CSE 332 Autumn 2024
Lecture 25: Concurrency 3 & Minimum Spanning Trees

Nathan Brunelle

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

Deadlock

- Occurs when two or more threads are mutually blocking each other
- T1 is blocked by T2, which is blocked by T3, …, Tn is blocked by T1
  - A cycle of blocking

Bank Account

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

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);
```

Thread 1	Thread 2
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 simultaneously needed to complete a task, and different threads may obtain them in a different order
- Option 1: Address the number of locks
  - Have a coarser lock granularity
  - E.g. one lock for ALL bank accounts
- Option 2: Address simultaneous need
  - 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: Address order of acquisition
  - 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);
                } } }
        }
}
```

Parallel Code Conventional Wisdom

Memory Categories

All memory must fit one of three categories:

1. Thread Local: Each thread has its own copy
2. Shared and Immutable: There is just one copy, but nothing will ever write to it
3. Shared and Mutable: There is just one copy, it may change
   - Requires Synchronization!

Thread Local Memory

- Whenever possible, avoid sharing resources
- Dodges all race conditions, since no other threads can touch it!
  - No synchronization necessary! (Remember Ahmdal's law)
- Use whenever threads do not need to communicate using the resource
  - E.g., each thread should have its on Random object
- In most cases, most objects should be in this category

Immutable Objects

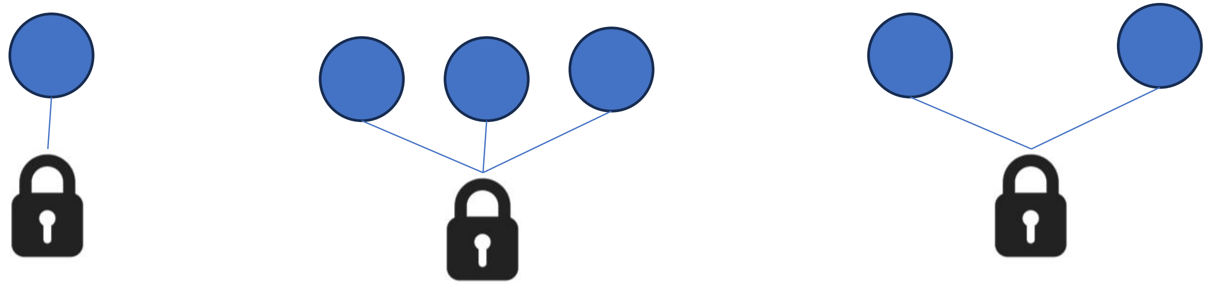
- Whenever possible, avoid changing objects
  - Make new objects instead
- Parallel reads are not data races
  - If an object is never written to, no synchronization necessary!
- Many programmers over-use mutation, minimize it

Shared and Mutable Objects

- For everything else, use locks
- Avoid all data races
  - Every read and write should be projected with a lock, even if it "seems safe"
  - Almost every Java/C program with a data race is wrong
- Even without data races, it still may be incorrect
  - Watch for bad interleavings as well!

Consistent Locking

- For each location needing synchronization, have a lock that is always held when reading or writing the location
- The same lock can (and often should) “guard” multiple fields/objects
  - Clearly document what each lock guards!
  - In Java, the lock should usually be the object itself (i.e. “this”)
- Have a mapping between memory locations and lock objects and stick to it!



Lock Granularity

- Coarse Grained: Fewer locks guarding more things each
  - One lock for an entire data structure
  - One lock shared by multiple objects (e.g. one lock for all bank accounts)
- Fine Grained: More locks guarding fewer things each
  - One lock per data structure location (e.g. array index)
  - One lock per object or per field in one object (e.g. one lock for each account)
- Note: there’s really a continuum between them…

Example: Separate Chaining Hashtable

- Coarse-grained: One lock for the entire hashtable
- Fine-grained: One lock for each bucket
- Which supports more parallelism in insert and find?
- Which makes rehashing easier?
- What happens if you want to have a size field?

Tradeoffs

- Coarse-Grained Locking:
  - Simpler to implement and avoid race conditions
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
  - Much easier for operations that modify data-structure shape
- Fine-Grained Locking:
  - More simultaneous access (performance when coarse grained would lead to unnecessary blocking)
  - Can make multi-location operations more difficult: say, rotations in an AVL tree
- Guideline:
  - Start with coarse-grained, make finer only as necessary to improve performance

Similar But Separate Issue: Critical Section Granularity

- Coarse-grained
  - For every method that needs a lock, put the entire method body in a lock
- Fine-grained
  - Keep the lock only for the sections of code where it's necessary
- Guideline:
  - Try to structure code so that expensive operations (like I/O) can be done outside of your critical section
  - E.g., if you're trying to print all the values in a tree, maybe copy items into an array inside your critical section, then print the array's contents outside.

Atomicity

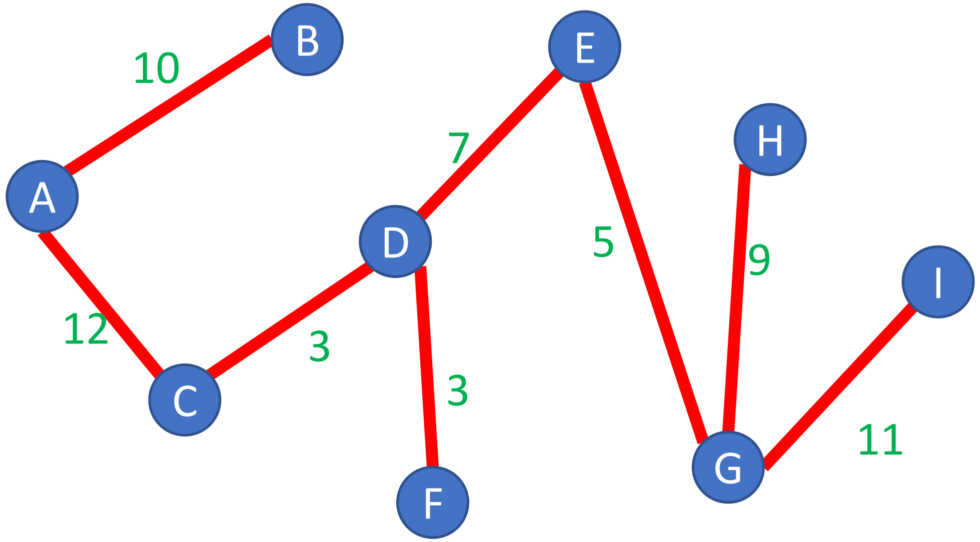
- Atomic: indivisible
- Atomic operation: one that should be thought of as a single step
- Some sequences of operations should behave as if they are one unit
  - Between two operations you may need to avoid exposing an intermediate state
  - Usually ADT operations should be atomic
    - You don’t want another thread trying to do an insert while another thread is rotating the AVL tree
- Think first in terms of what operations need to be atomic
  - Design critical sections and locking granularity based on these decisions

