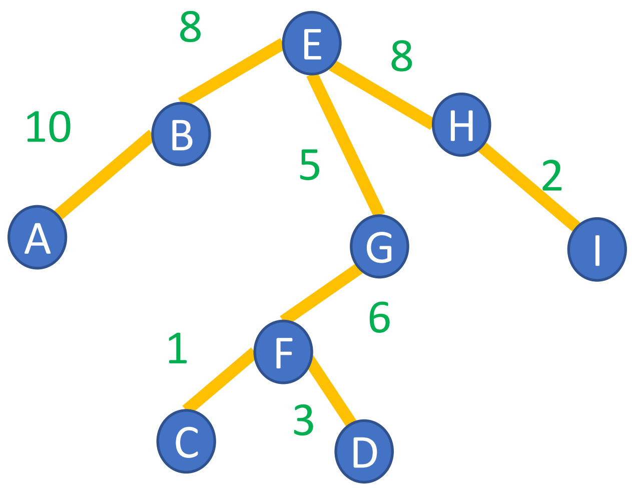
A Tree $T = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$



How many edges does T have?

$V - 1$

Pick some arbitrary root node and rearrange tree

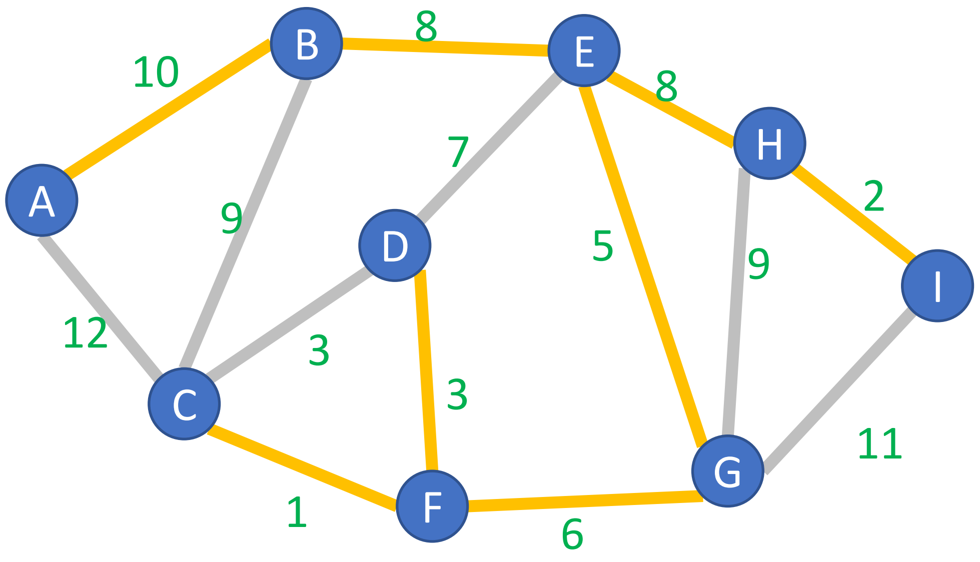


Any set of V-1 edges in the graph that doesn't have any cycles is guaranteed to be a spanning tree!

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

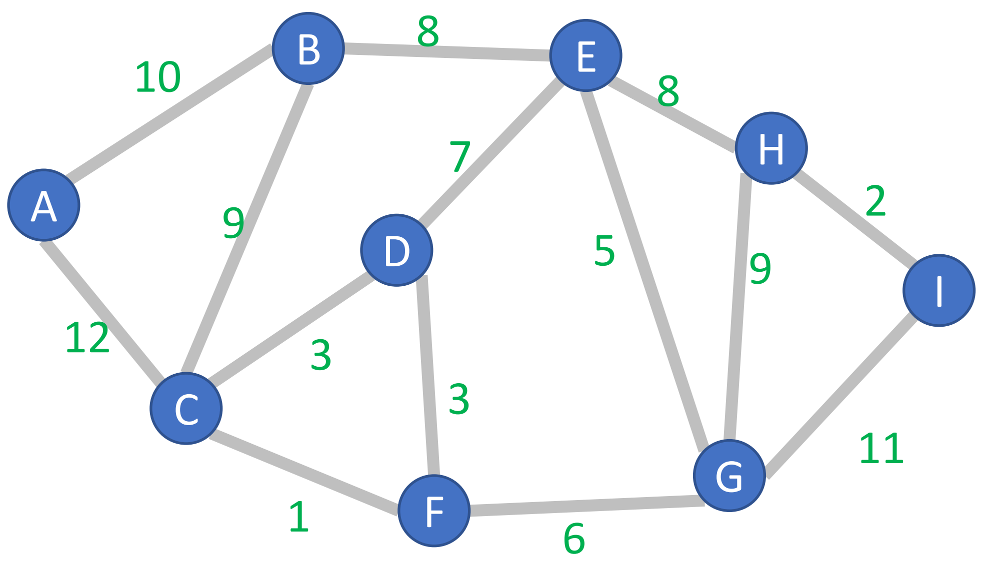


$$Cost(T) = \sum_{e \in E_T} w(e)$$

Kruskal’s Algorithm

Start with an empty tree A

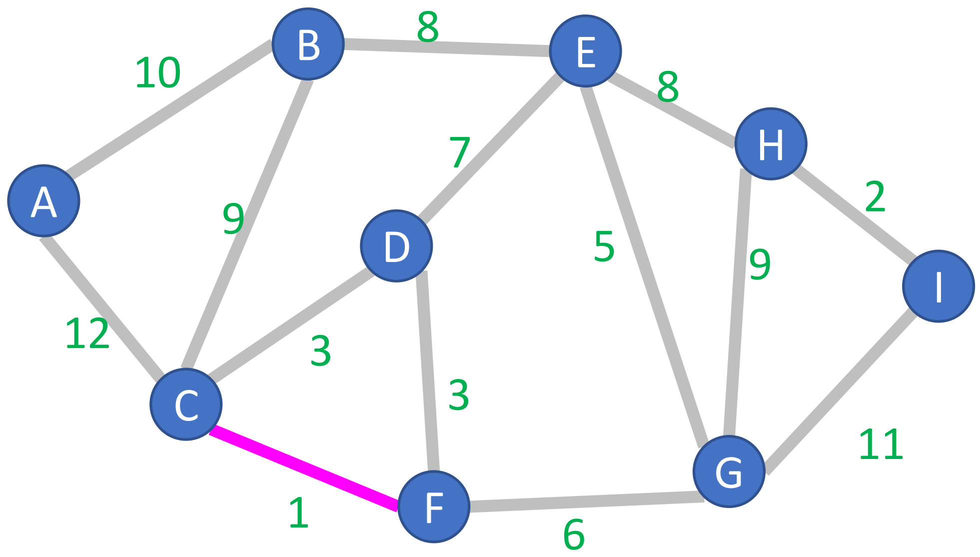
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

