CS 170
Efficient Algorithms and Intractable Problems

Lecture 10
Huffman Codes and Minimum Spanning Trees

Nika Haghtalab and John Wright

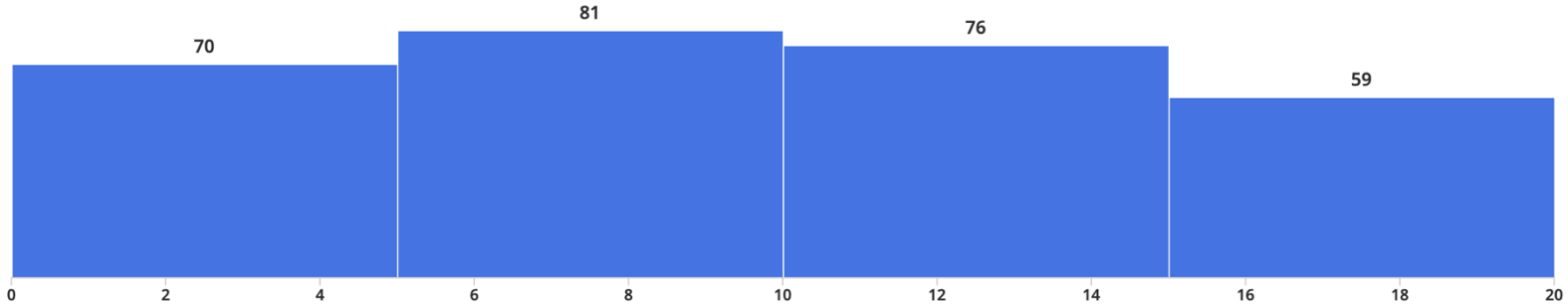
EECS, UC Berkeley

Announcements

Quiz 1 done and graded:

→ Good practice for midterm 1

→ Signaling that you need more preparation and practice for midterm 1



Minimum	Median	Maximum	Mean	Std Dev
	9.5	20.0	9.47	5.41

Midterm 1 on Thursday

→ Read the midterm logistics post on Ed.

→ No class on Thursday

→ We will have review sessions

Last Lecture and Today: Greedy Algorithms

Algorithms that build up a solution

piece by piece, always choosing the next piece

that offers the most obvious and immediate benefit!

We saw:

- Scheduling
- Optimal encoding
- Satisfiability

Today:

- Minimum Spanning Trees

Recap: A Pattern in Greedy Algorithm and Analyses

Greedy makes a series of choices. We show that no choice rules out the optimal solution. How?

Inductive Hypothesis:

→The first m choices of greedy match the first m steps of some optimal solution.

→Or, after greedy makes m choices, achieving optimal solution is still a possibility.

Base case: → At the beginning, achieving optimal is still possible!

Inductive step: **Use problem-specific structure**

If the first m choices match, we can change OPT's $m + 1^{st}$ choice to that of greedy's, and still have a valid solution that no worst than OPT.

Conclusion: The greedy algorithm outputs an optimal solution.

Today

More on greedy algorithms:

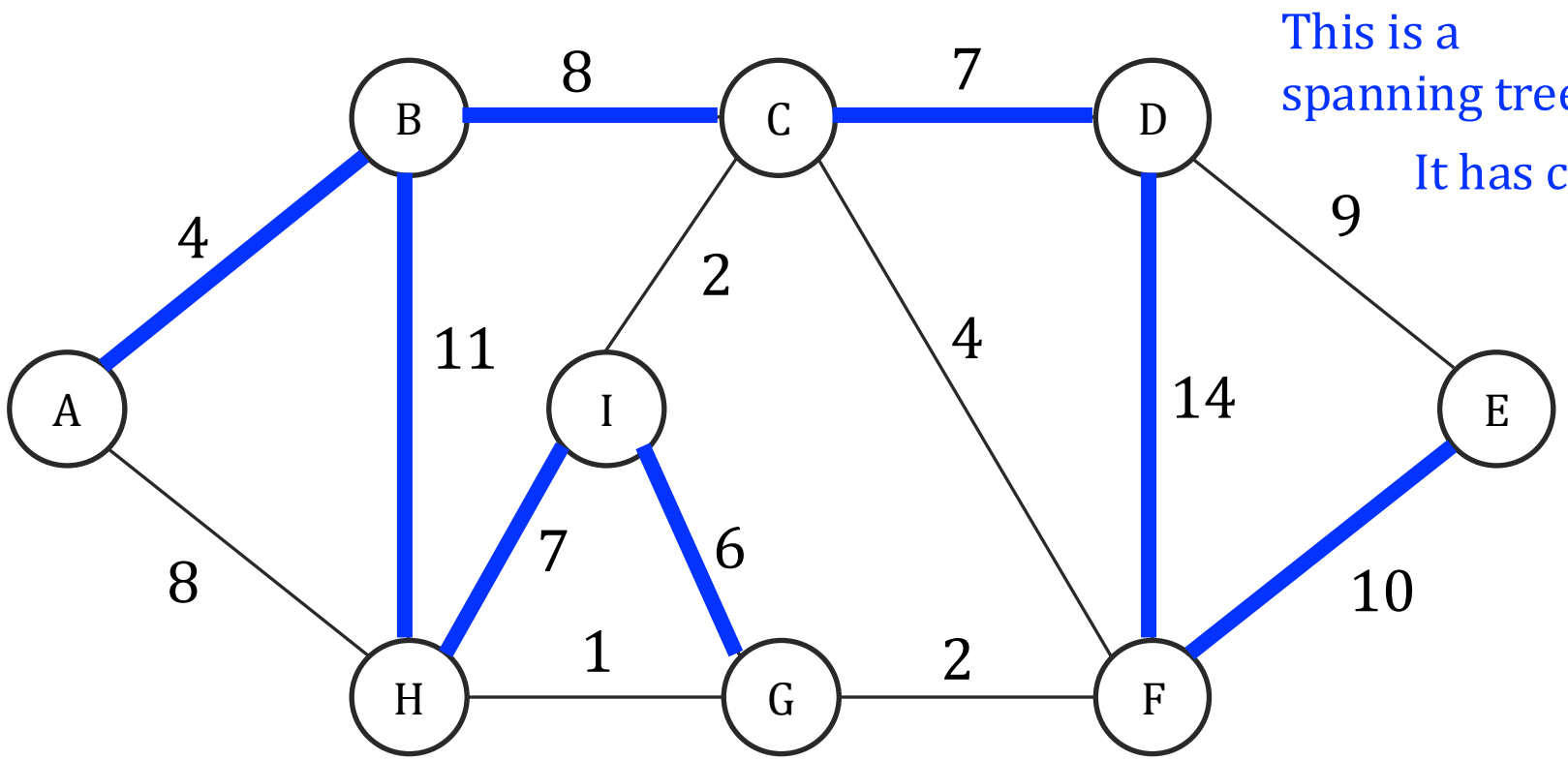
Minimum Spanning Trees

Minimum Spanning Trees

Definition: A spanning tree, is a tree that **connects all vertices** of a graph G.

Cost of a tree

$$\text{cost(T)} = \sum_{e \in T} w_e$$



This is a spanning tree.

It has cost 67

Minimum Spanning Tree (MST) Problem:

Input: a weighted graph $G = (V, E)$ with non-negative weights.

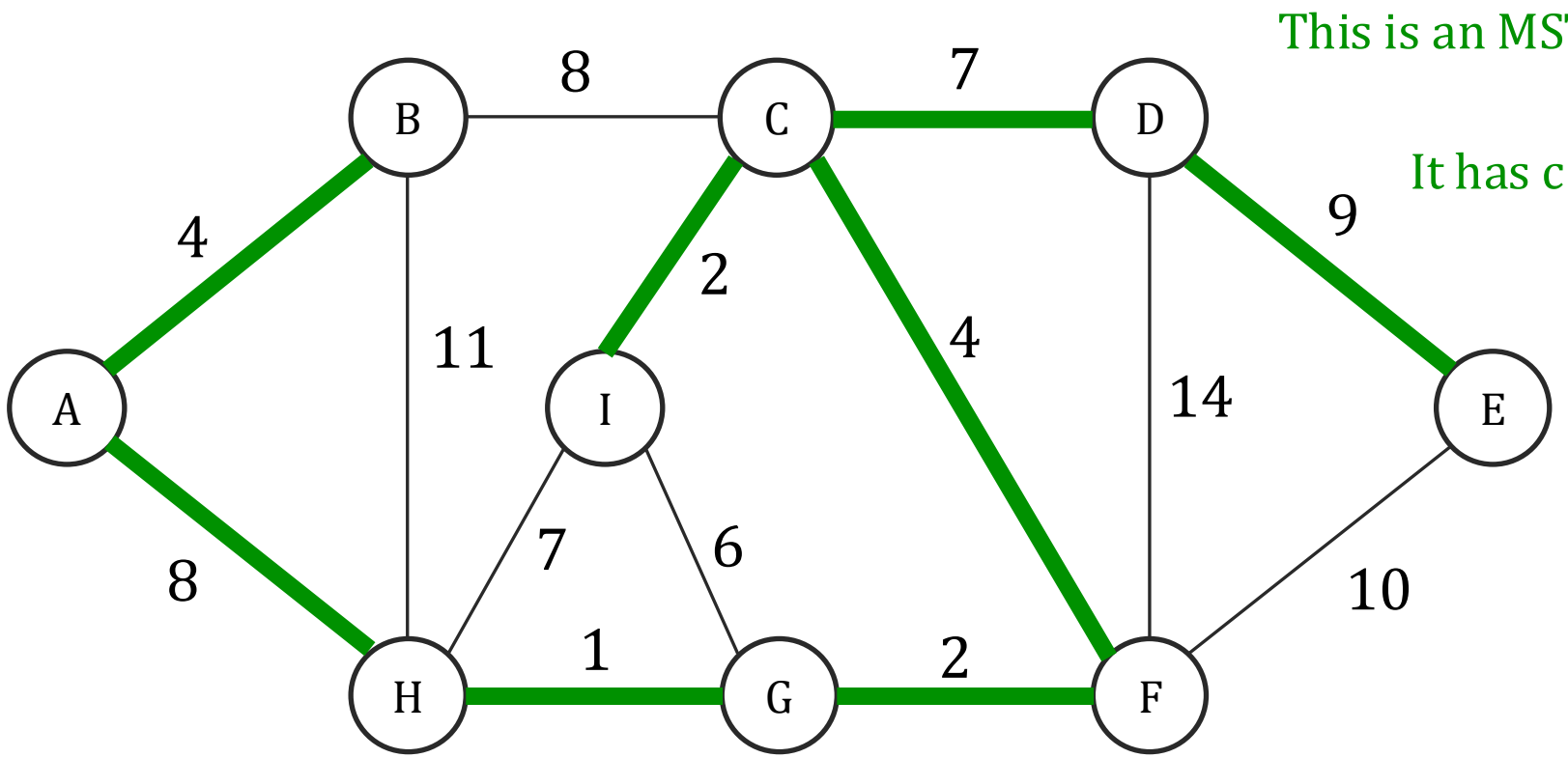
Output: A set of edges that connected graph and has the **smallest cost**.

Minimum Spanning Trees

Definition: A spanning tree, is a tree that **connects all vertices** of a graph G.

Cost of a tree

$$\text{cost(T)} = \sum_{e \in T} w_e$$



Minimum Spanning Tree (MST) Problem:

Input: a weighted graph $G = (V, E)$ with non-negative weights.

Output: A set of edges that connected graph and has the **smallest cost**.

MST applications and Algorithms

Biggest applications:

- Network design: Connecting cities with roads/electricity/telephone/ …
- Pre-processing for other algorithms.

We will see two greedy algorithms for building Minimum Spanning Trees.

What do MSTs look like?

Facts about Trees

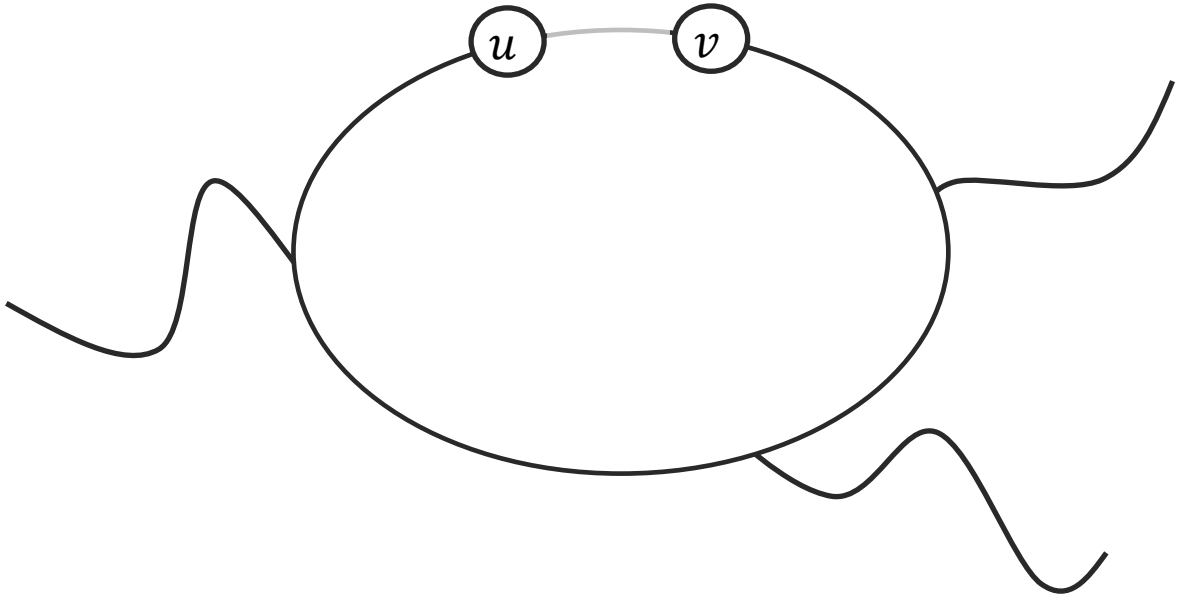
The following are two equivalent definition of a tree on n vertices.

1. A connected acyclic graph.
2. A connected graph with $n - 1$ edges.

Any **minimum weight** set of edges that **connects all vertices** is a **tree**! Why?

If a set of edges connecting all vertices has a cycle, we can remove one of its edges and still connect all vertices.

→ Removing any edge on the cycle, keeps the graph still connected.