Use Pre-Tested Code

- Whenever possible, use built-in libraries!
- Other people have already invested tons of effort into making things both efficient and correct, use their work when you can!
  - Especially true for concurrent data structures
  - Use thread-safe data structures when available
    - E.g. Java as ConcurrentHashMap

Definition: Tree

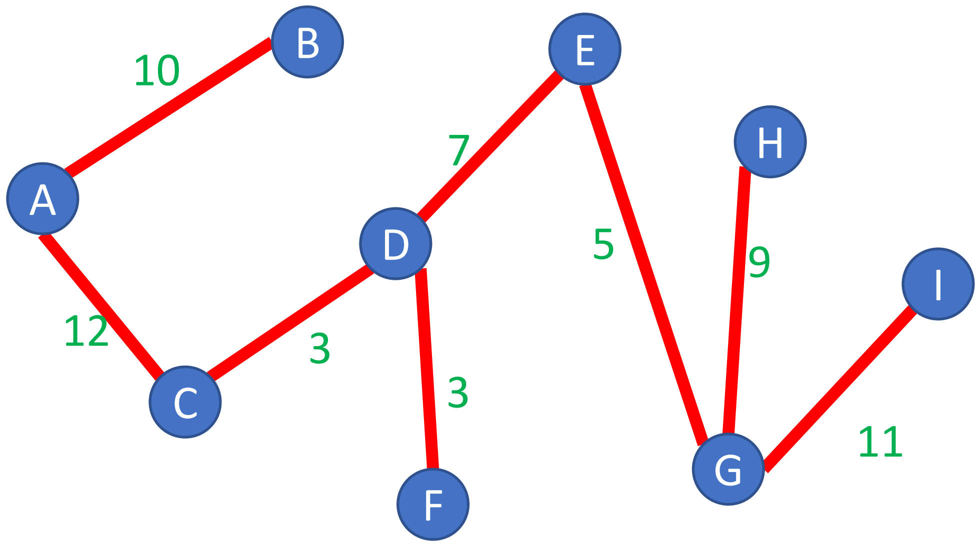
A connected graph with no cycles



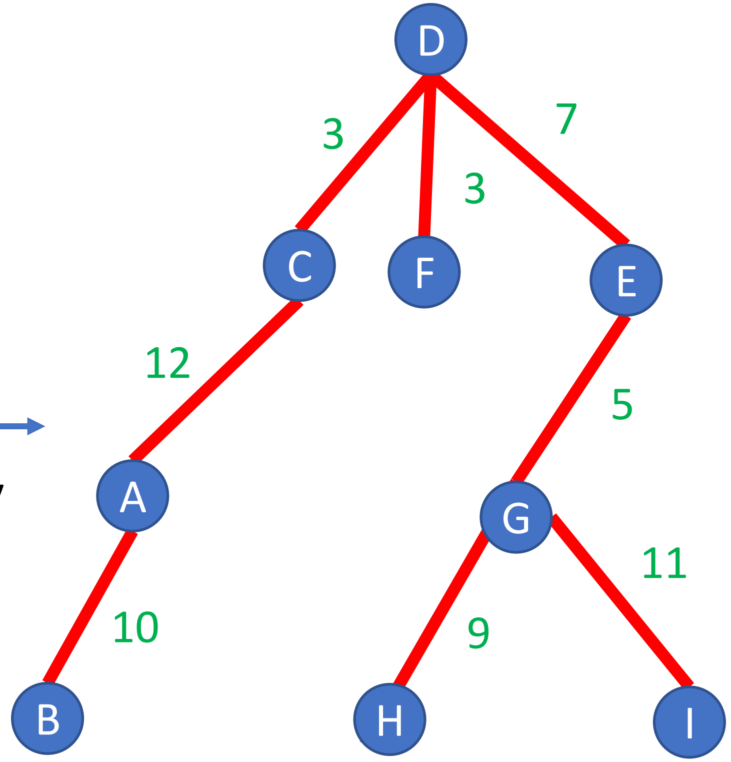
Note: A tree does not need a root, but they often do!