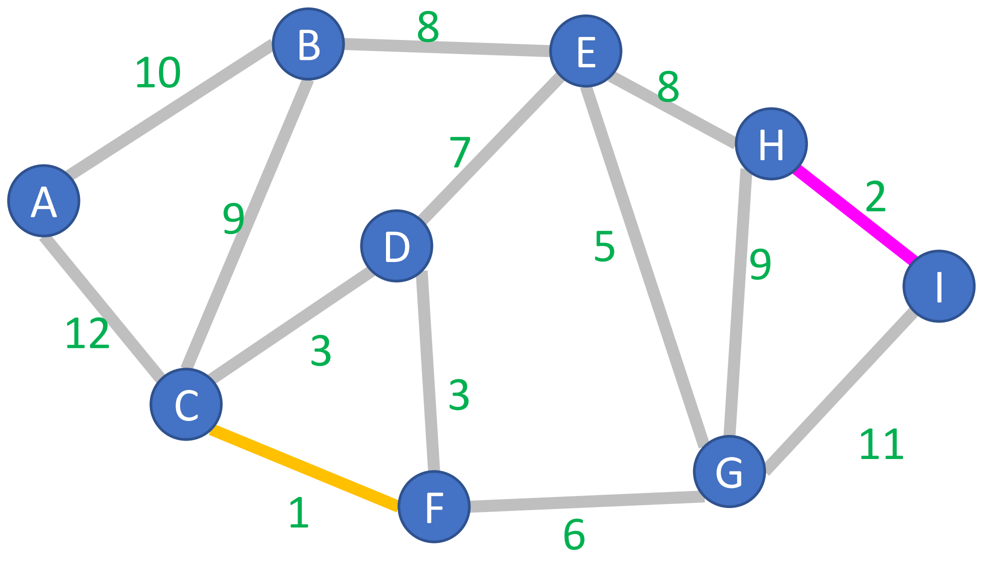
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

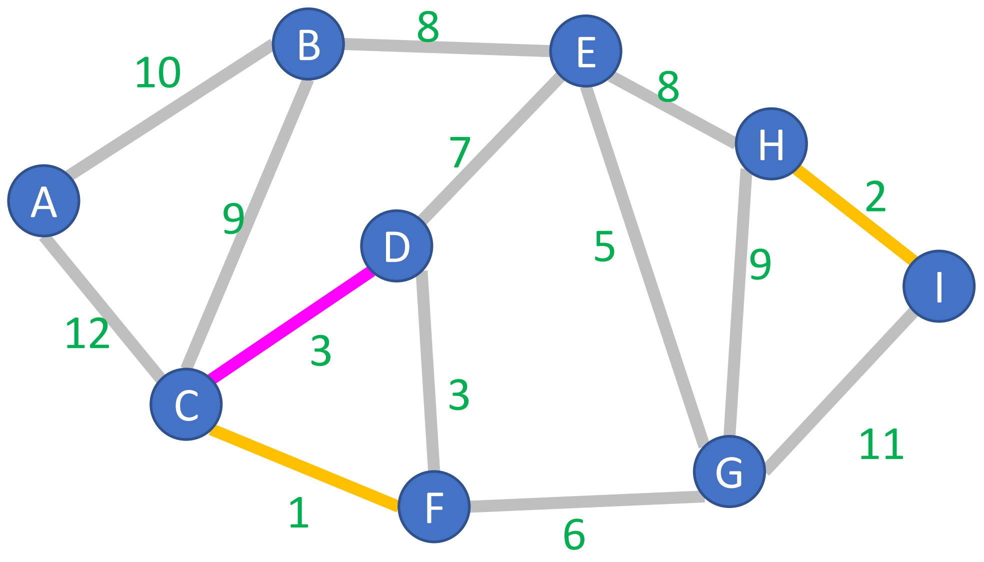
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

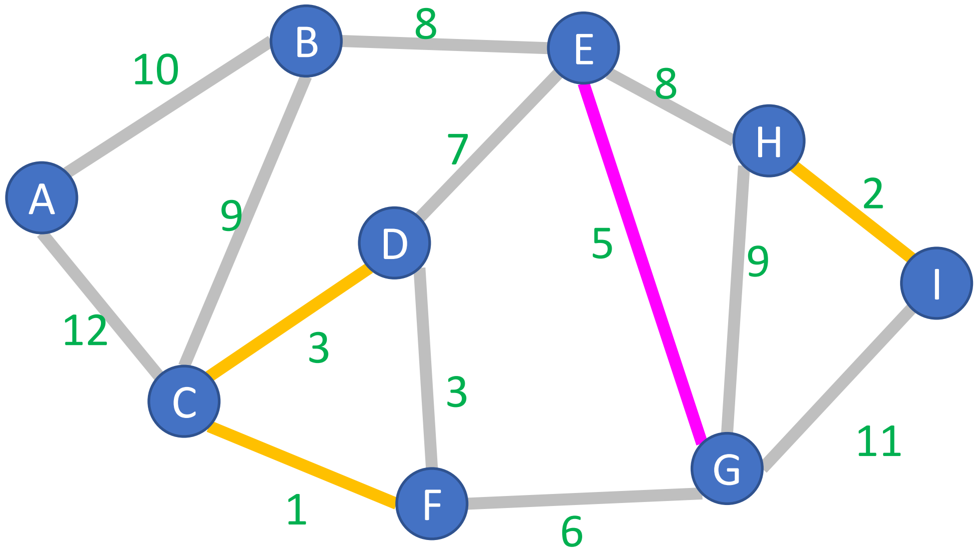
Add to A the lowest-weight edge that does not create a cycle