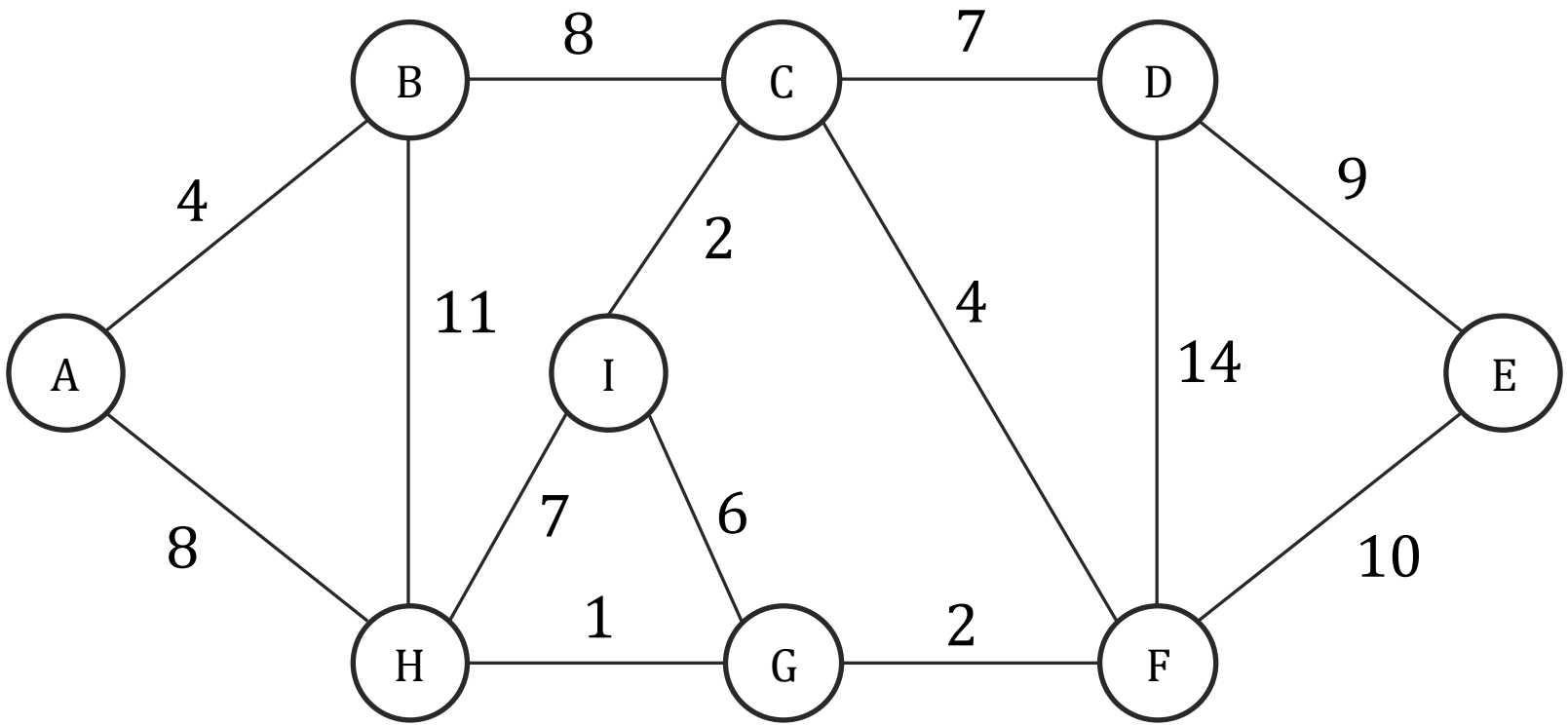


Graph Structures and Facts

Cuts and Graphs

Definition: A **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

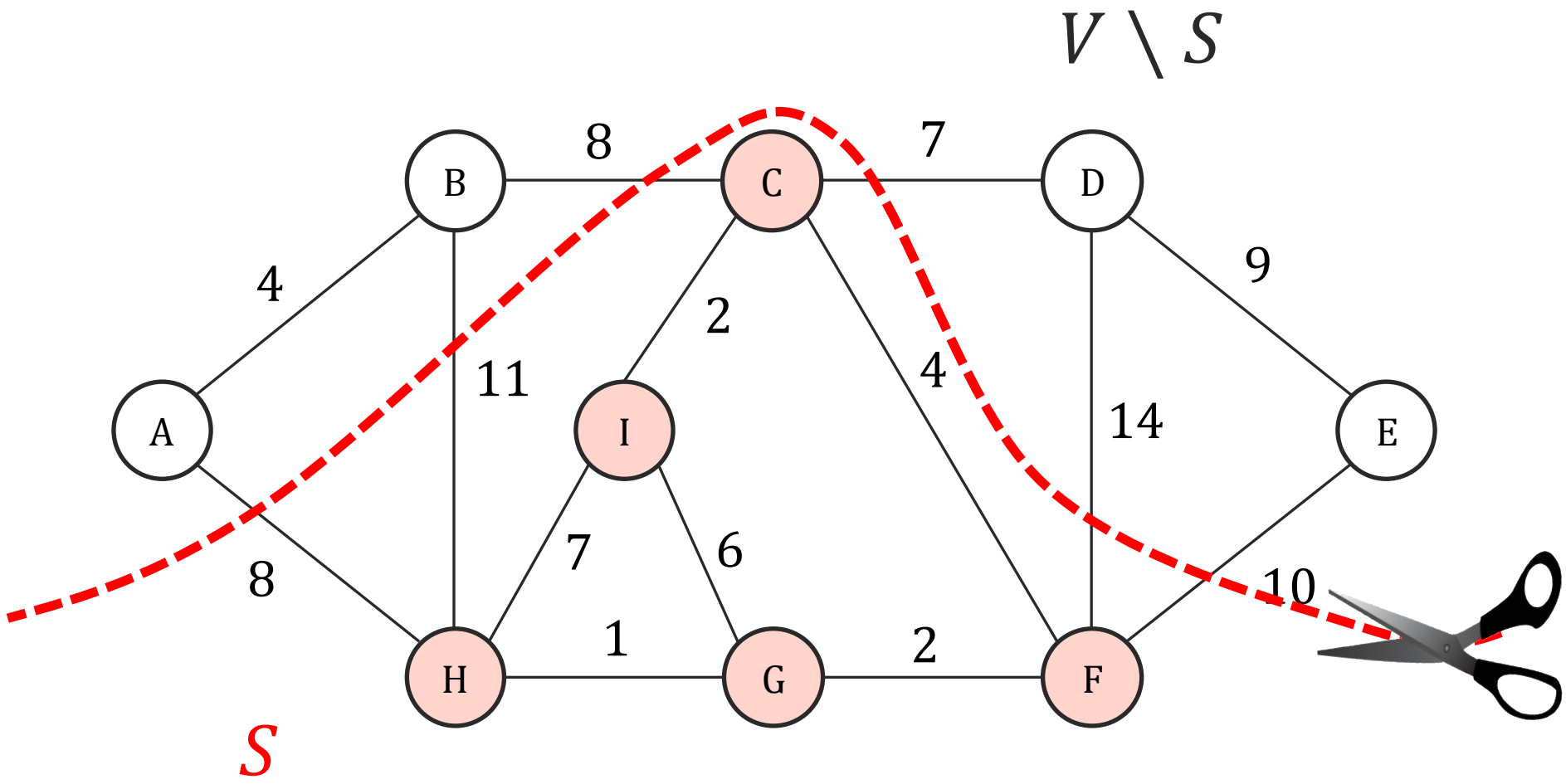
→ we'll color them differently to make the two sets clear.



Cuts and Graphs

Definition: A **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

→ we'll color them differently to make the two sets clear.

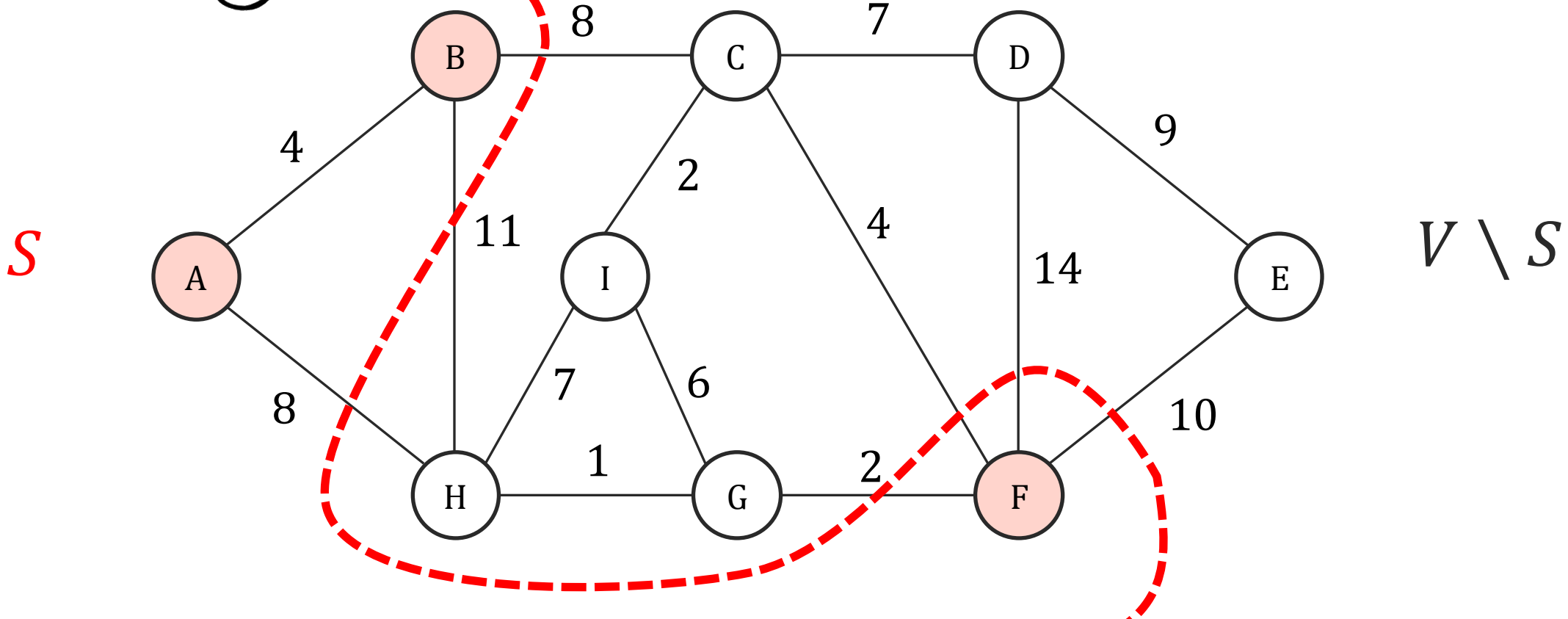


Cuts and Graphs

Definition: A **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

→ we'll color them differently to make the two sets clear.

This is another cut



Greedy Algorithms and Cuts

Imagine, we already discovered some of the edges X of a minimum spanning tree T.

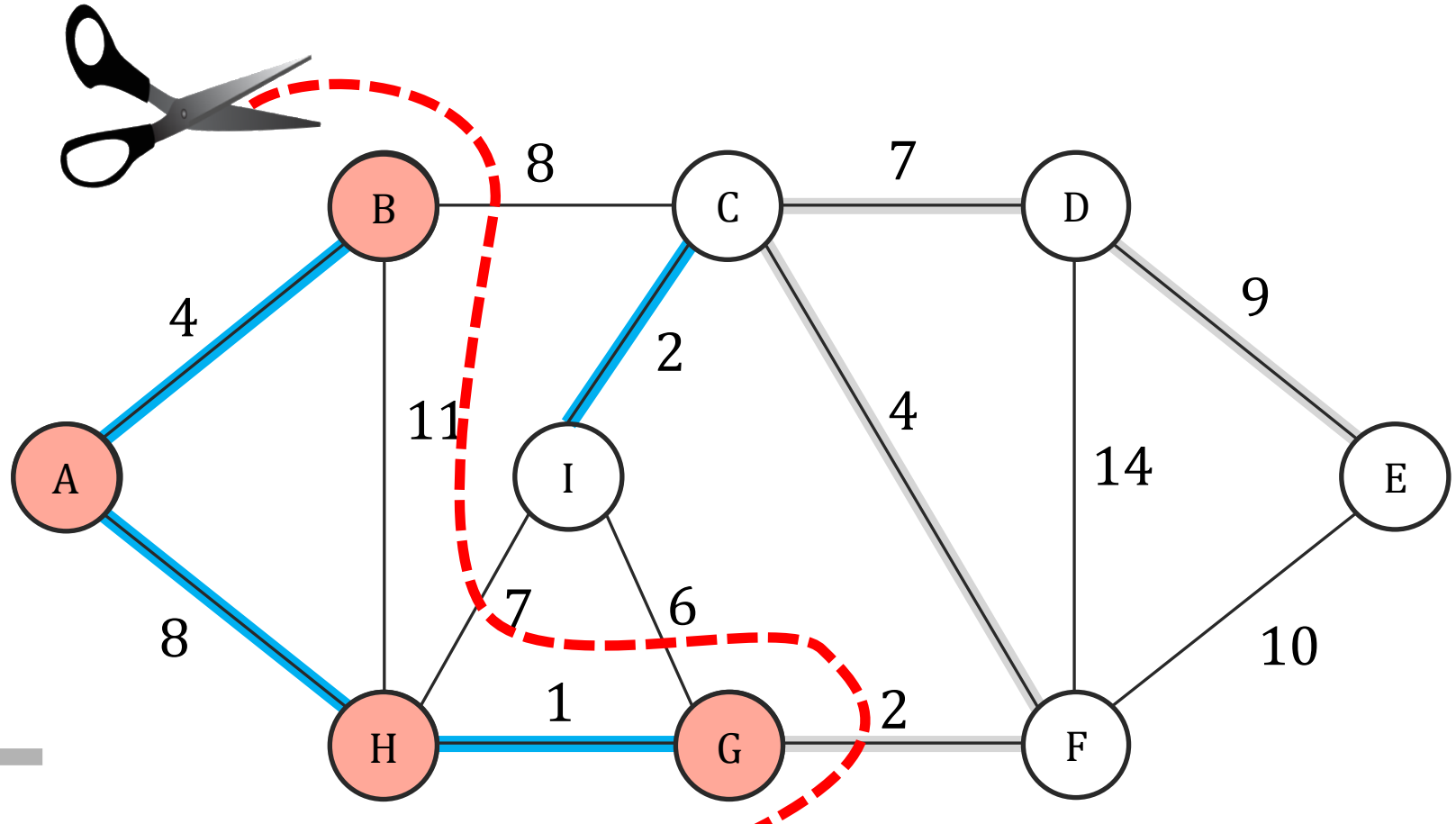
Take any **cut** where edges X don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut?