Definition: Tree

A connected graph with no cycles

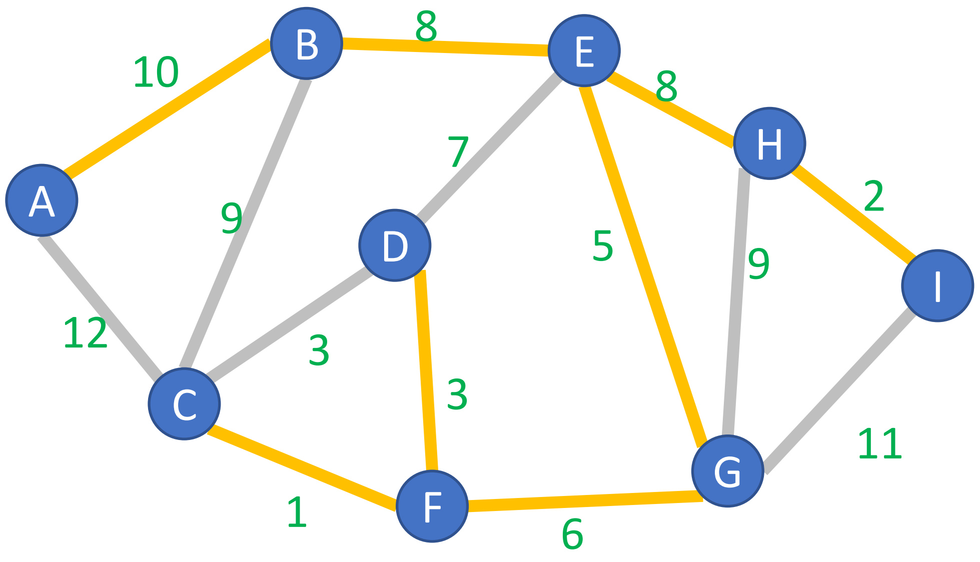


Pick some arbitrary root node and rearrange tree



Definition: Spanning Tree

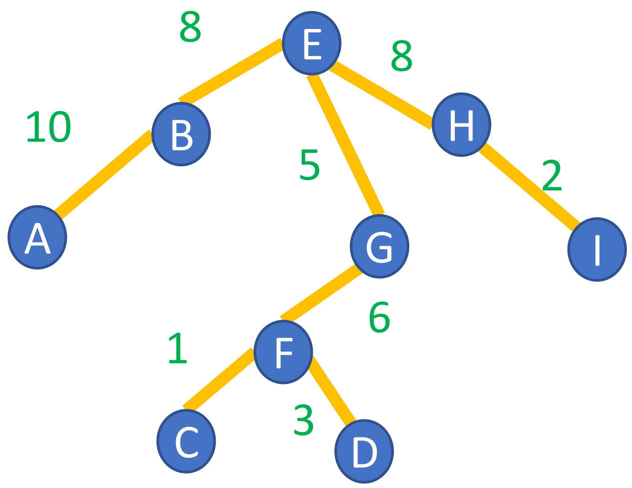
A Tree $T = (V_T, E_T)$ which connects ("spans") all the nodes in a graph $G = (V, E)$



Pick some arbitrary root node and rearrange tree

How many edges does T have?

$V - 1$

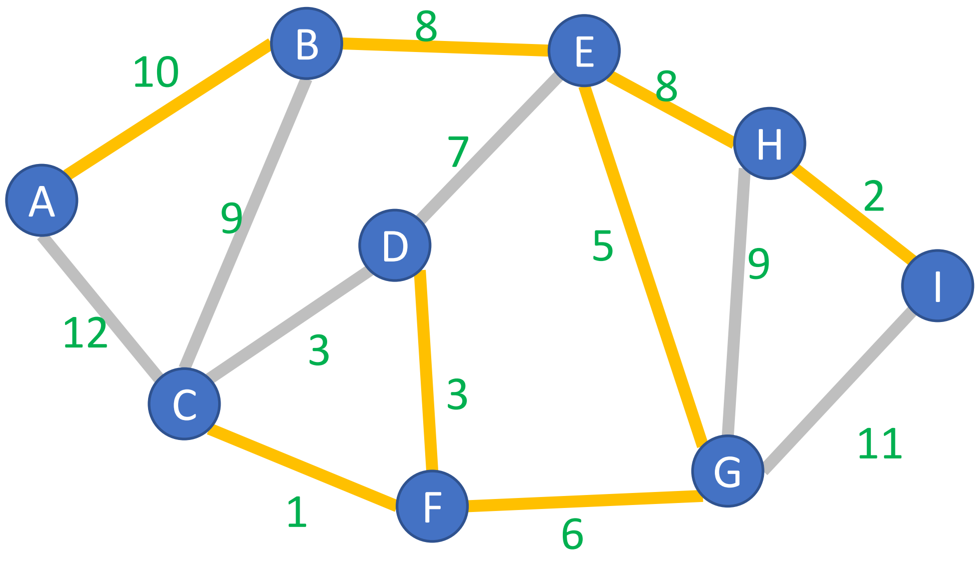


Any set of V-1 edges in the graph that doesn't have any cycles is guaranteed to be a spanning tree!