Kruskal’s Algorithm

Start with an empty tree A

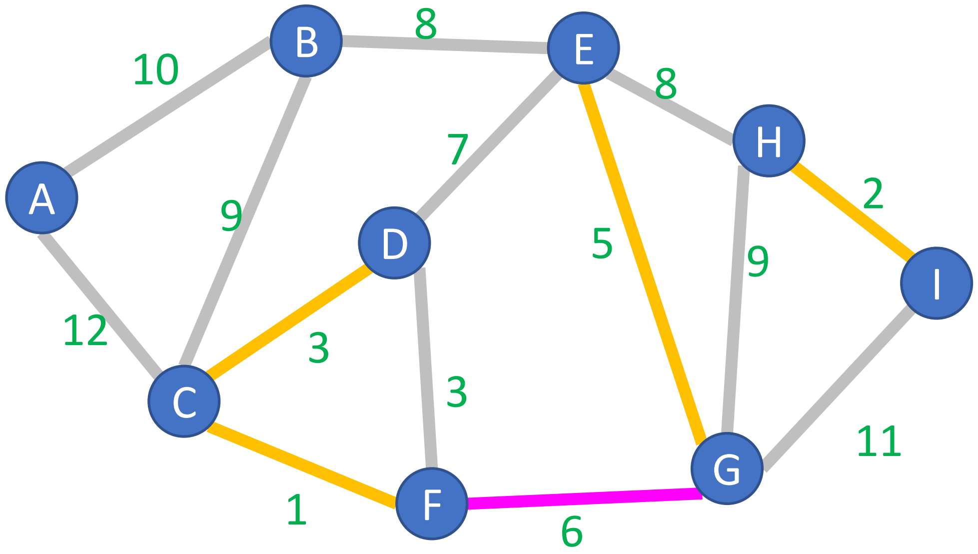
Add to A the lowest-weight edge that does not create a cycle



Kruskal’s Algorithm

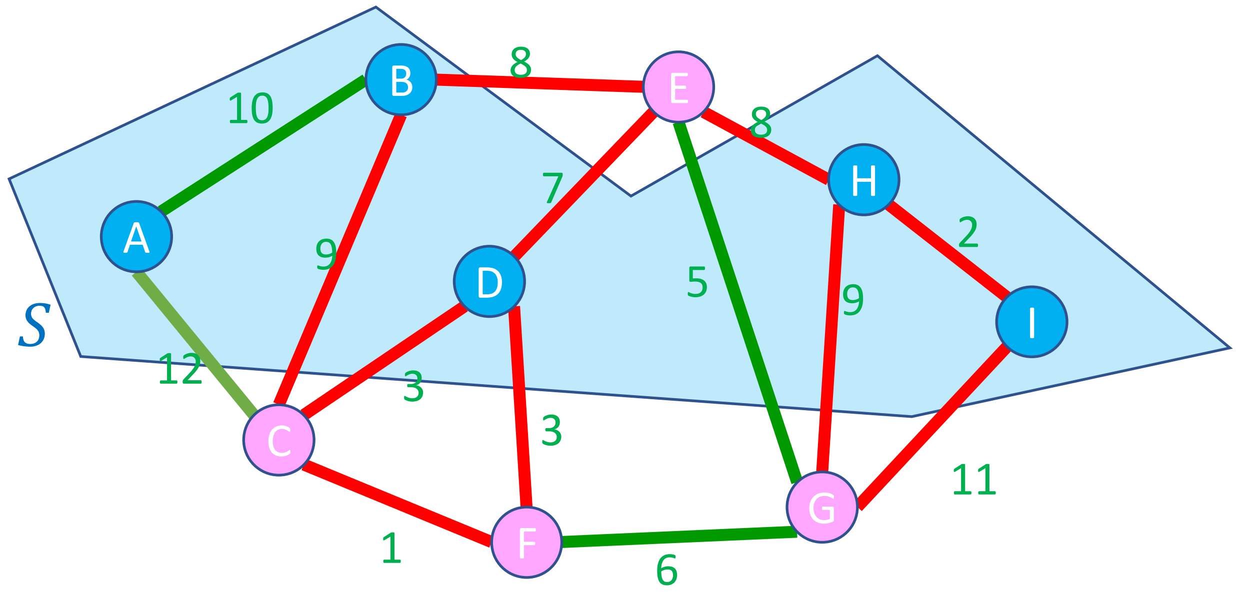
Start with an empty tree A

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

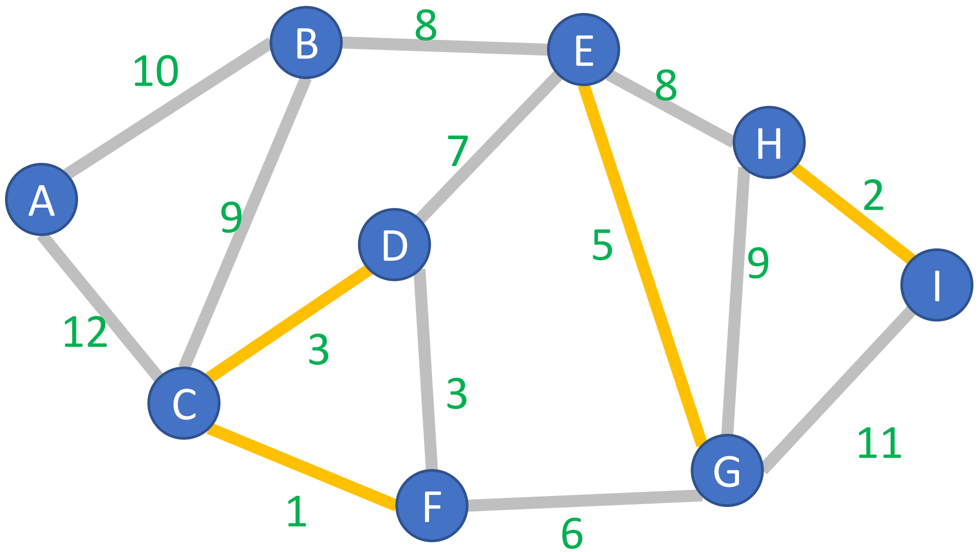
A set of edges R Respects a cut if no edges cross the cut e.g. $R = \{(A, B), (E, G), (F, G)\}$

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.