Edges X:

MST T:

and



Greedy Algorithms and Cuts

Imagine, we already discovered some of the edges X of a minimum spanning tree T.

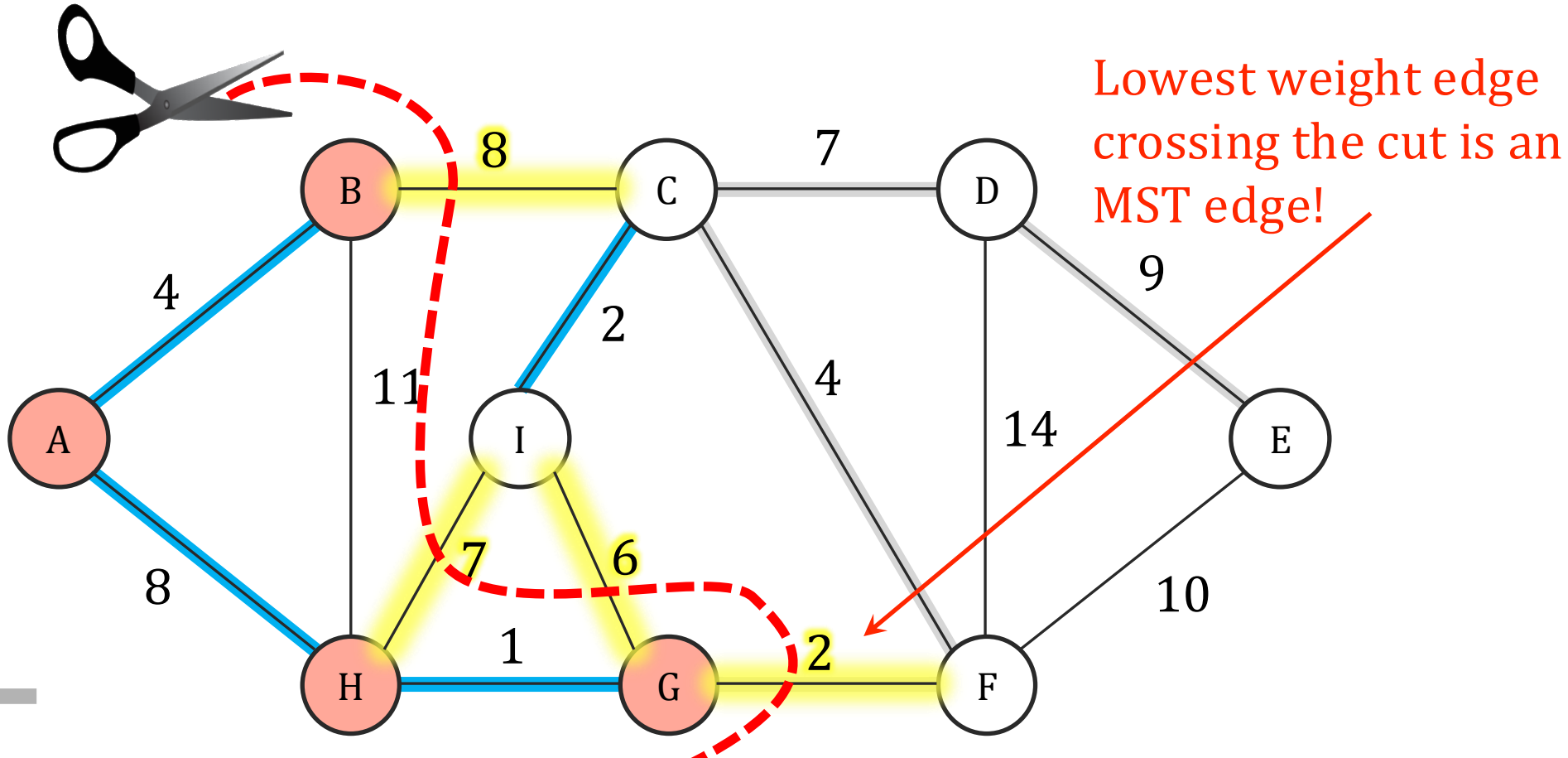
Take any **cut** where edges X don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut?

Edges X:

MST T:

and



Formally: The Cut Property

Claim: Suppose $X \subseteq E$ is part of an MST for graph G. Consider a cut S, V $\setminus S$, such that

- X has no edges from S to $V \setminus S$.

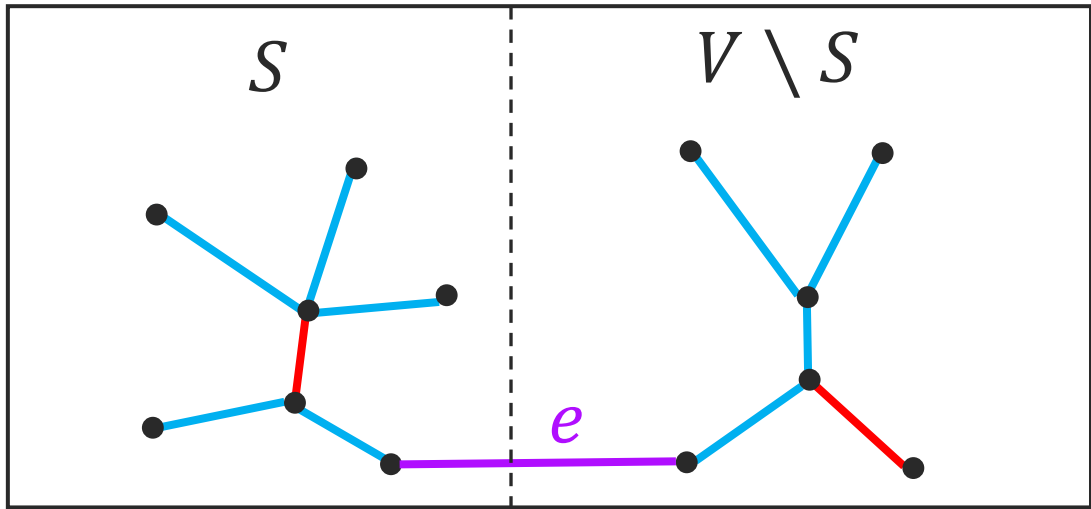
Let $e \in E$ be the smallest weight edge from S to $V \setminus S$.

Then $X \cup \{e\}$ is **also a subset of an MST for graph** G.

Proof: Take the MST T that satisfies the conditions of the above claim

Case 1) $e \in T$. Then by definition $X \cup \{e\} \in T$.

X: blue edges
T: blue and red edges.



Formally: The Cut Property

Claim: Suppose $X \subseteq E$ is part of an MST for graph G. Consider a cut S, V $\setminus S$, such that

- X has no edges from S to $V \setminus S$.

Let $e \in E$ be the smallest weight edge from S to $V \setminus S$.

Then $X \cup \{e\}$ is **also a subset of an MST for graph** G.

Proof: Take the MST T that satisfies the conditions of the above claim.

X: blue edges
T: blue and red edges.

Case 2) $e \notin T$**.** Then, $T \cup \{e\}$ must form a cycle

→ This cycle must have another edge $e' \in T$ that crosses from S to $V \setminus S$.

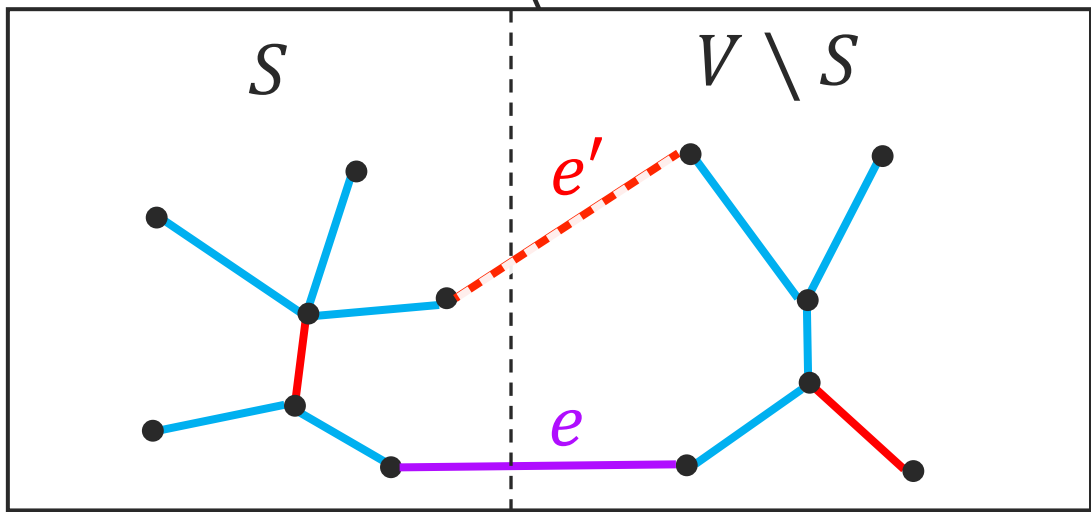
Consider $T' = T \cup \{e\} \setminus e'$:

→T' also connects all vertices of the graph

→$cost(T') = cost(T) + w_e - w_{e'} \leq cost\ (T)$.

→So, T' is also a minimum spanning tree!

$X \cup \{e\}$ is **also a subset of an MST for graph** G



Greedy Algorithms based on the Cut Property

Any algorithm that fits the following form finds an MST.

Meta Algorithm for MST

$X = \{\}$
Repeat until $|X| = |V| - 1$
  Pick $S \subseteq V$, s.t. X has no edges from S to $V \setminus S$
  $e \leftarrow$ smallest weight edge from S to $V \setminus S$
  $X \leftarrow X \cup \{e\}$

Different Algorithms pick S differently

Claim: The meta Algorithm above returns a minimum spanning tree.
Proof: By induction …
Induction step:
The cut property ensures that $X \cup \{e\}$ is always a subset of an MST.

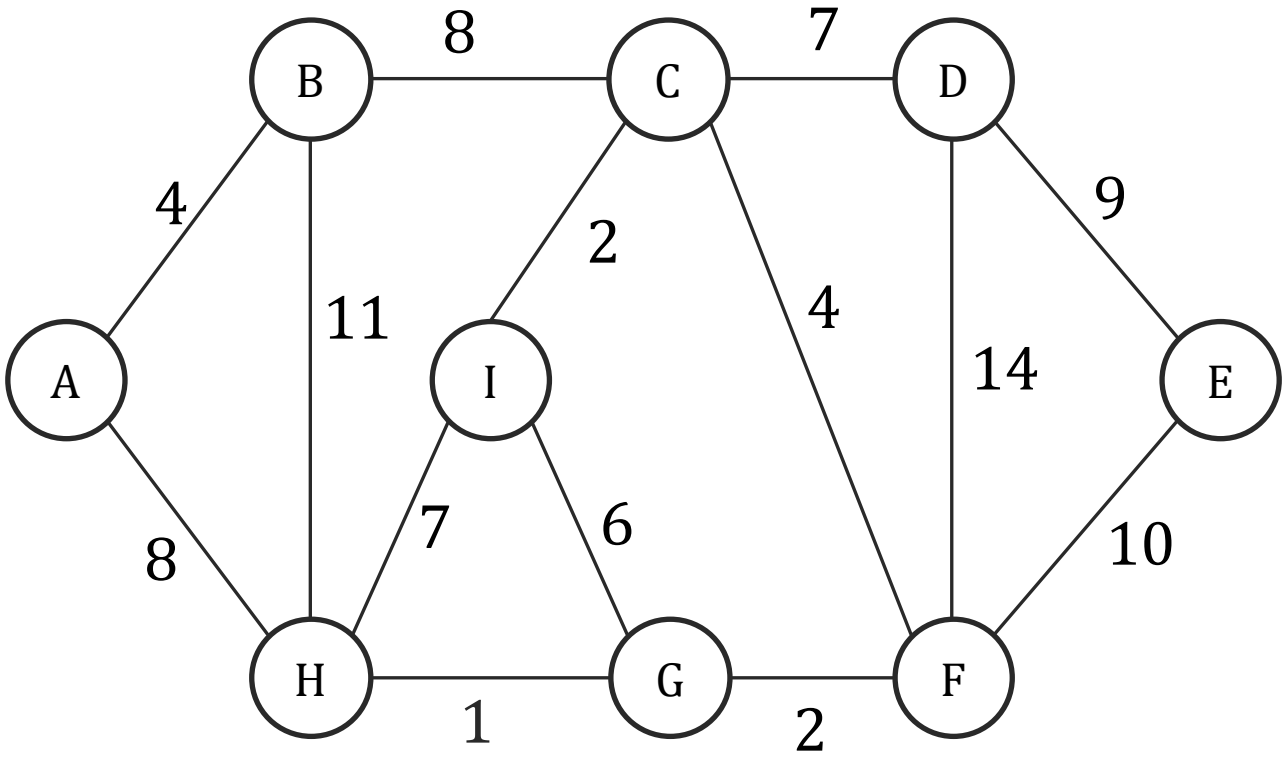
Easy: Practice formalizing this induction.