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



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

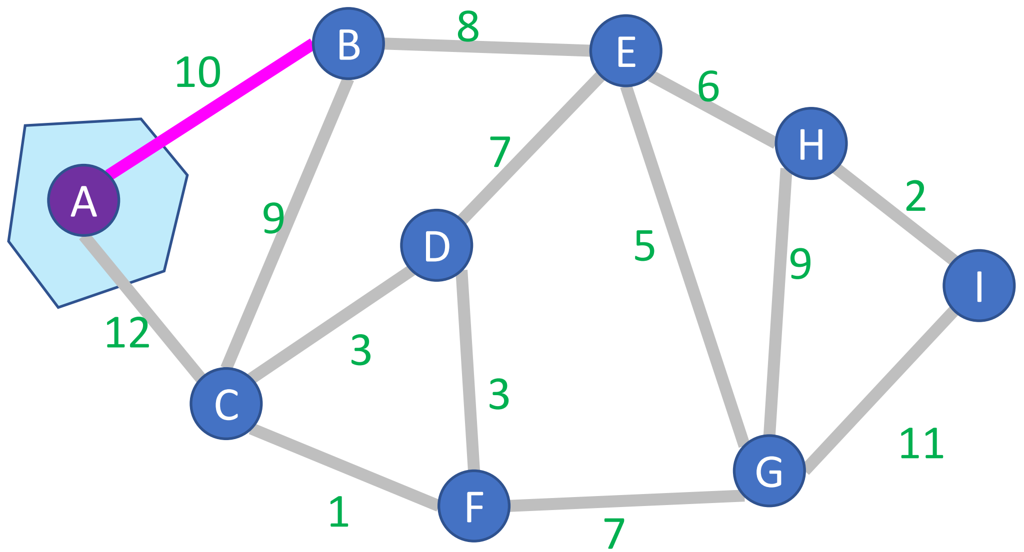
Prim's Algorithm

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



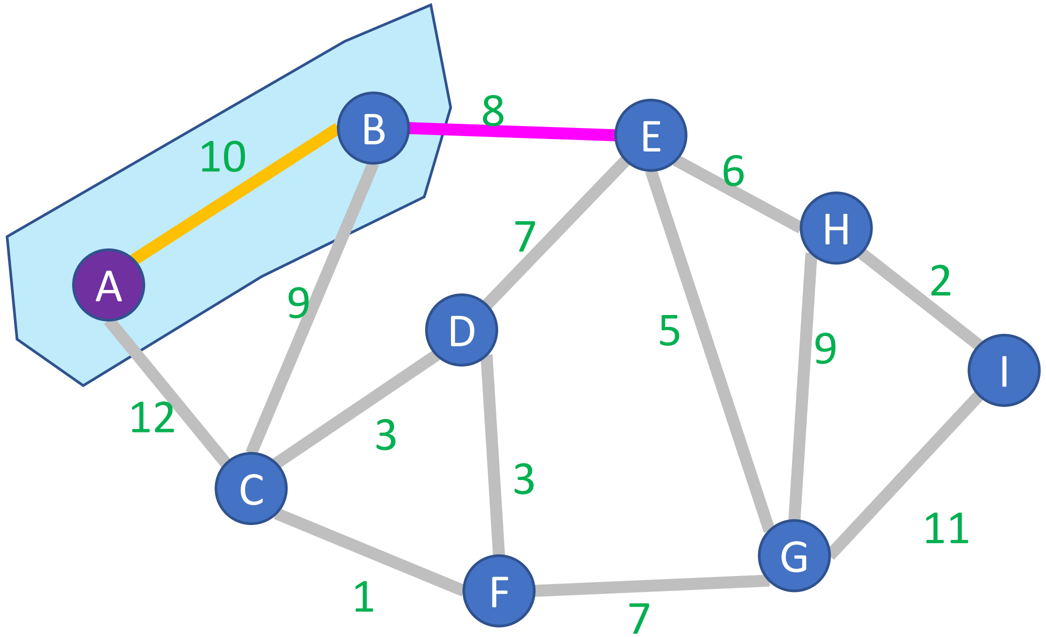
Prim's Algorithm

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



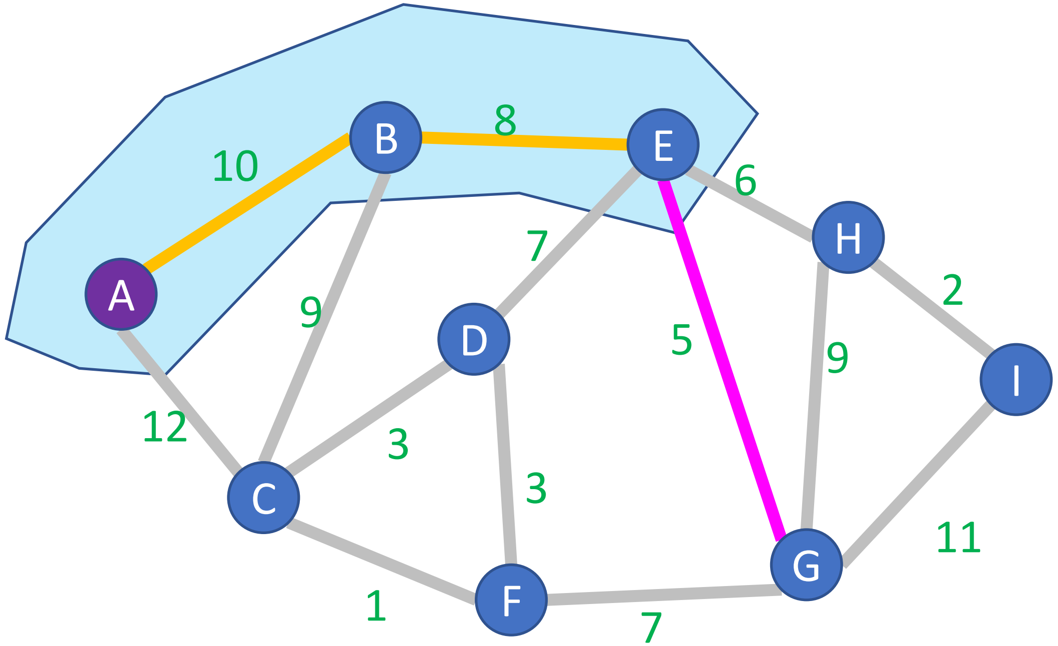
Prim's Algorithm

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



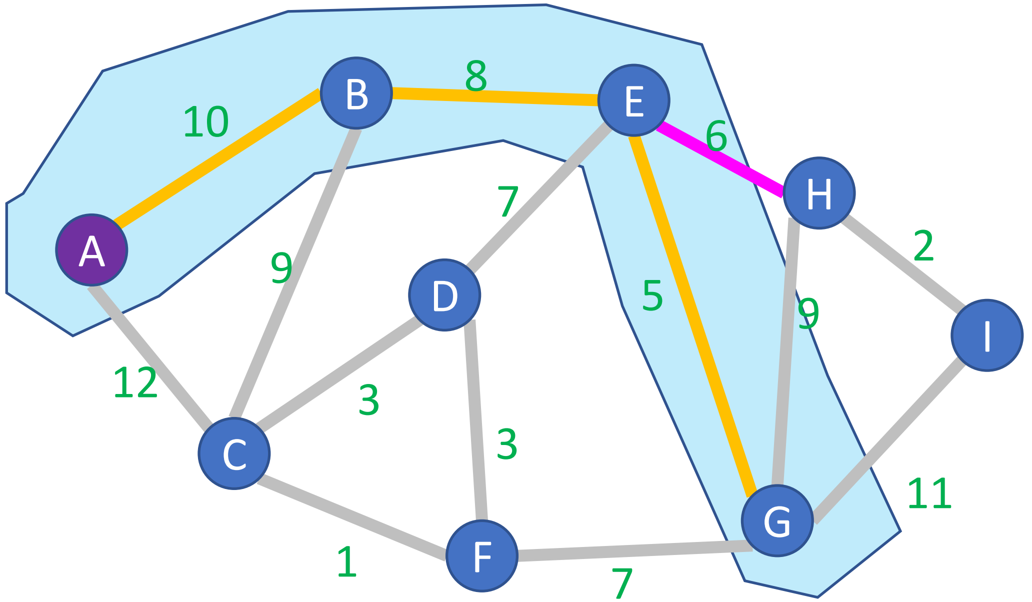