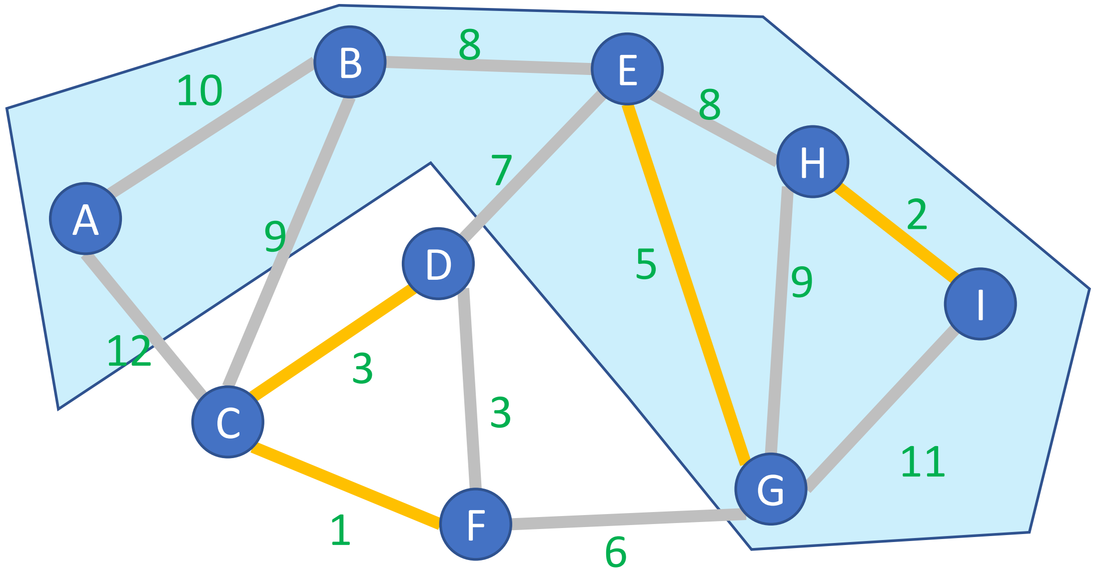
Cut Theorem

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.



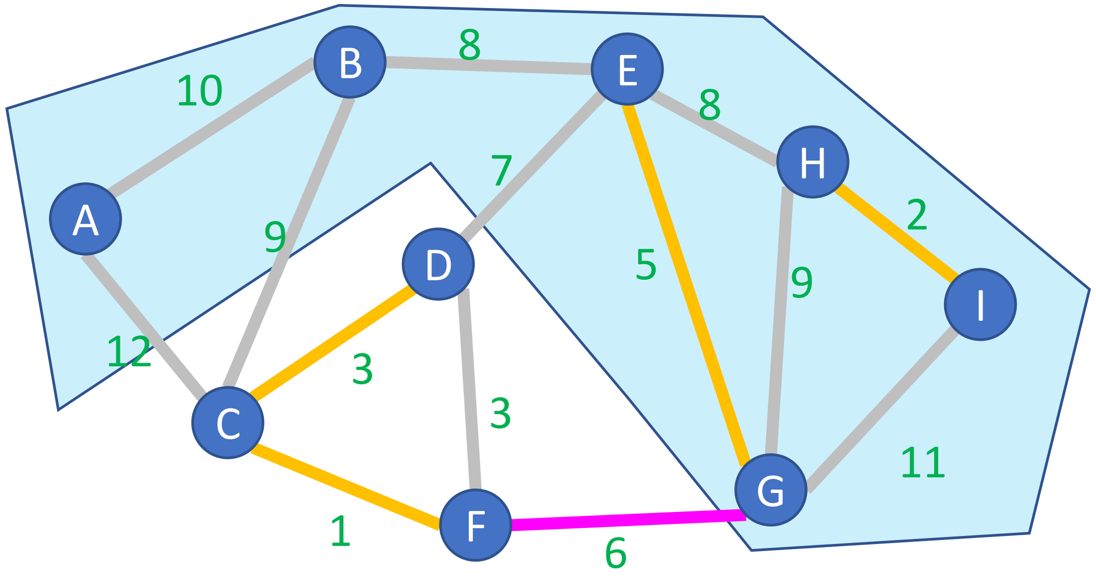
Cut Theorem

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.



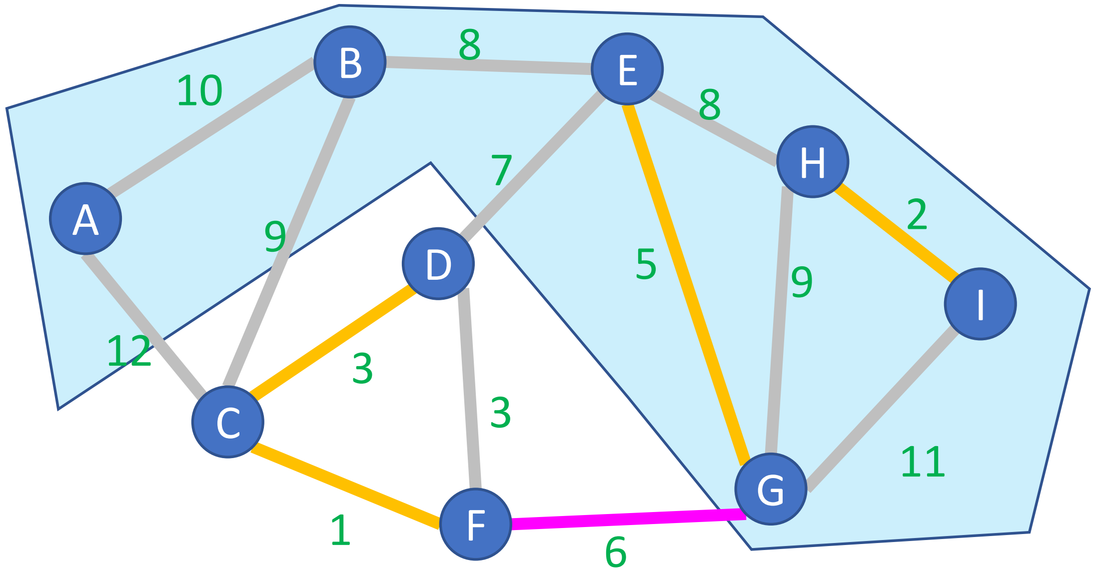
Cut Theorem

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

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.