Kruskal’s Algorithm

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

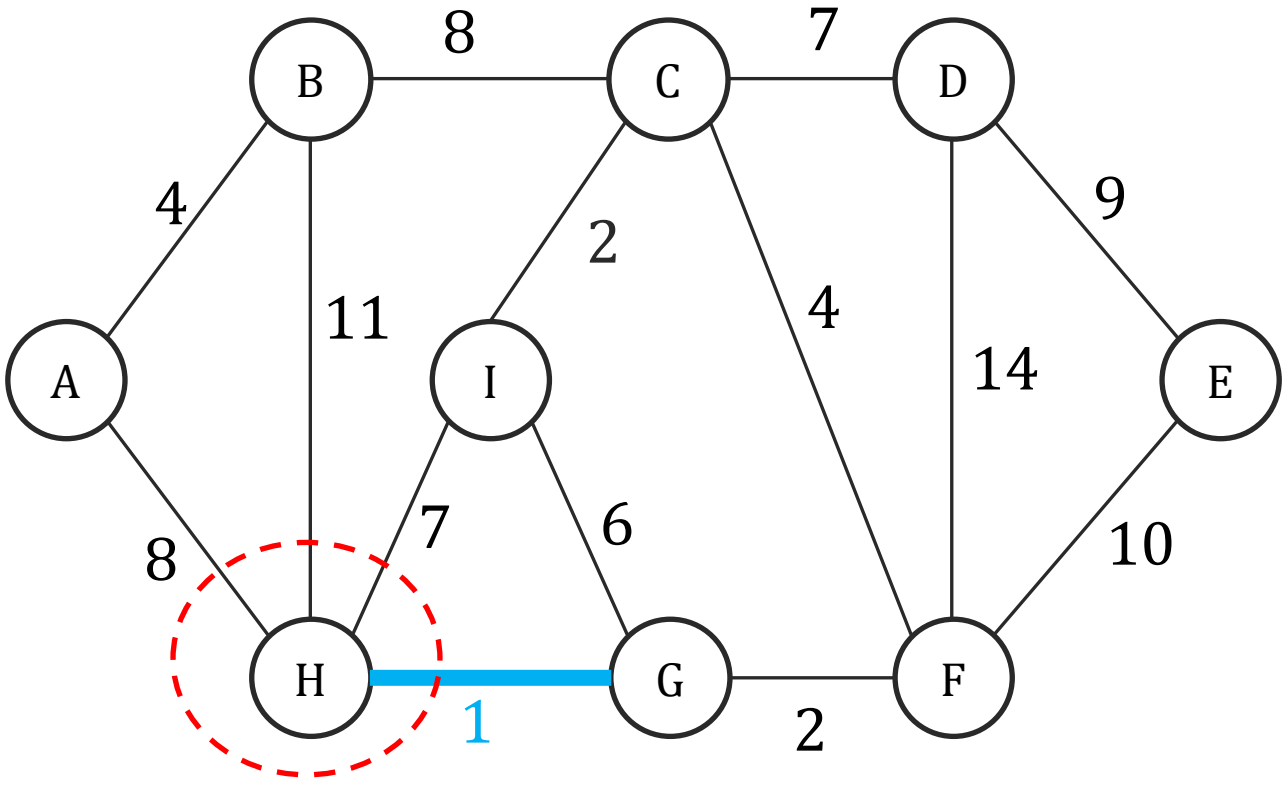
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

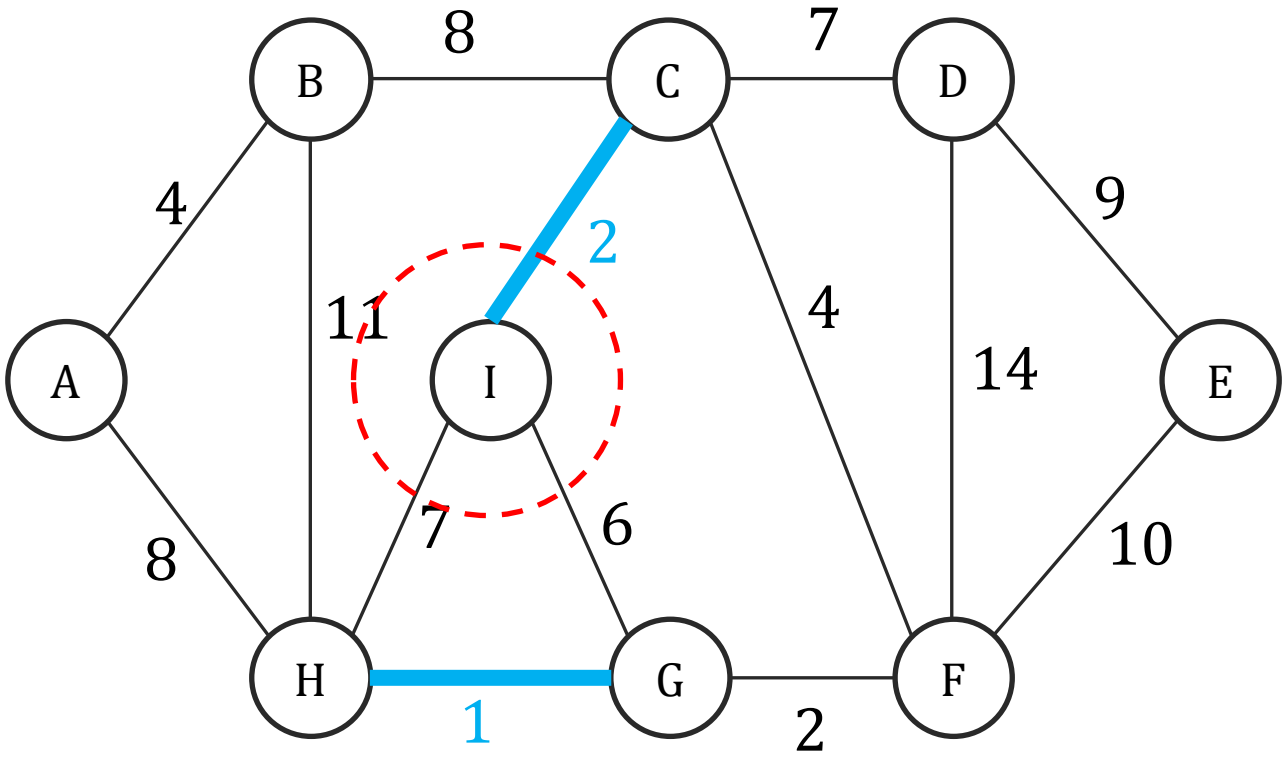
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

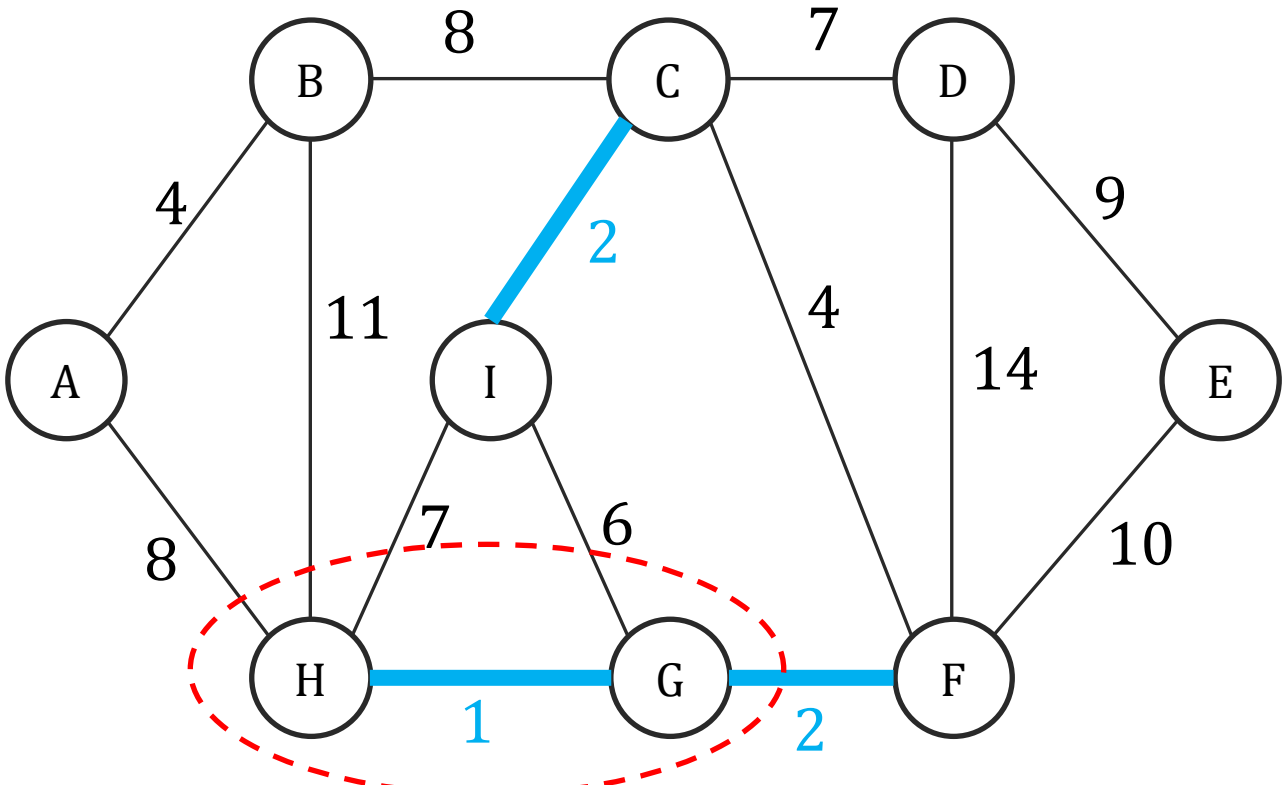
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

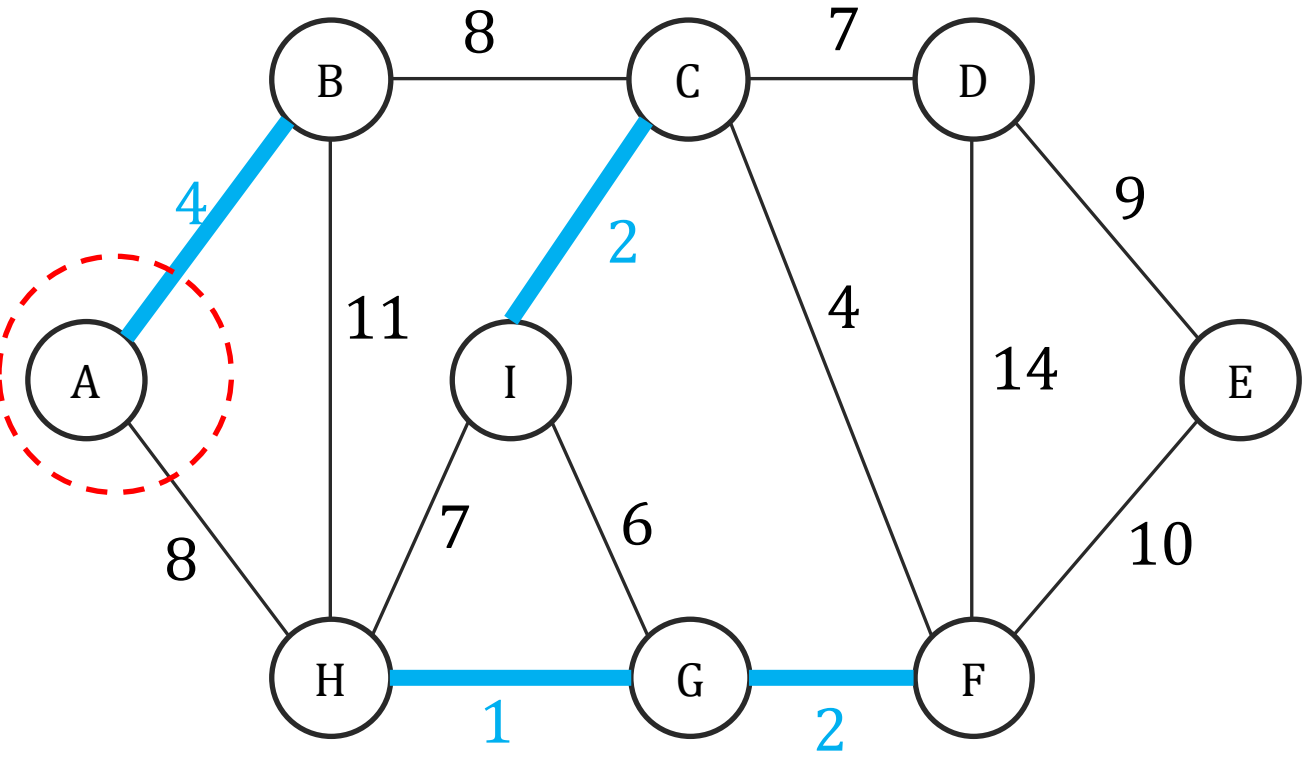
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

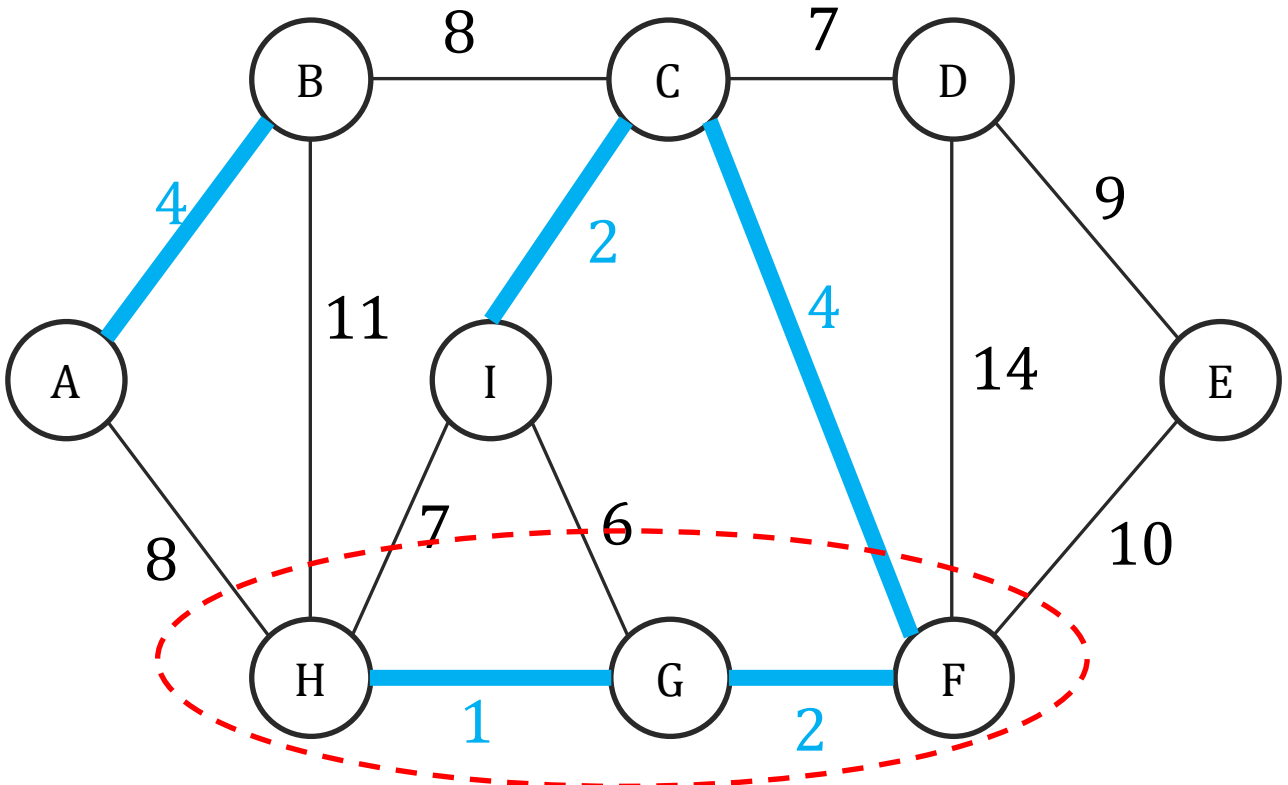
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