Prim's Algorithm

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

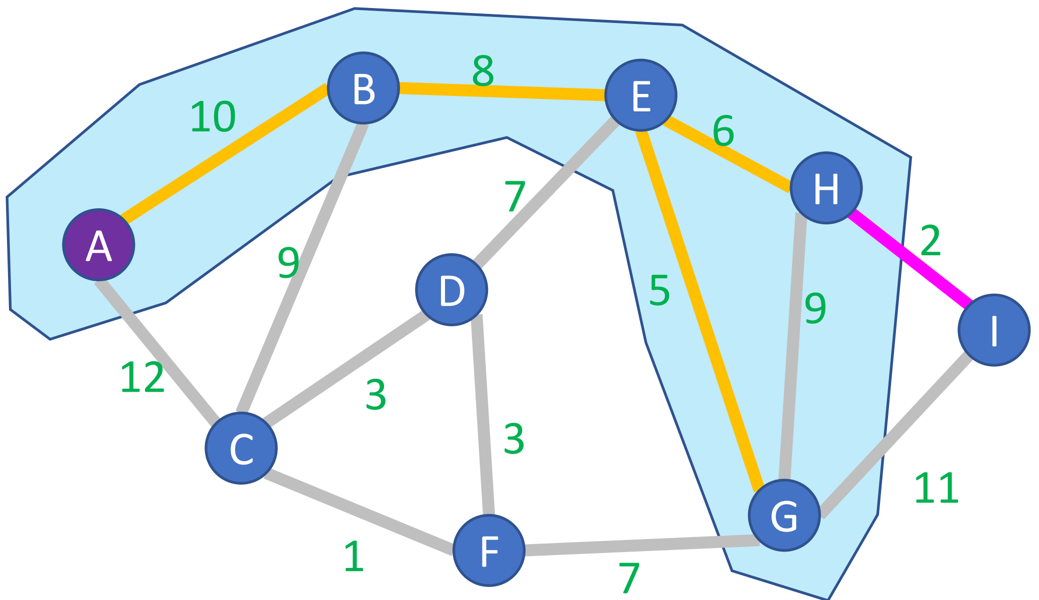
Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

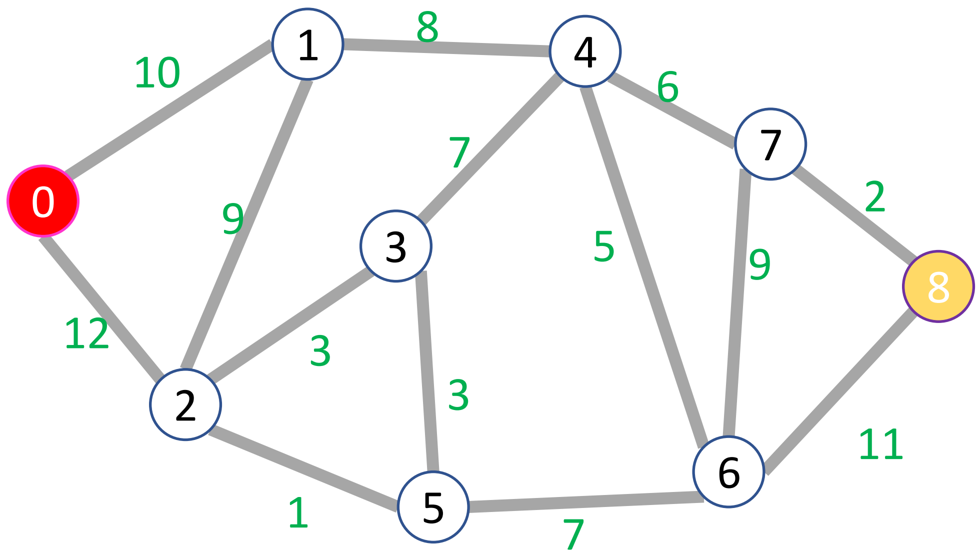
Add the min-weight edge which connects to node in A with a node not in A

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



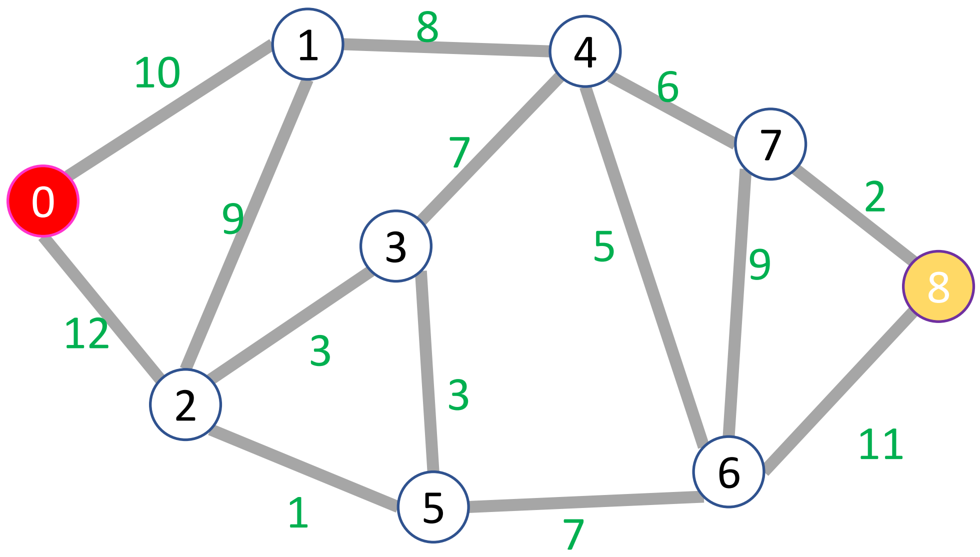
Dijkstra’s Algorithm

```
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;
}
```