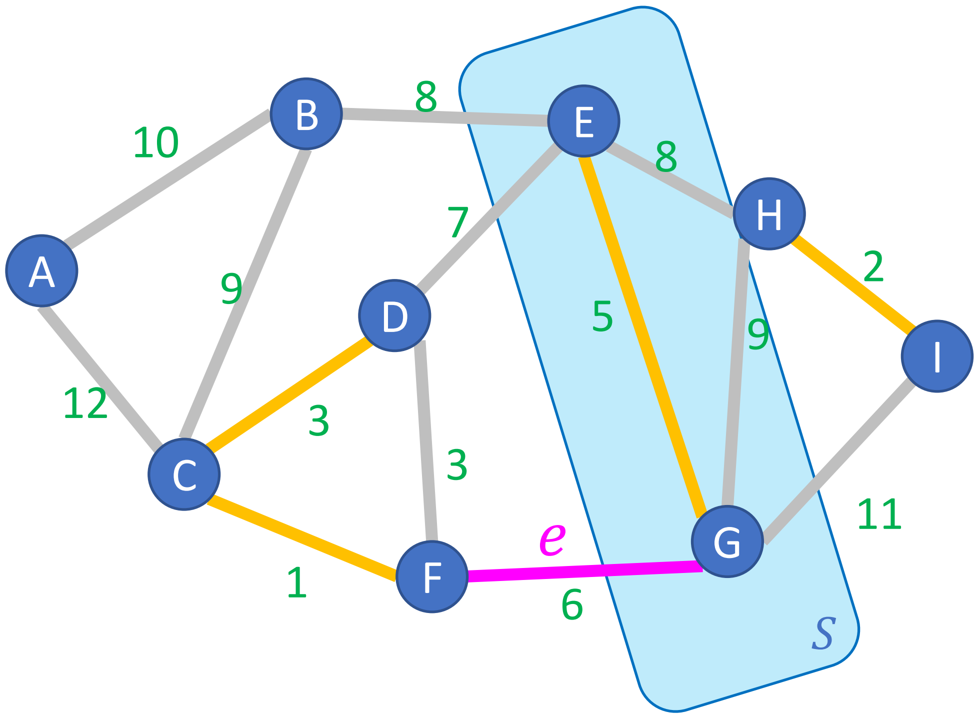


Proof of Kruskal's Algorithm

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

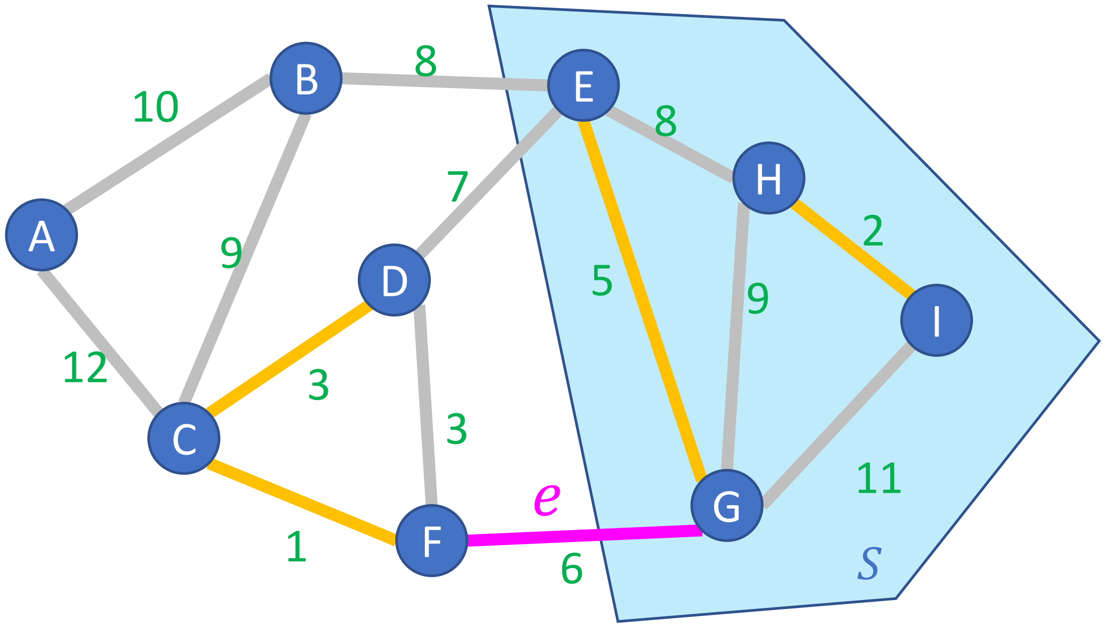
e is the minimum cost edge that crosses this cut, so by the Cut Theorem, Kruskal's is optimal!

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)

$O(E \log V)$

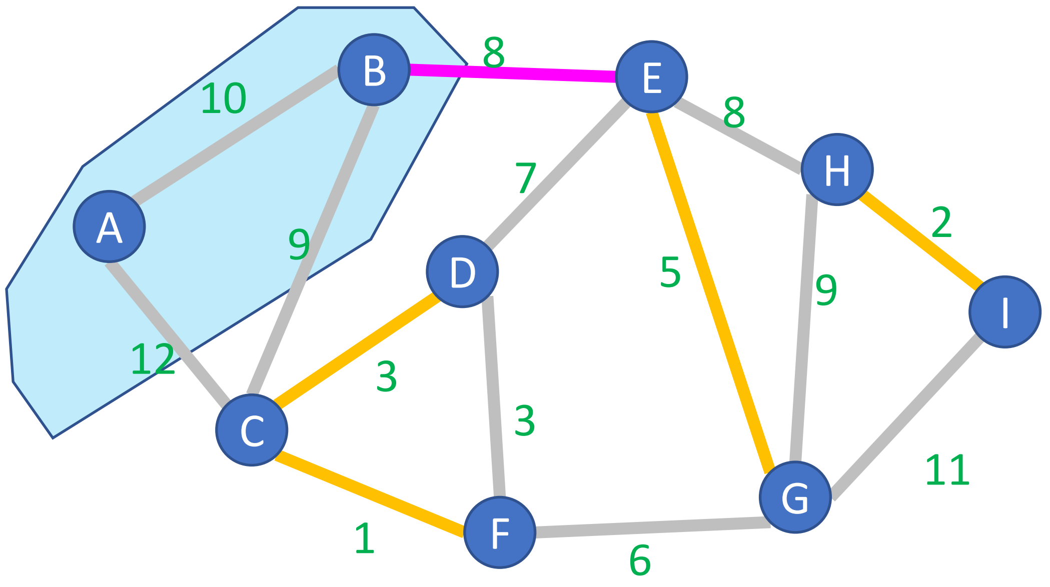
General MST Algorithm

Start with an empty tree A

Repeat $V-1$ times:

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

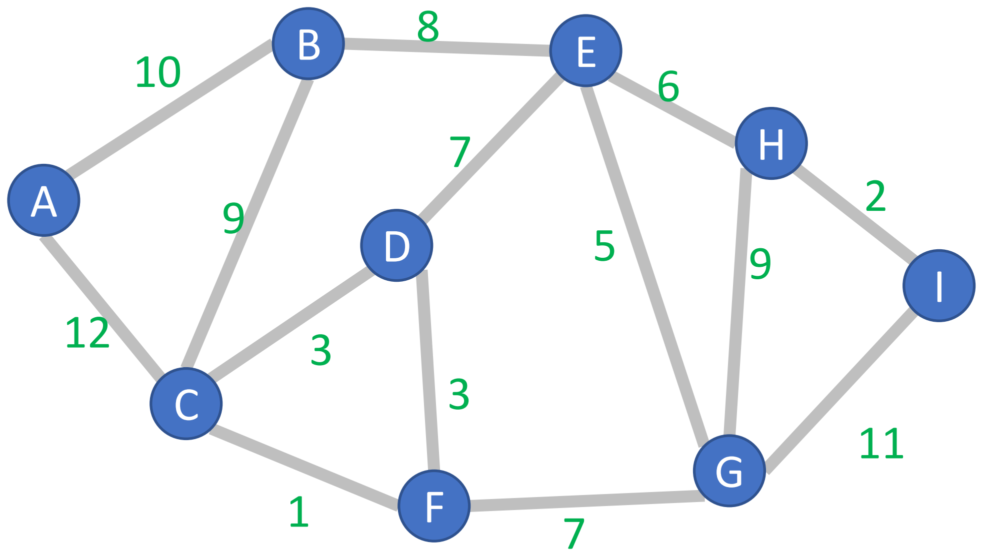
Add the min-weight edge which crosses $(S, V-S)$