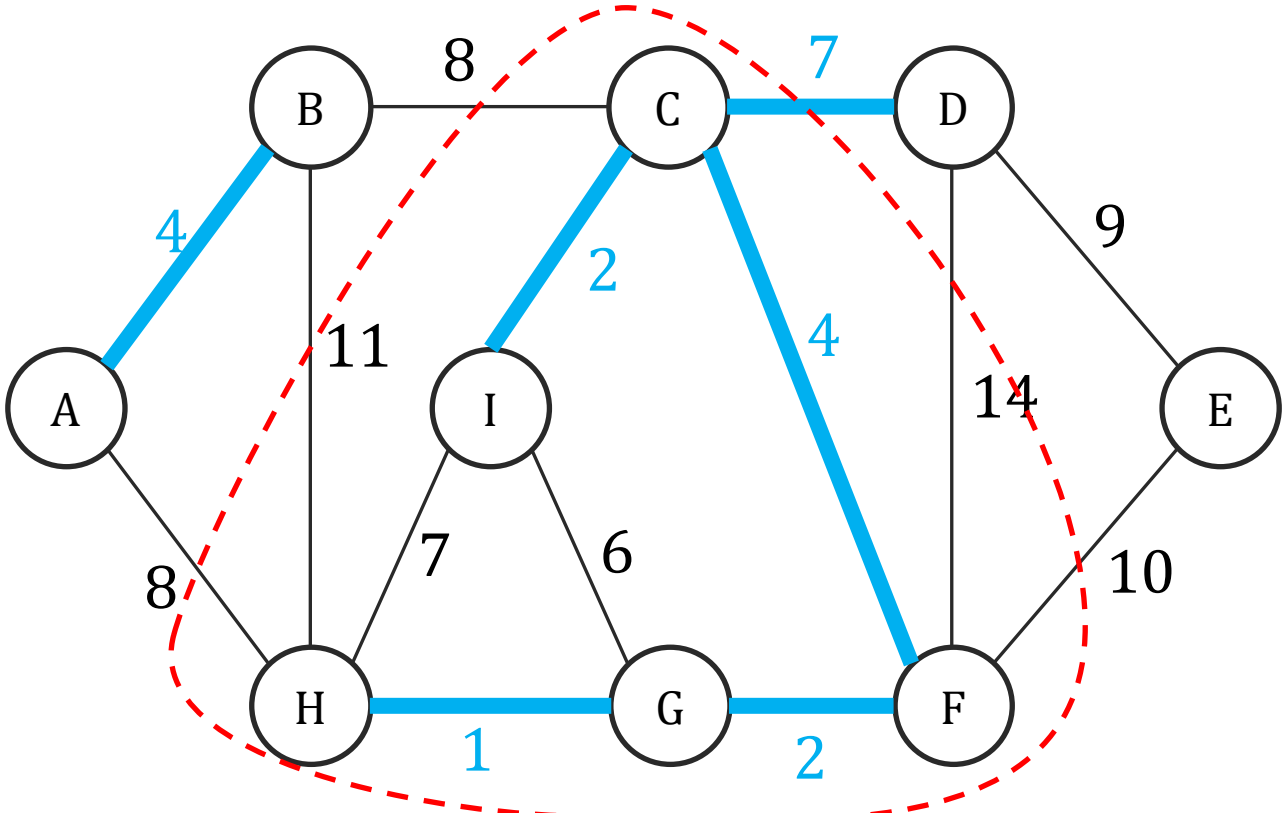
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

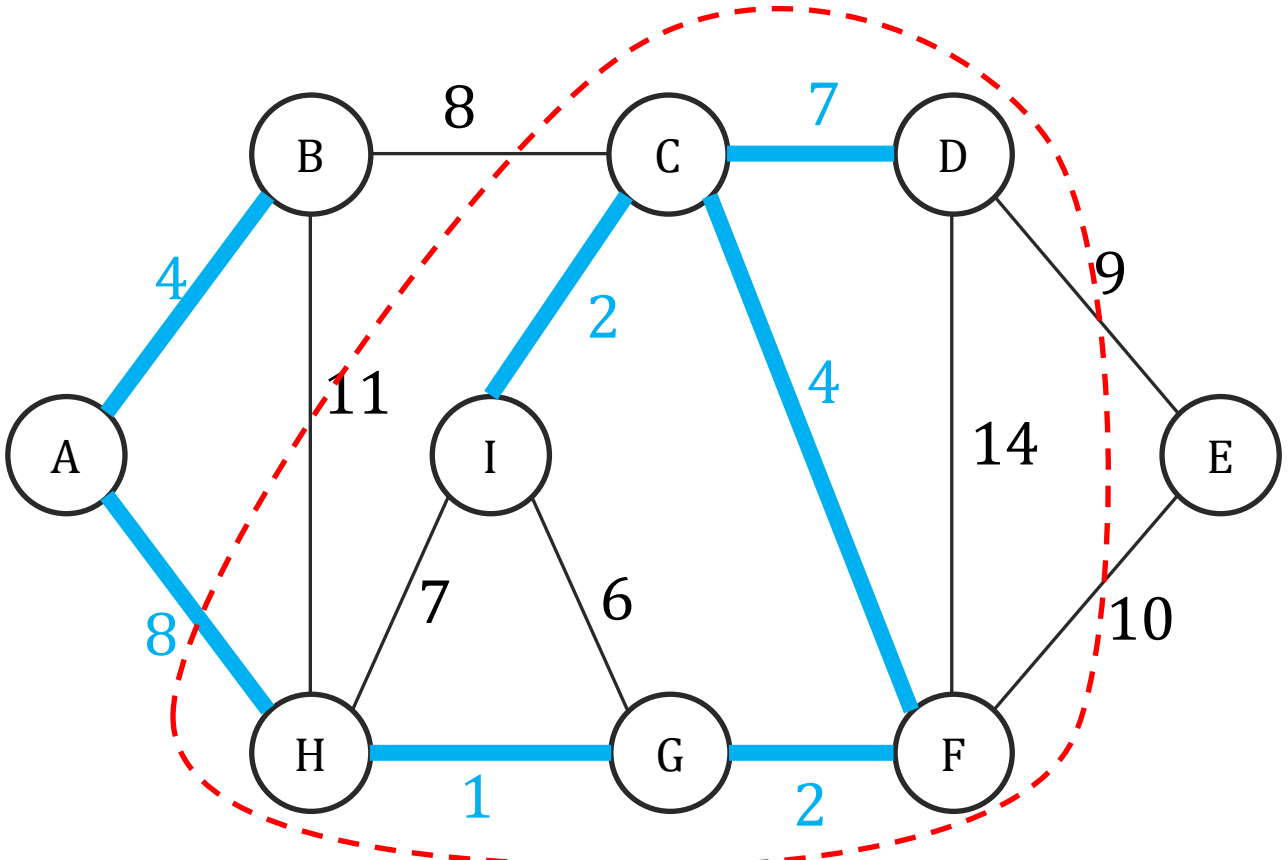
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

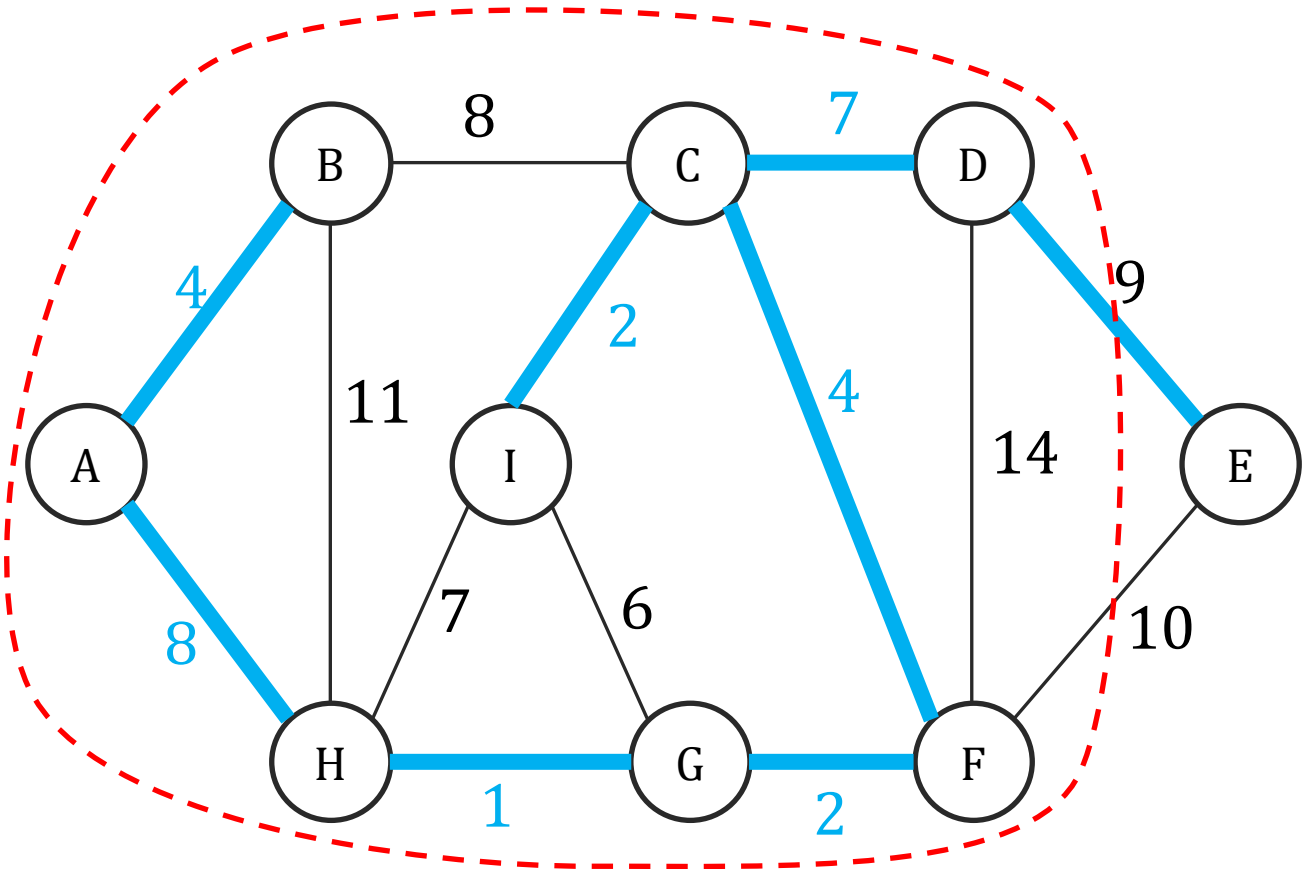
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Correctness

Does Kruskal return a minimum spanning tree?

- Since $X \cup \{(u, v)\}$ doesn't have a cycle, u and v belong to two different connected components of X.
- Let $S \leftarrow$ Connected component including u
- So (u, v) is the lightest edge from S to $V \setminus S$.

→Kruskal fits the meta algorithm description, so it find an MST.

Kruskal's Runtime and Union-Find

How do we quickly check if $X \cup \{(u, v)\}$ has a cycle?

→ We need to check if u's connected component in X = v's connected component in X

Union-FIND: A data-structure for **disjoint sets**

- makeSet(u): create a set from element u. Takes $O(1)$
- find(u): return the set that includes element u. Takes $O(\log(n))$
- union(u, v): Merge two sets containing u and v. Takes $O(\log(n))$

Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u, v) \in$ E in increasing order of weight
    If find$(v) \neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Runtime of Kruskal's Algorithm

Sorting m edges: $O(m\log(m)) = O(m\log(n))$. Since $m \leq n^2$.

Everything else:

- n calls to makeSet
- $2m$ calls to find: 2 calls per edge to find its endpoints.
- $n-1$ calls to union: A tree has $n-1$ edges.

Total: $O((m+n)\log(n))$. For connected graphs = $O(m\log(n))$.

Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u,v) \in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u,v)\}$
      union(u,v)
return X

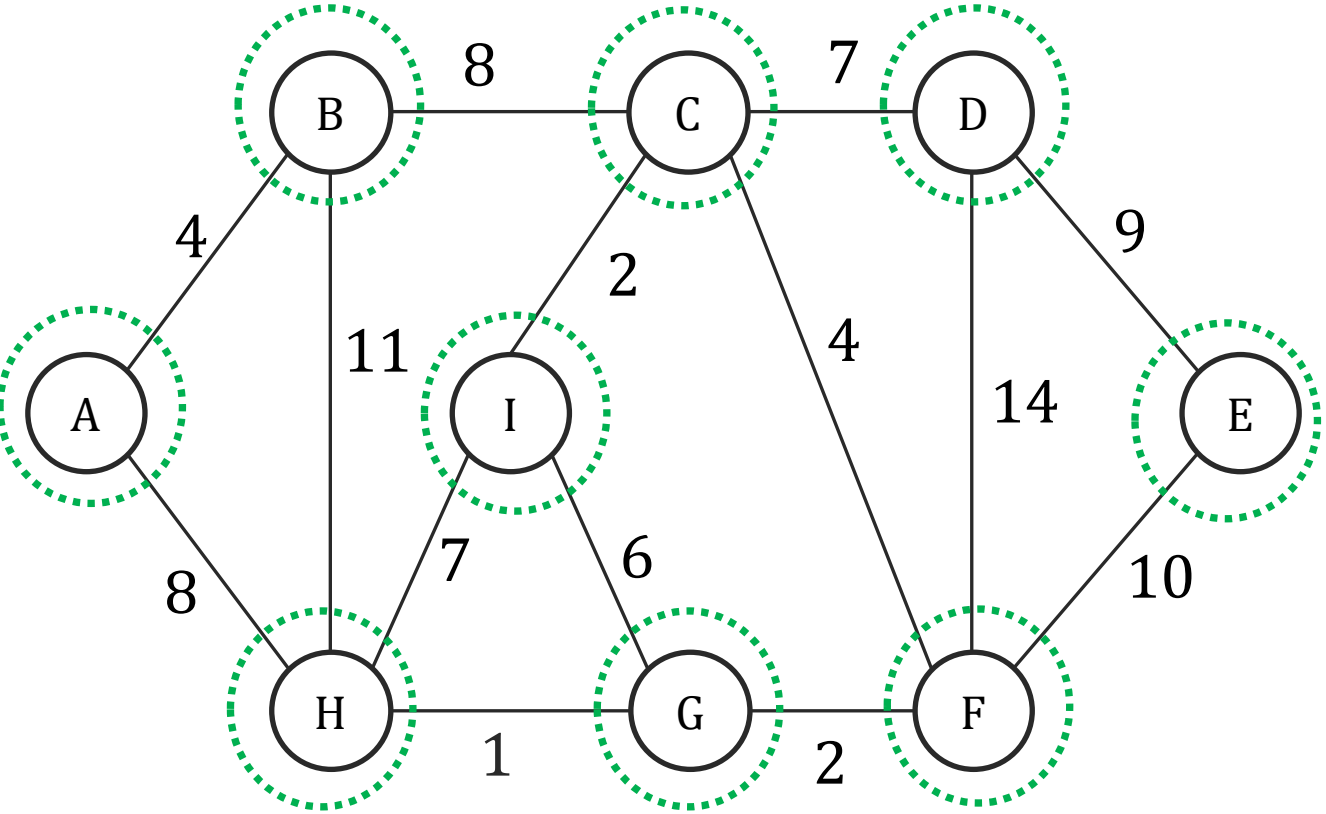
Run through Kruskal's Algorithm and track the connected components