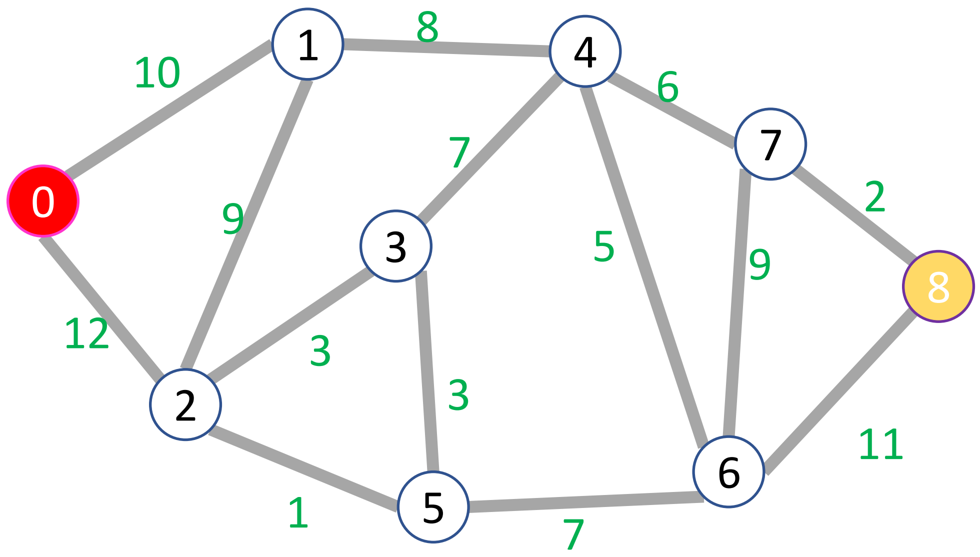
Prim's Algorithm

```
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;
}
```



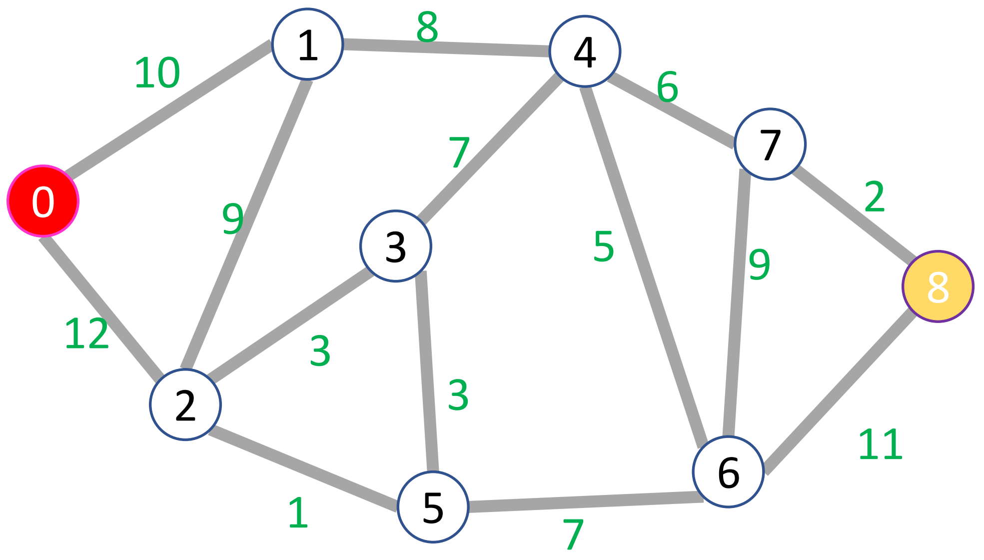
Dijkstra’s Algorithm

```
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;
}
```



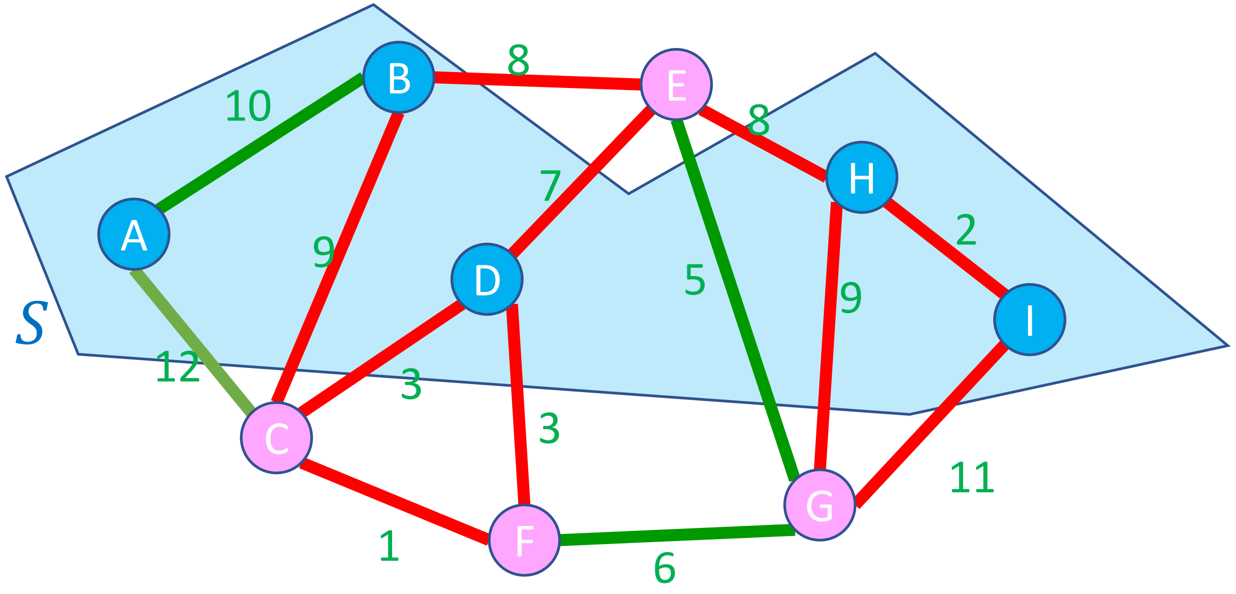
Prim's Algorithm

```
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;
}
```



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)