Prim's Algorithm

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 endpoint of edges in A
e is the min-weight edge that grows the tree



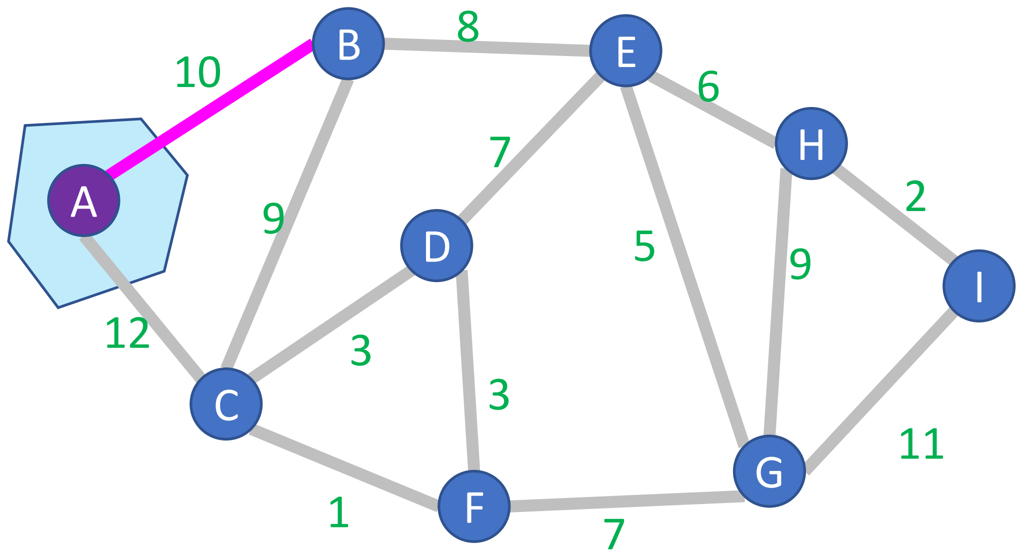
Prim's Algorithm

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



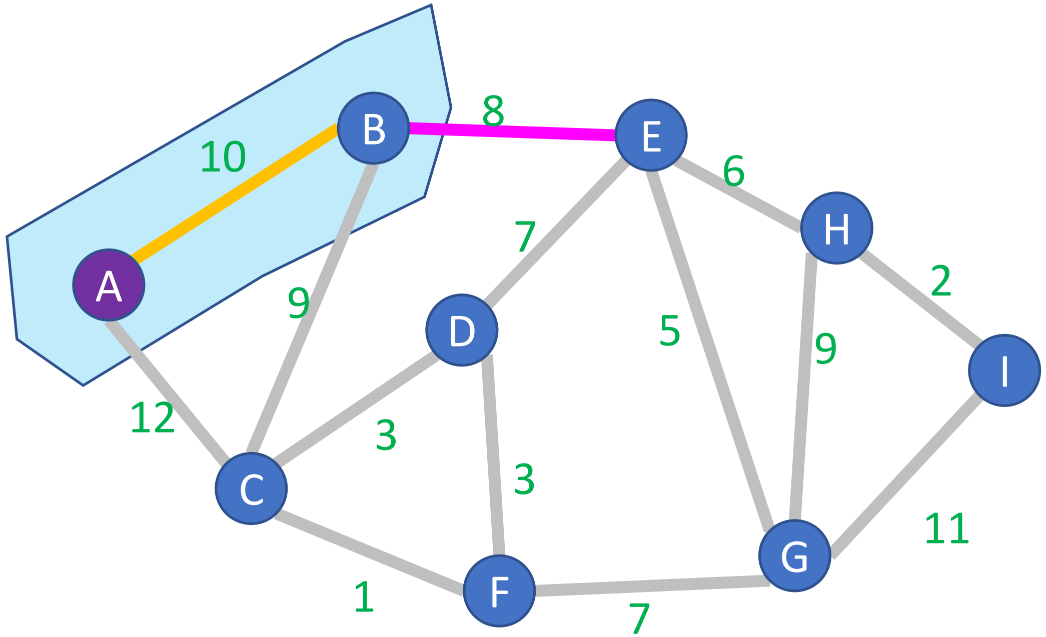
Prim's Algorithm

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



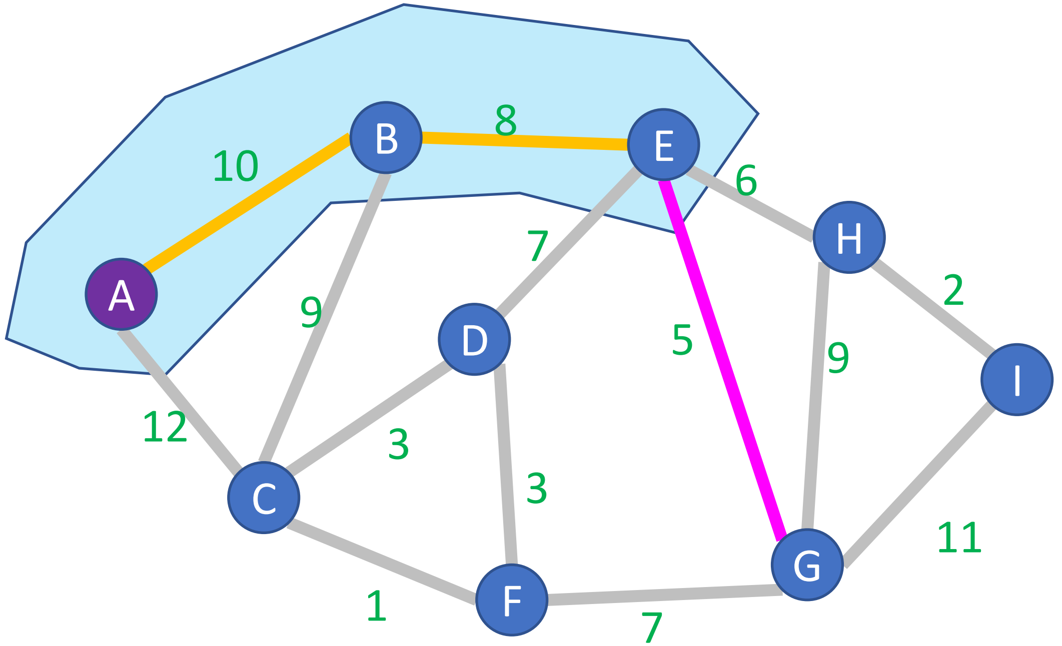