Answer in the next few slides

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

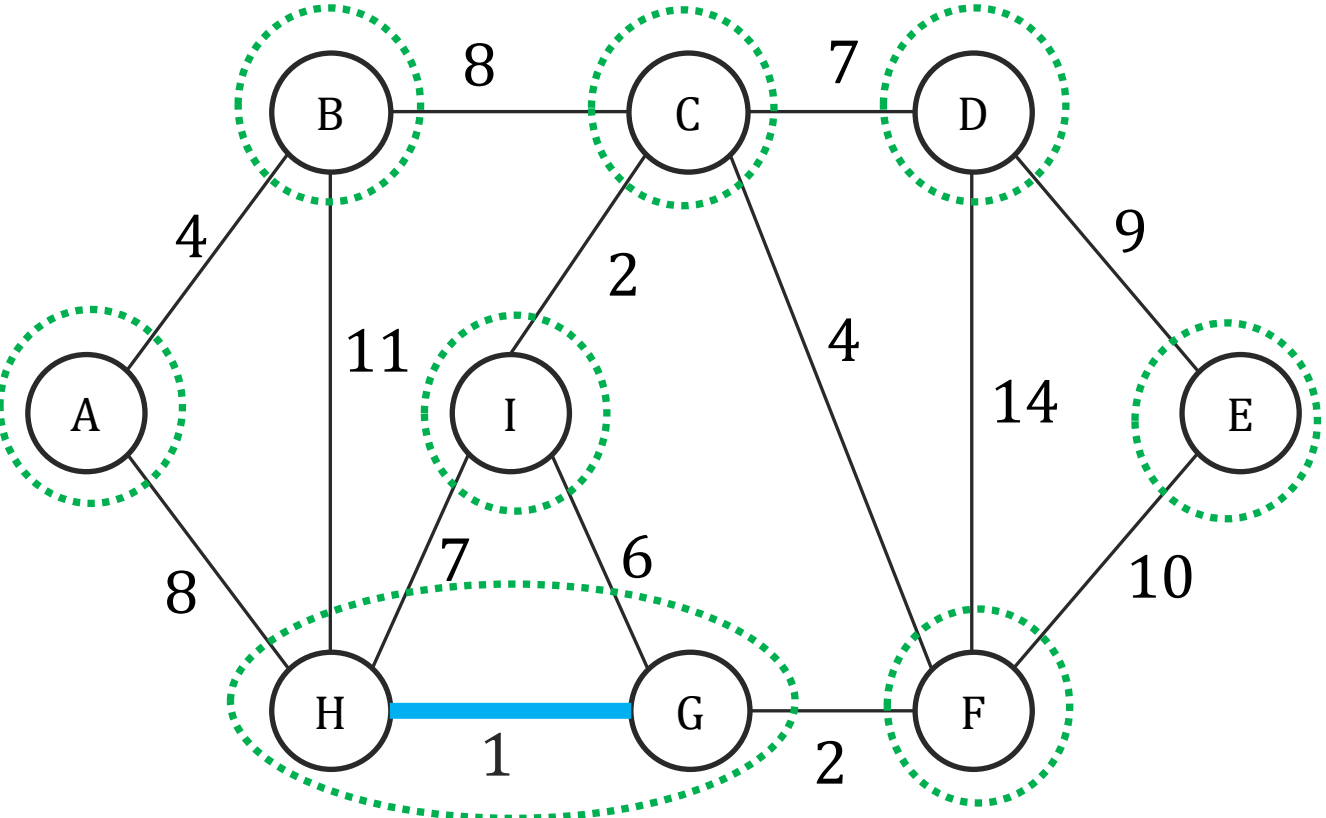


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

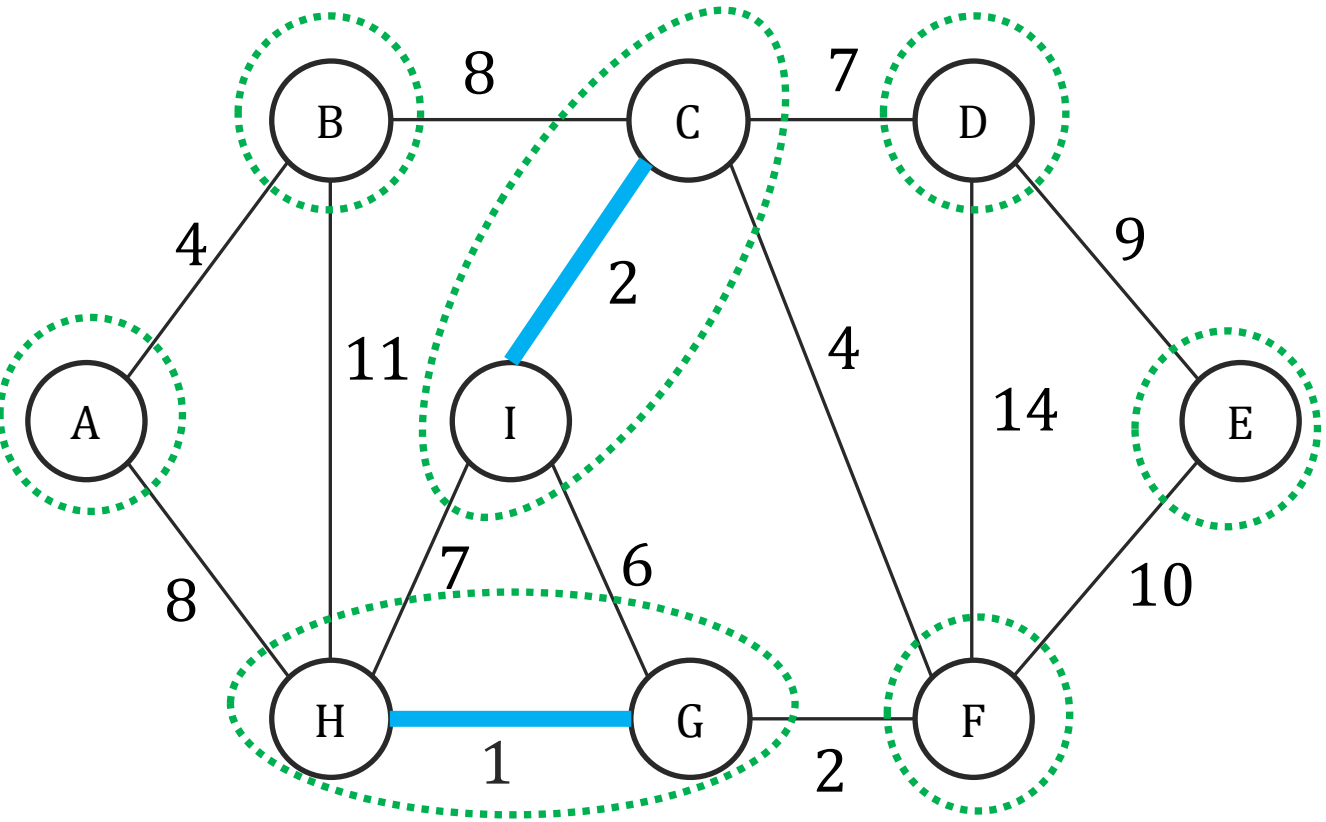


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

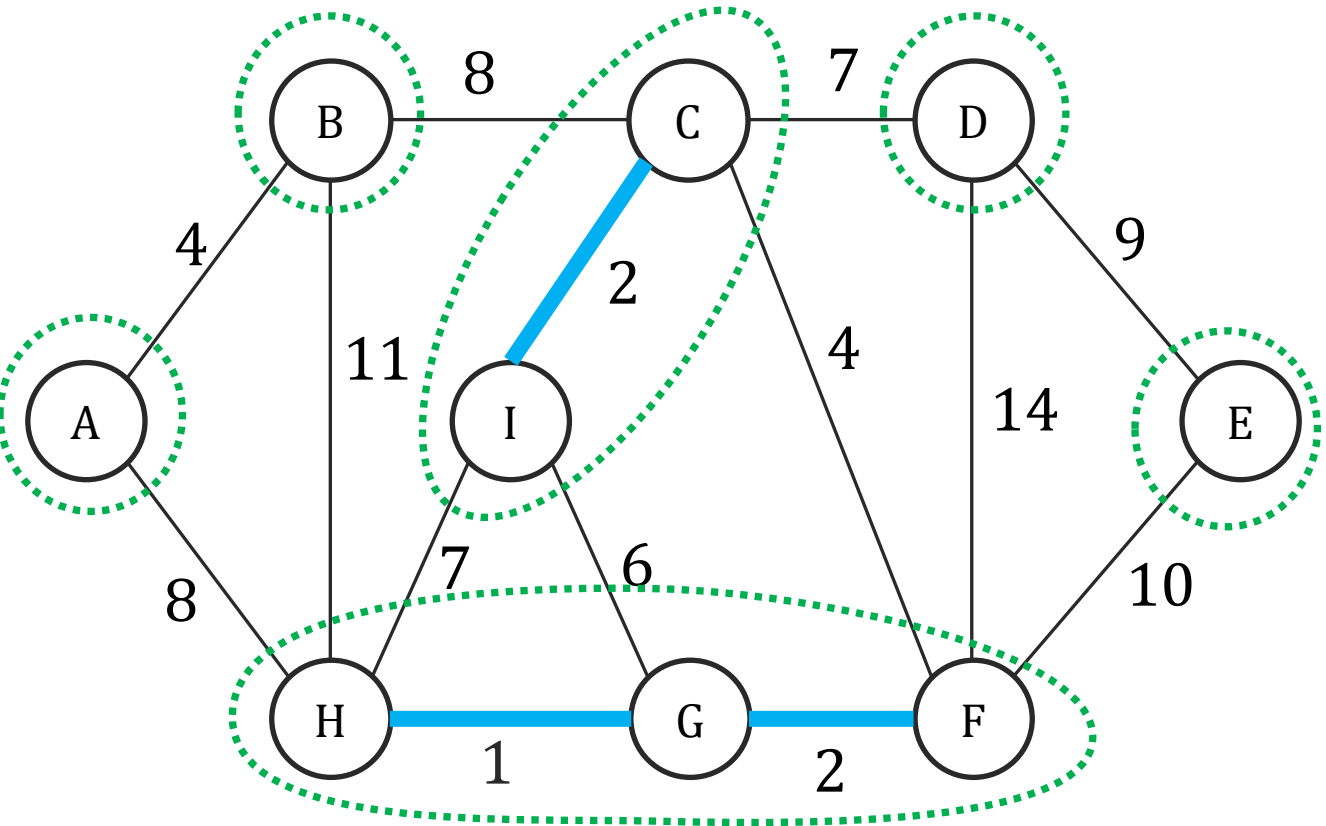


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