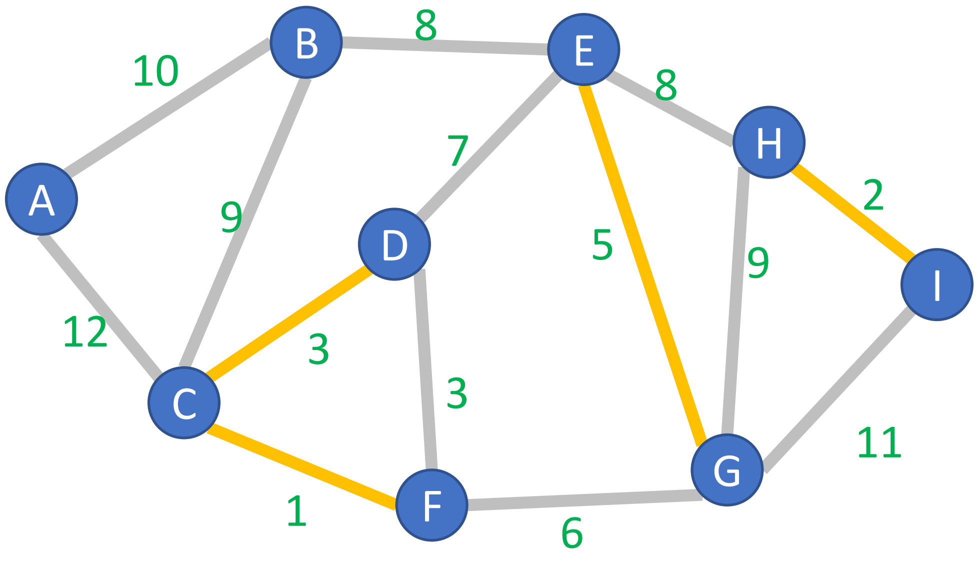
A set of edges R Respects a cut if no edges cross the cut e.g. $R = \{(A, B), (E, G), (F, G)\}$

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.

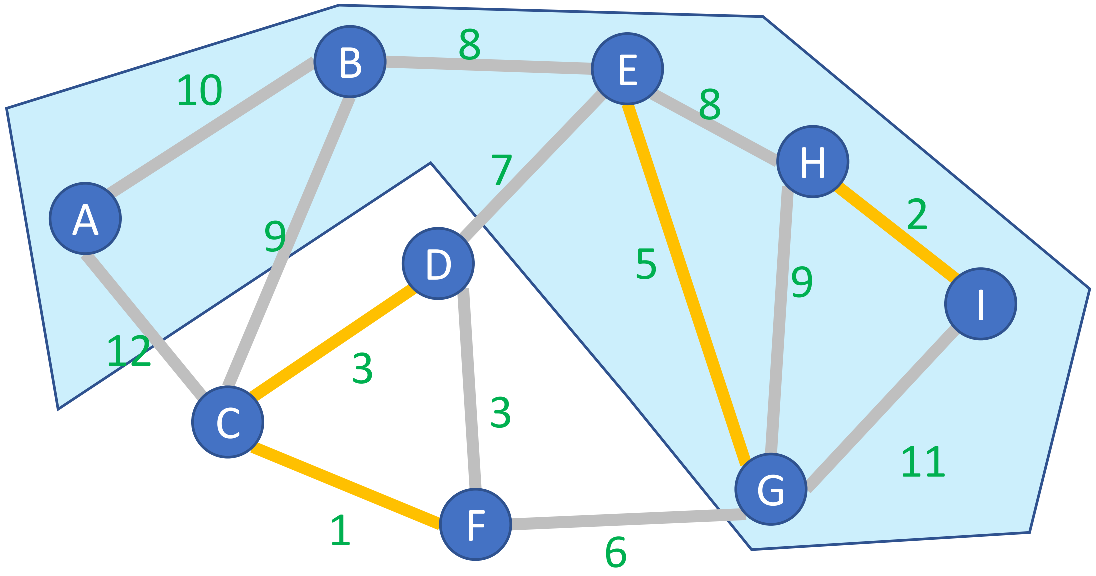
Cut Theorem

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.



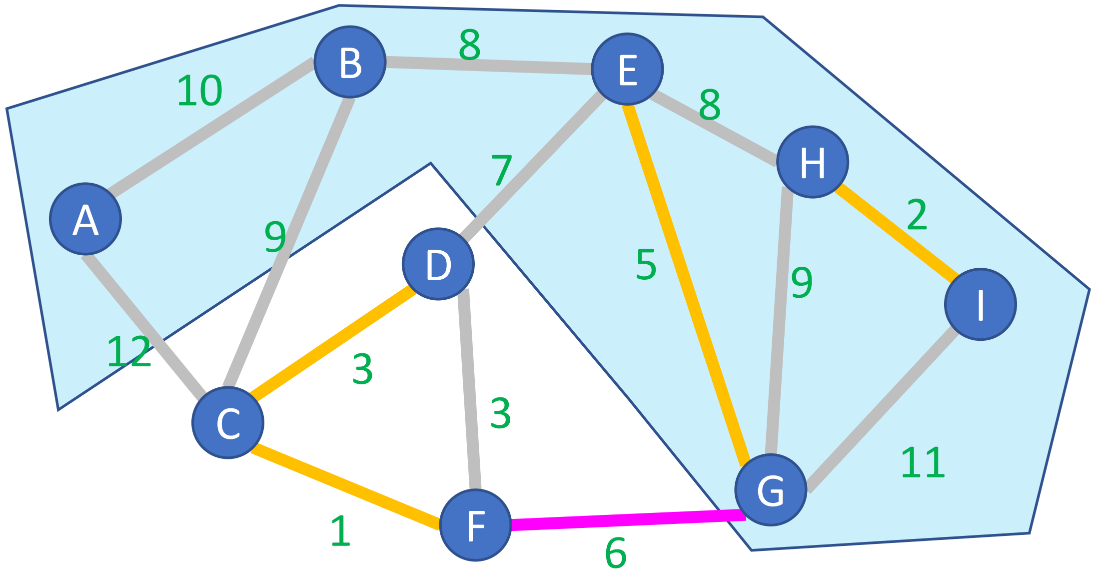
Cut Theorem

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.