Prim's Algorithm

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



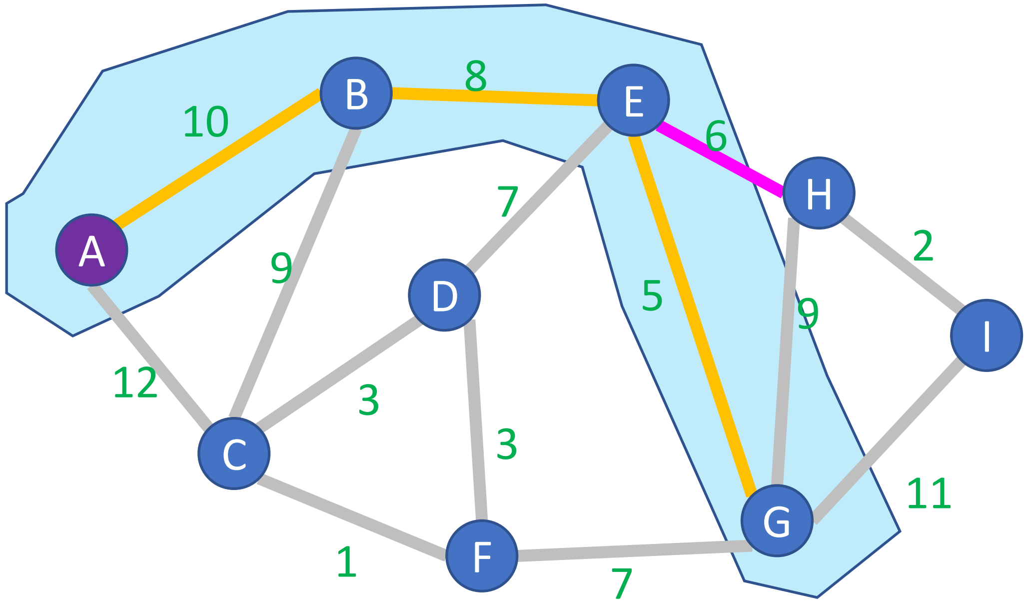
Prim's Algorithm

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

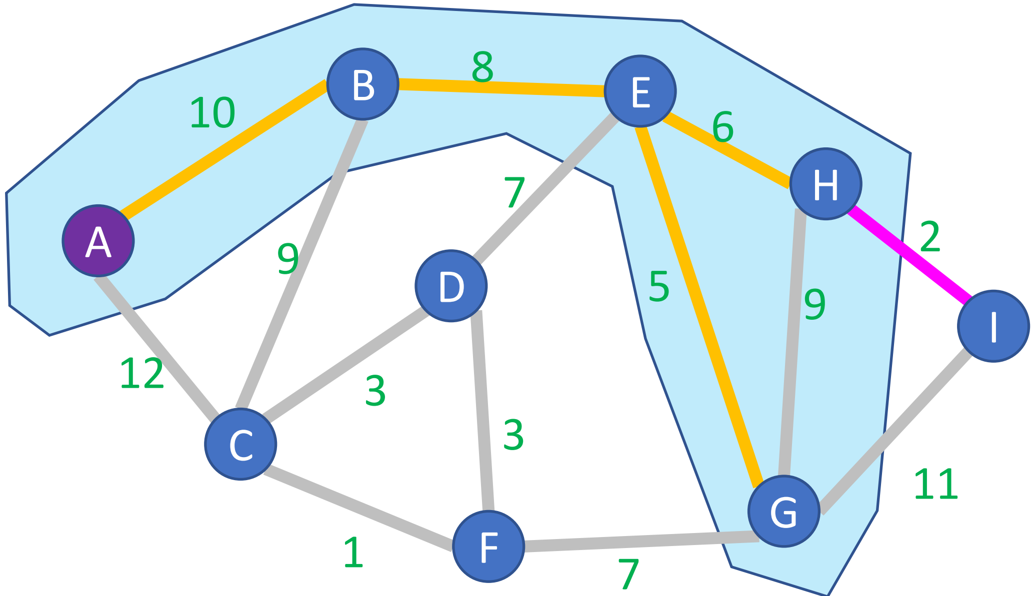
Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

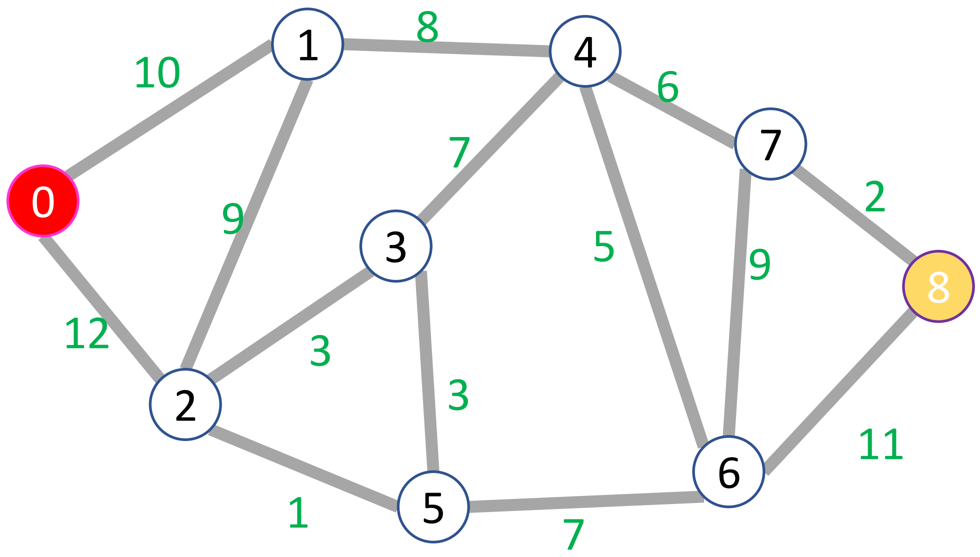
Add the min-weight edge which connects to node in A with a node not in A