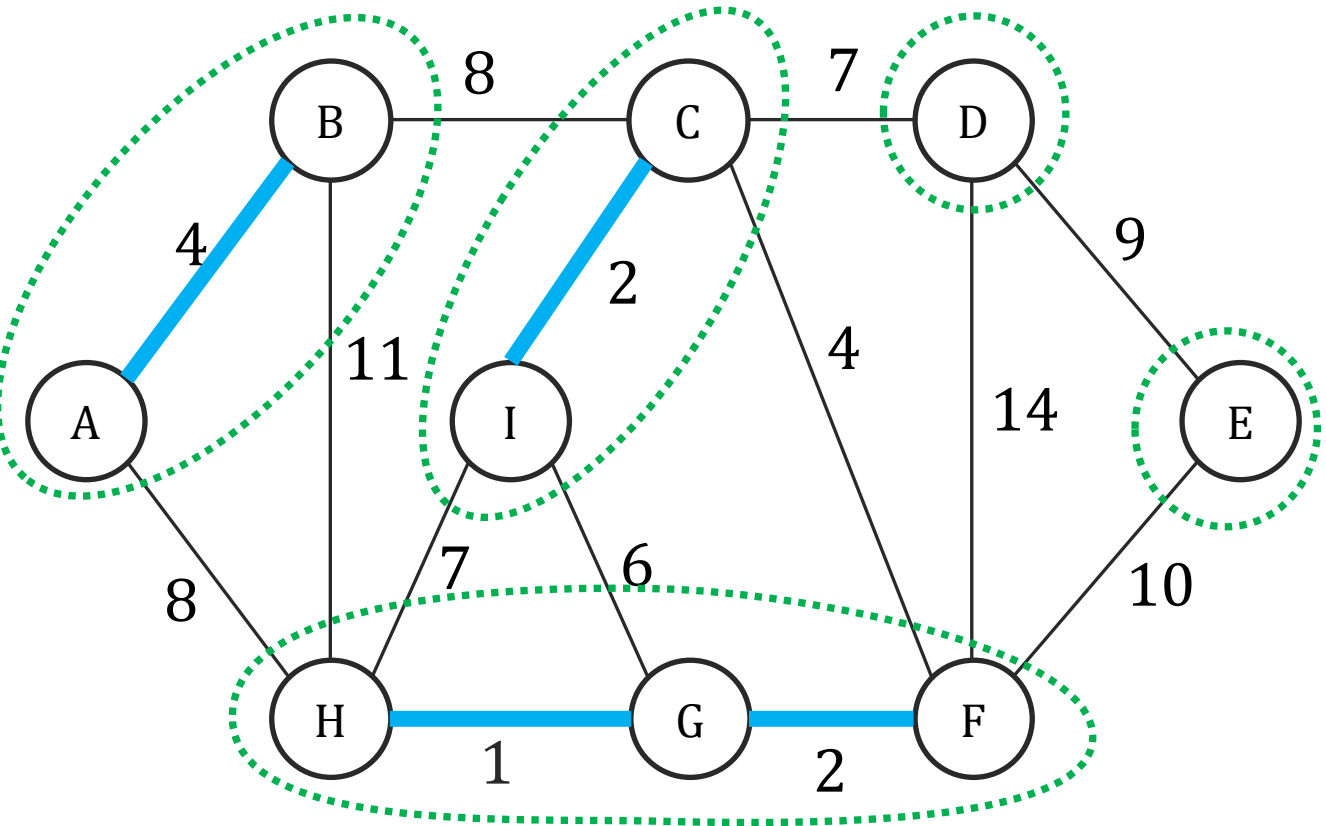


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

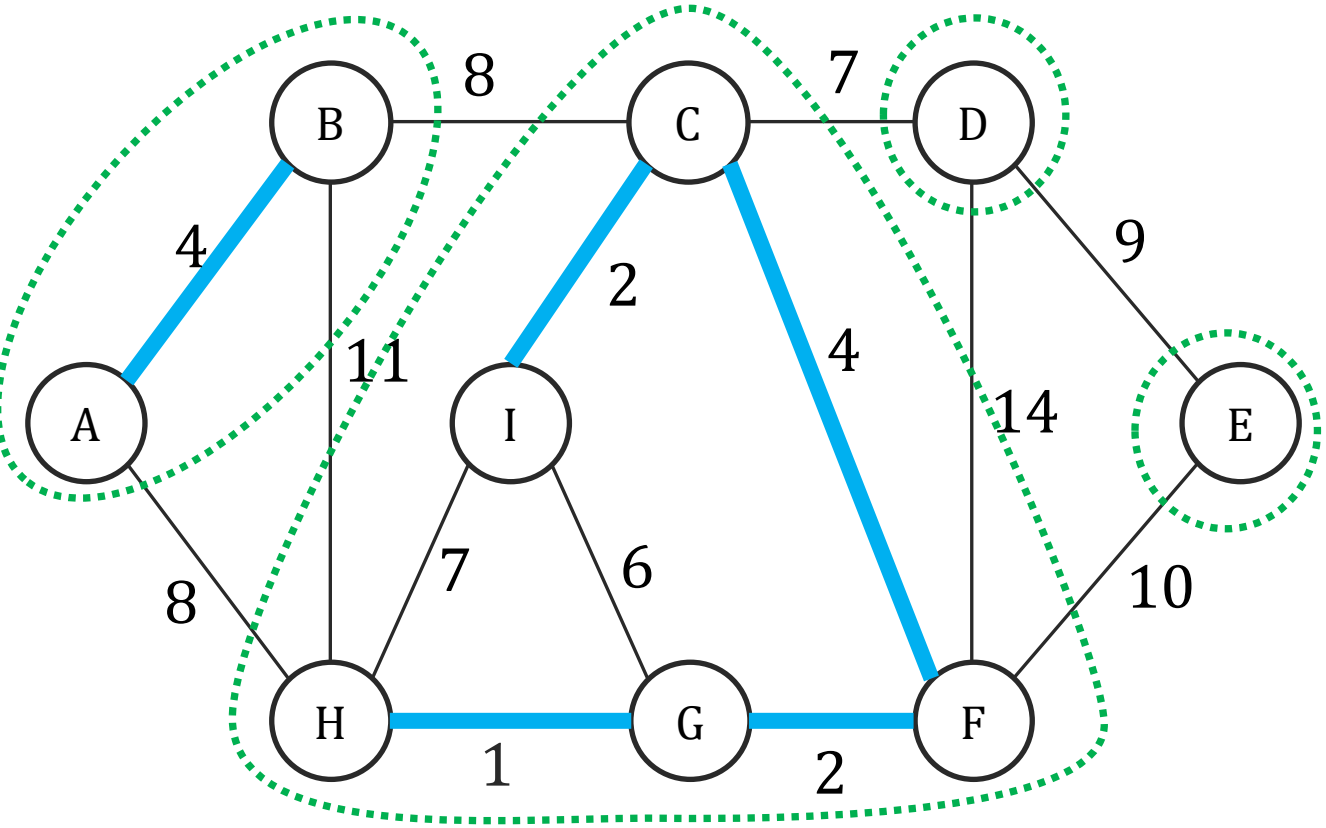


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

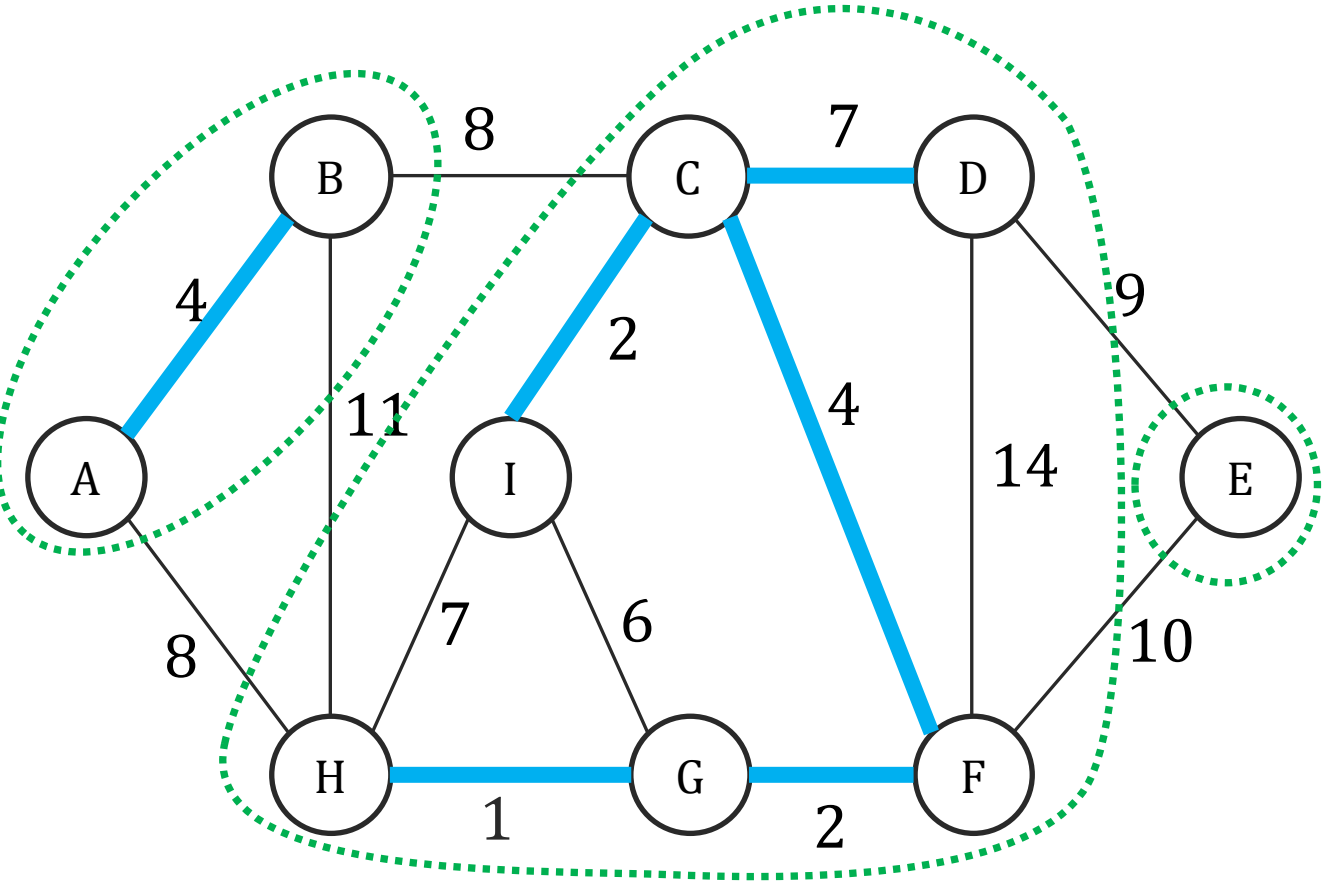


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u, v) \in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