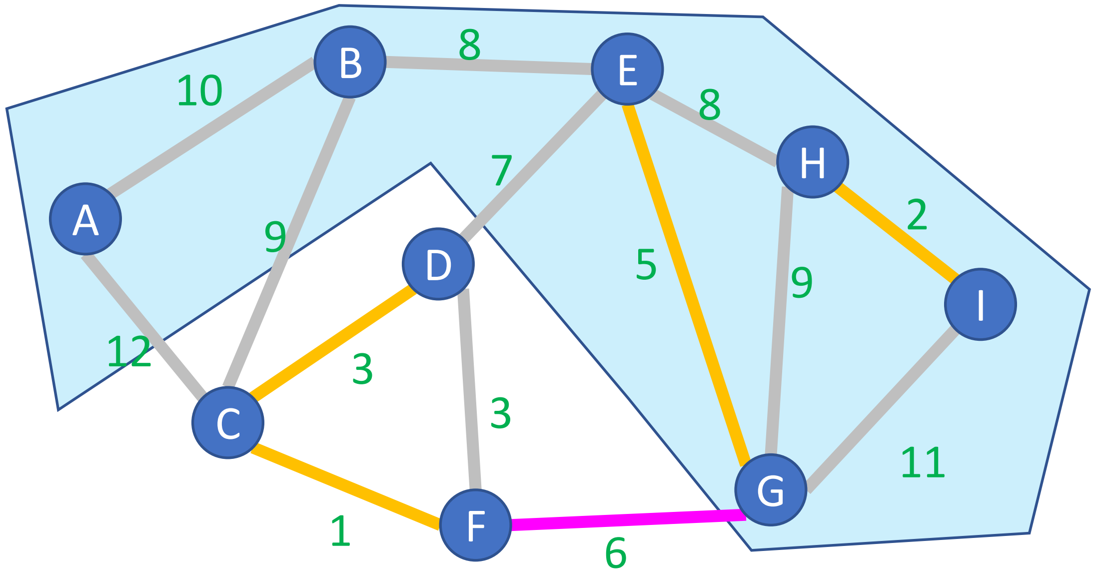
Cut Theorem

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.

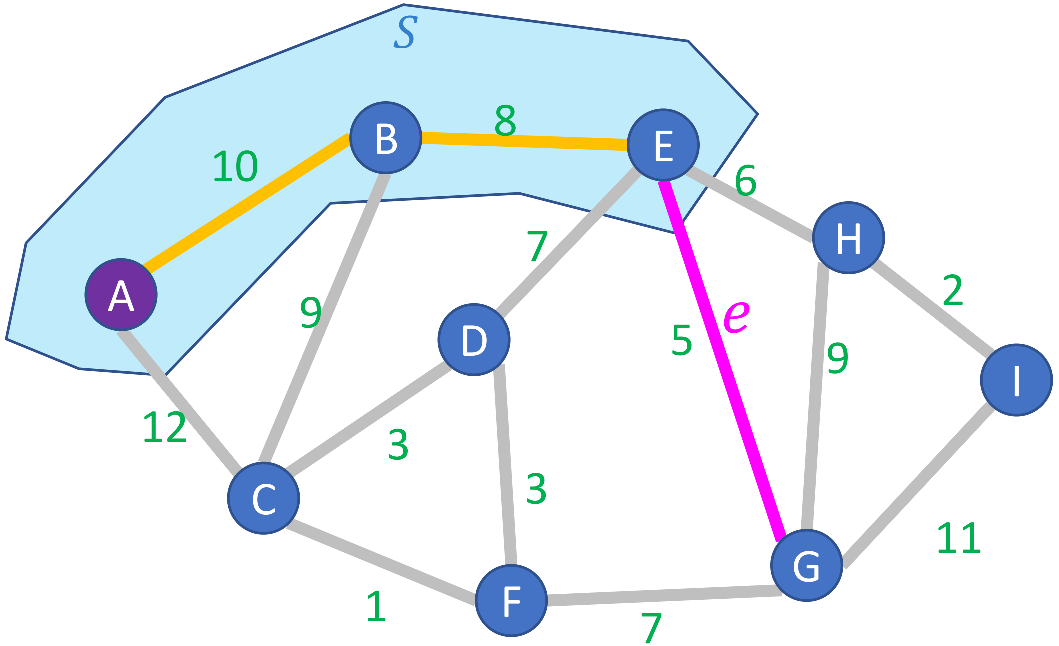


Proof of Prim's Algorithm

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!

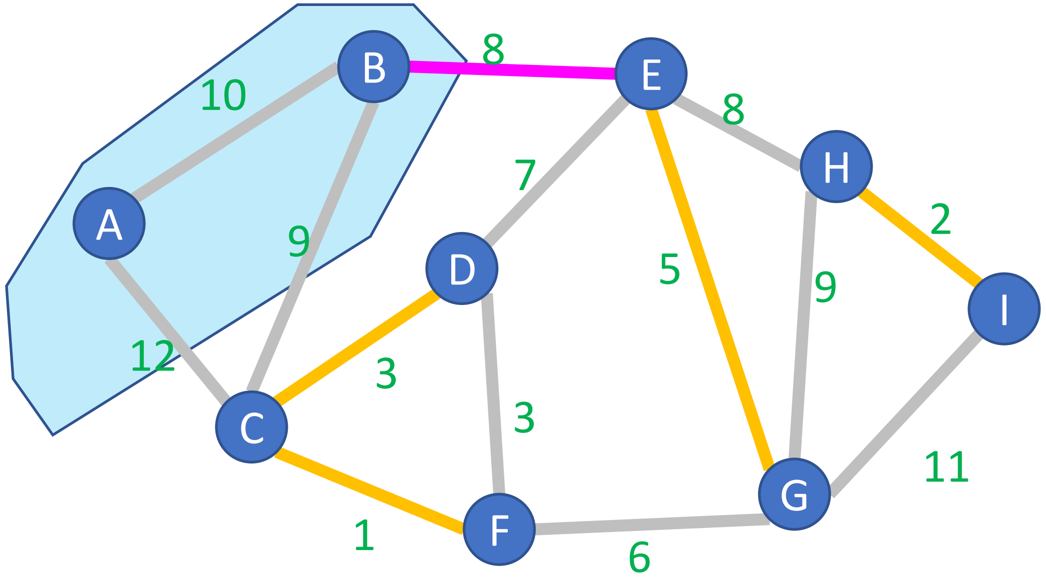
General MST Algorithm

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