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

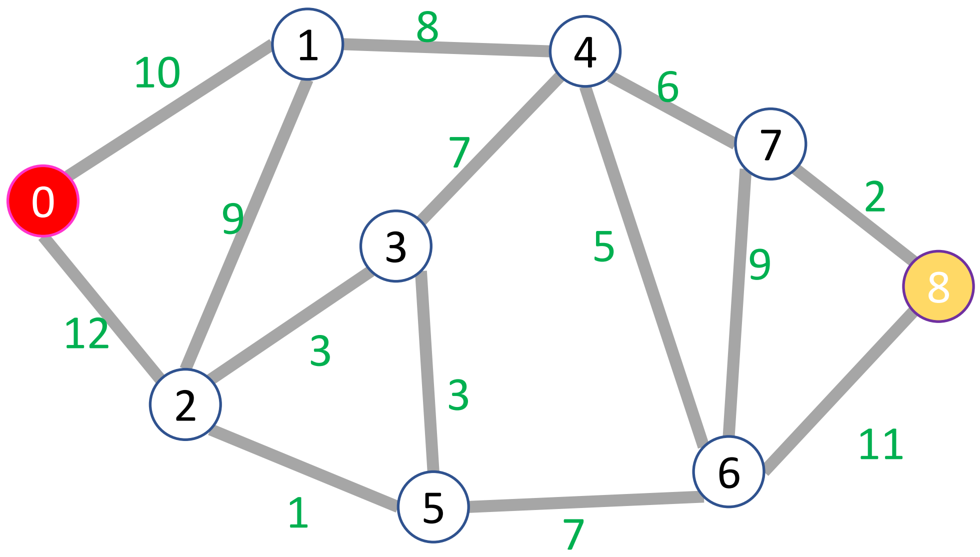


Dijkstra’s Algorithm

```
int dijkstras(graph, start, end){
        PQ = new minheap();
        PQ.insert(0, start);  // priority=0, value=start
        start.distance = 0;
        while (!PQ.isEmpty){
                current = PQ.extractmin();
                if (current.known){ continue;}
                current.known = true;
                for (neighbor : current.neighbors){
                        if (!neighbor.known){
                                new_dist = current.distance + weight(current,neighbor);
                                if(neighbor.dist != ∞){ PQ.insert(new_dist, neighbor);}
                                else if (new_dist < neighbor. distance){
                                        neighbor. distance = new_dist;
                                        PQ.decreaseKey(new_dist,neighbor); }
                        }
                }
        }
        return end.distance;
}
```



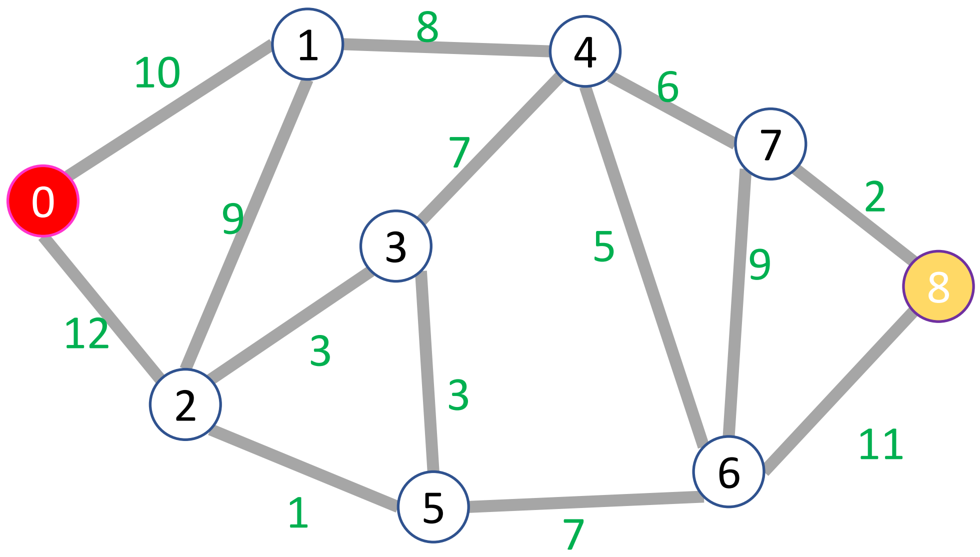
Prim’s Algorithm




```
int dijkstras(graph, start, end){
        PQ = new minheap();
        PQ.insert(0, start);  // priority=0, value=start
        start.distance = 0;
        while (!PQ.isEmpty){
                current = PQ.extractmin();
                if (current.known){ continue;}
                current.known = true;
                for (neighbor : current.neighbors){
                        if (!neighbor.known){
                                new_dist = weight(current,neighbor);
                                if(neighbor.dist != ∞){ PQ.insert(new_dist, neighbor);}
                                else if (new_dist < neighbor. distance){
                                        neighbor. distance = new_dist;
                                        PQ.decreaseKey(new_dist,neighbor); }
                        }
                }
        }
        return end.distance;
}
```

Dijkstra's Algorithm

```
int dijkstras(graph, start, end){
        PQ = new minheap();
        PQ.insert(0, start);  // priority=0, value=start
        start.distance = 0;
        while (!PQ.isEmpty){
                current = PQ.extractmin();
                if (current.known){ continue;}
                current.known = true;
                for (neighbor : current.neighbors){
                        if (!neighbor.known){
                                new_dist = current.distance + weight(current,neighbor);
                                if(neighbor.dist != ∞){ PQ.insert(new_dist, neighbor);}
                                else if (new_dist < neighbor. distance){
                                        neighbor. distance = new_dist;
                                        PQ.decreaseKey(new_dist,neighbor); }
                        }
                }
        }
        return end.distance;
}
```



Prim’s Algorithm

```
int dijkstras(graph, start, end){
        PQ = new minheap();
        PQ.insert(0, start);  // priority=0, value=start
        start.distance = 0;
        while (!PQ.isEmpty){
                current = PQ.extractmin();
                if (current.known){ continue;}
                current.known = true;
                for (neighbor : current.neighbors){
                        if (!neighbor.known){
                                new_dist = weight(current,neighbor);
                                if(neighbor.dist != ∞){ PQ.insert(new_dist, neighbor);}
                                else if (new_dist < neighbor. distance){
                                        neighbor. distance = new_dist;
                                        PQ.decreaseKey(new_dist,neighbor); }
                        }
                }
        }
        return end.distance;
}
```