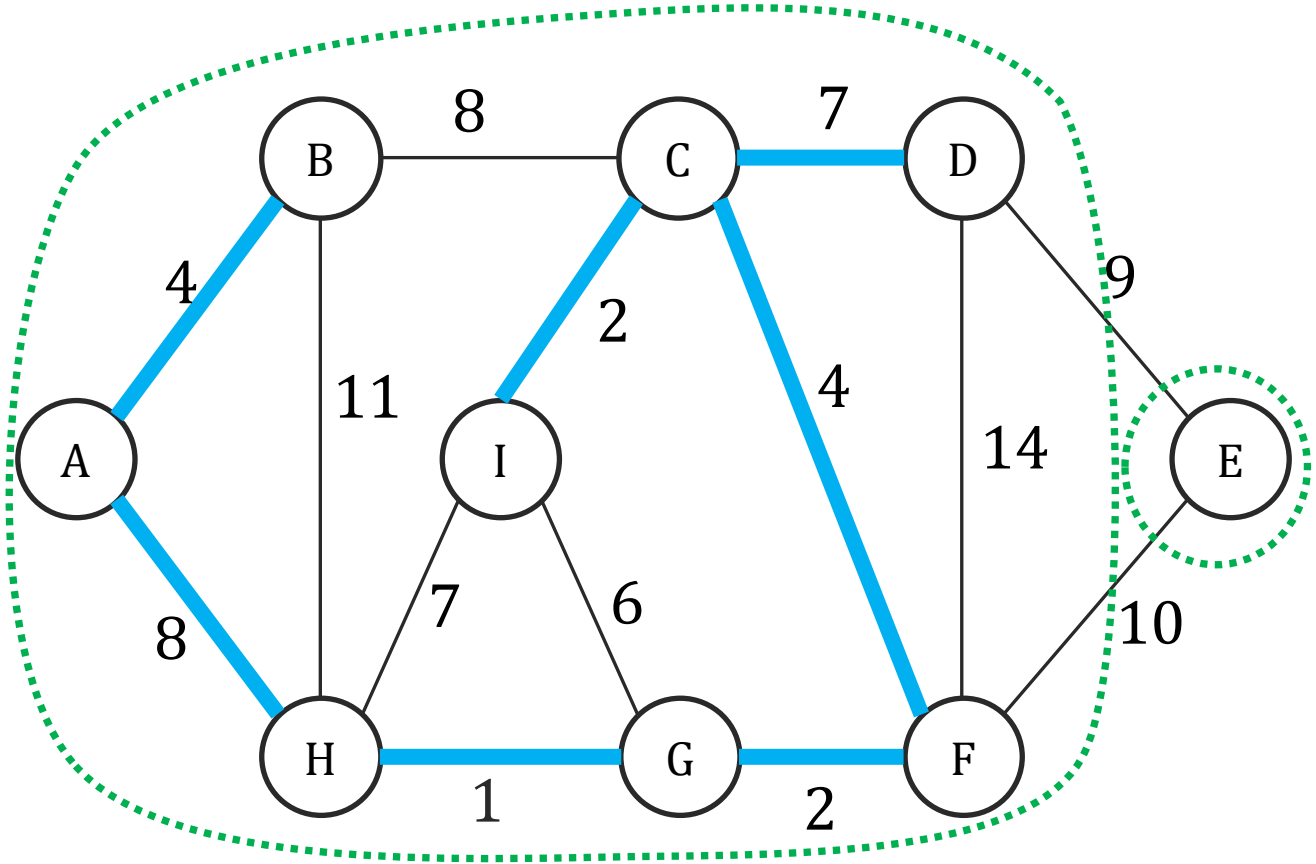


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

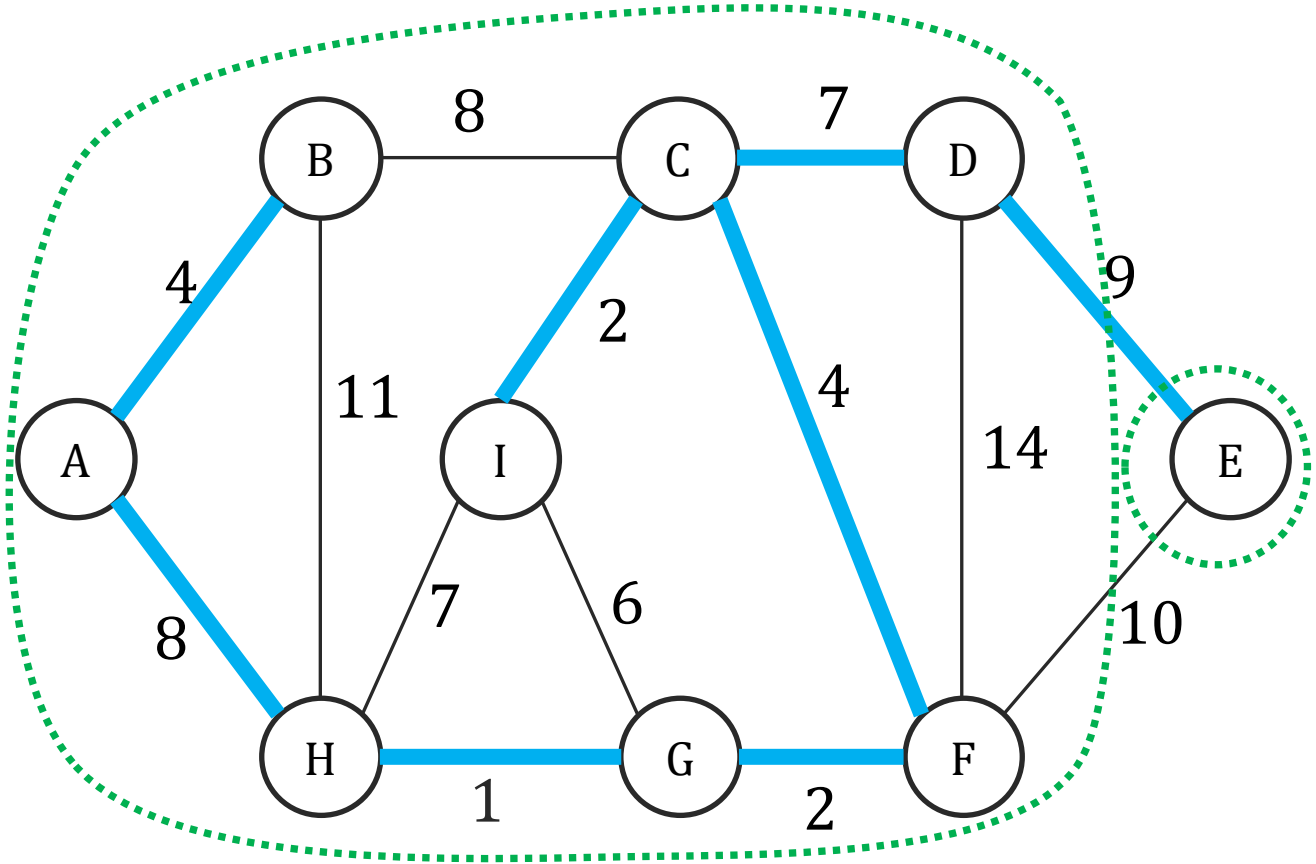


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

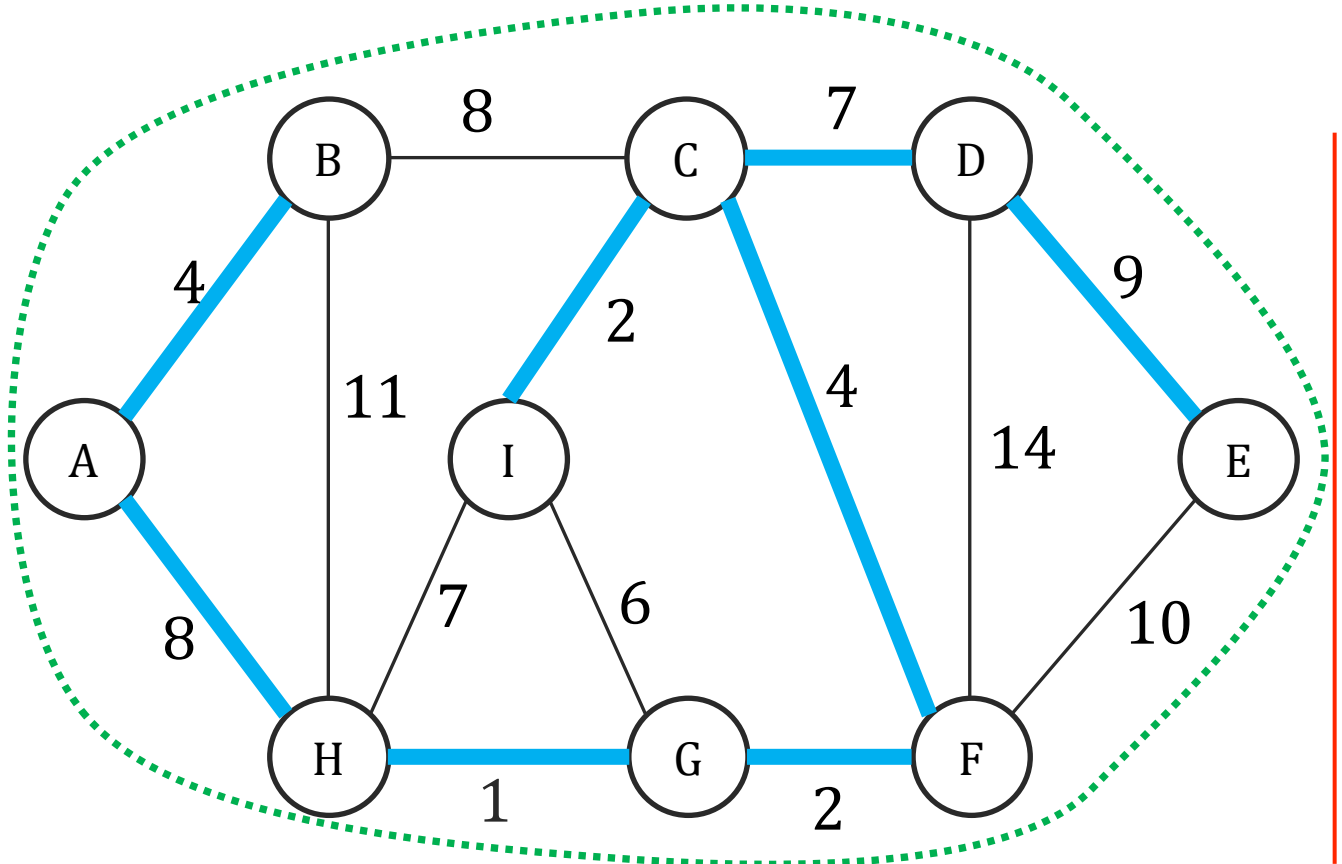


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.



Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Prim's Algorithm
A different MST greedy algorithm

A different greedy algorithm for MSTs

Idea:

- Keep X connected at all times, so S is the connected component representing X.
- Grow a tree greedily by adding the cheapest edge that can grow the tree.

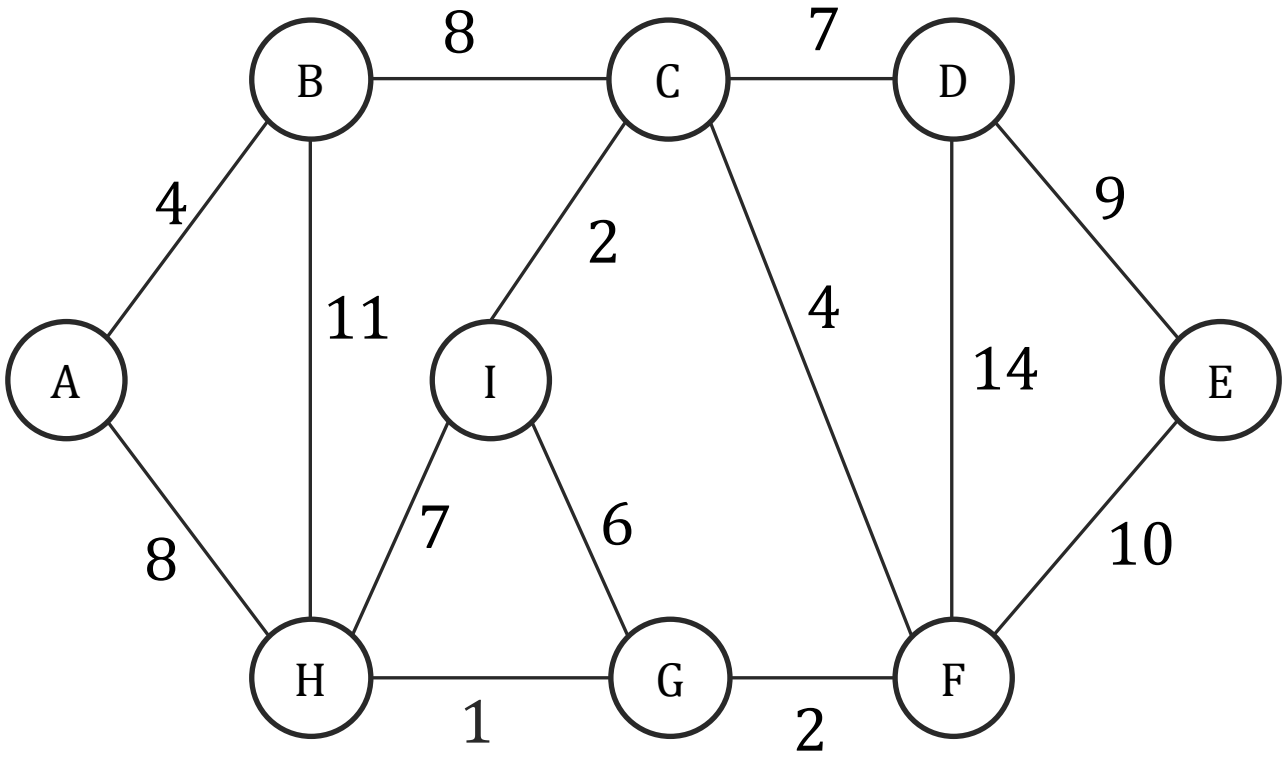
"Cut Property":
If X is a subset of an MST and has no edges from S to $V \setminus S$, then $X \cup \{e\}$ is also a subset of an MST.

Meta Algorithm for MST

$X = \{\}$
Repeat until $|X| = |V| - 1$
    Pick $S \subset V$, s.t. X has no edges from S to $V \setminus S$
    $e \leftarrow$ lightest weight edge from S to $V \setminus S$
    $X \leftarrow X \cup \{e\}$

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

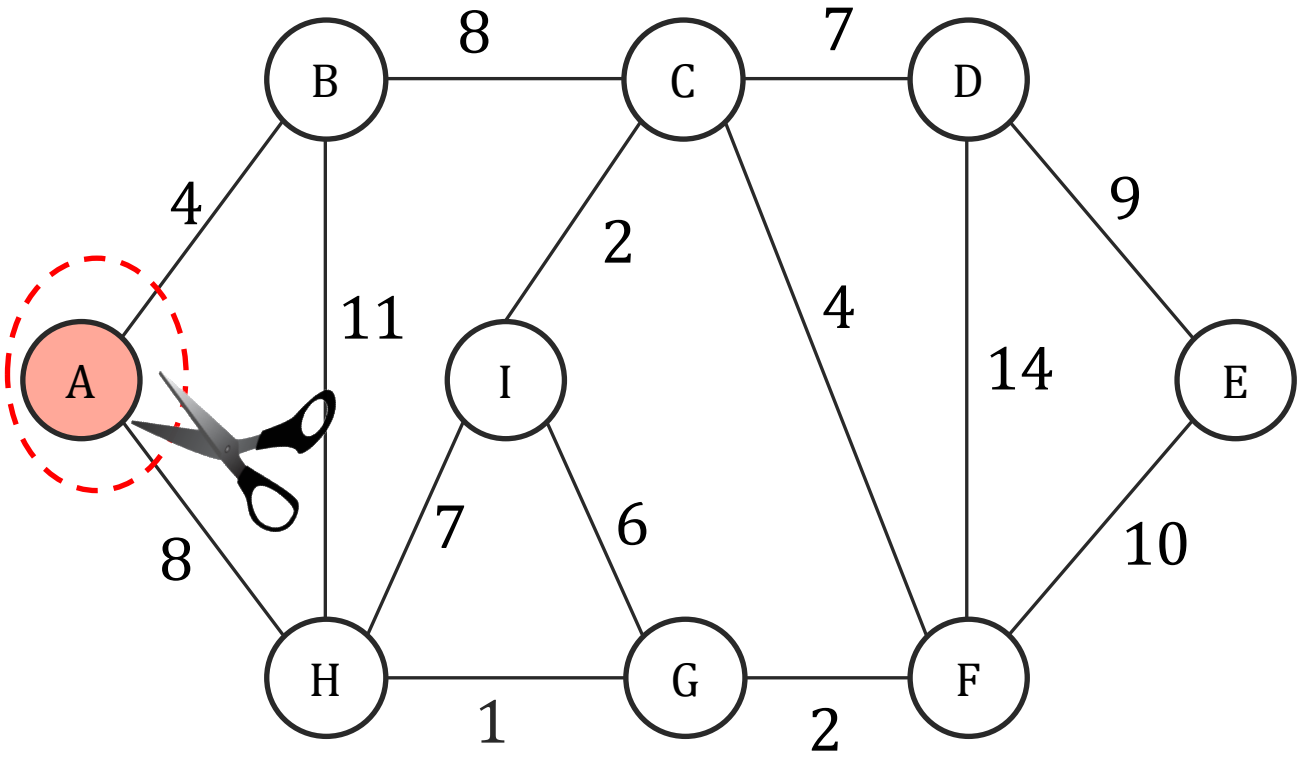
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

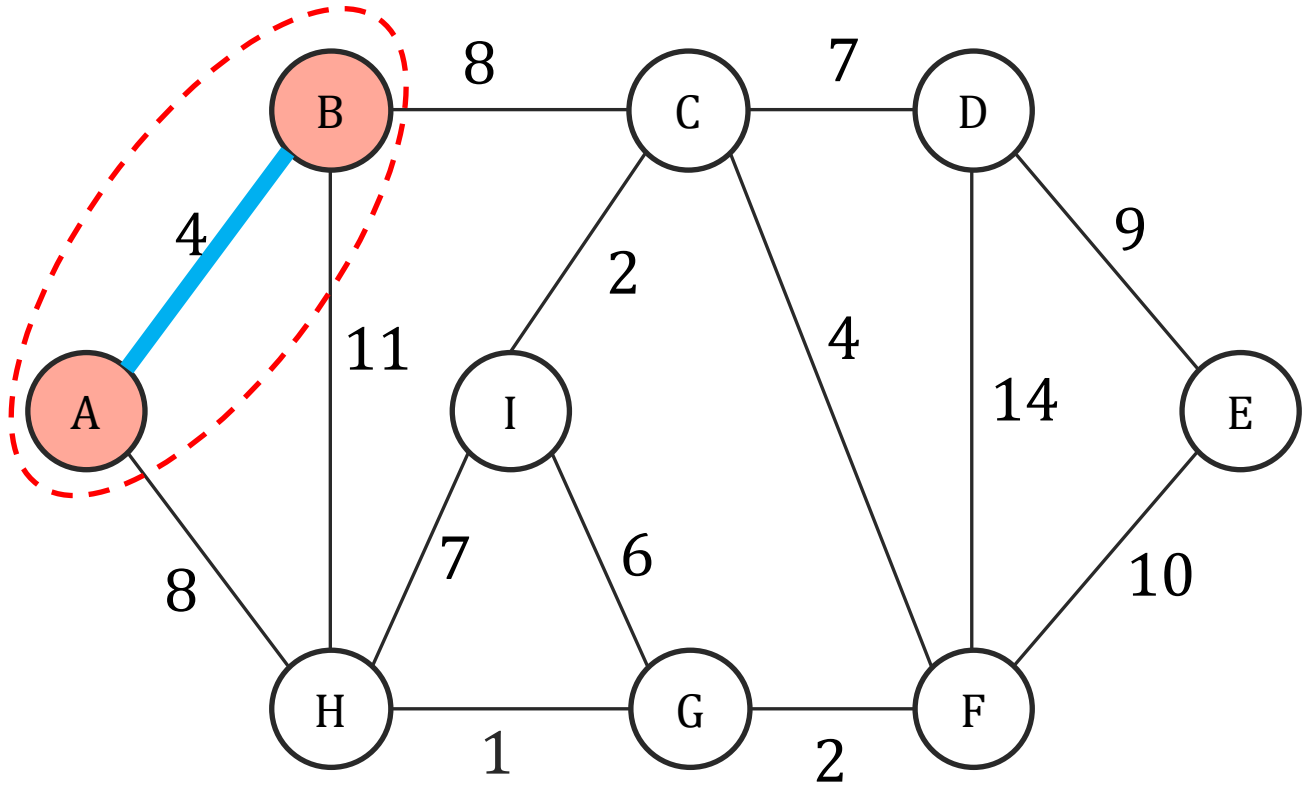
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

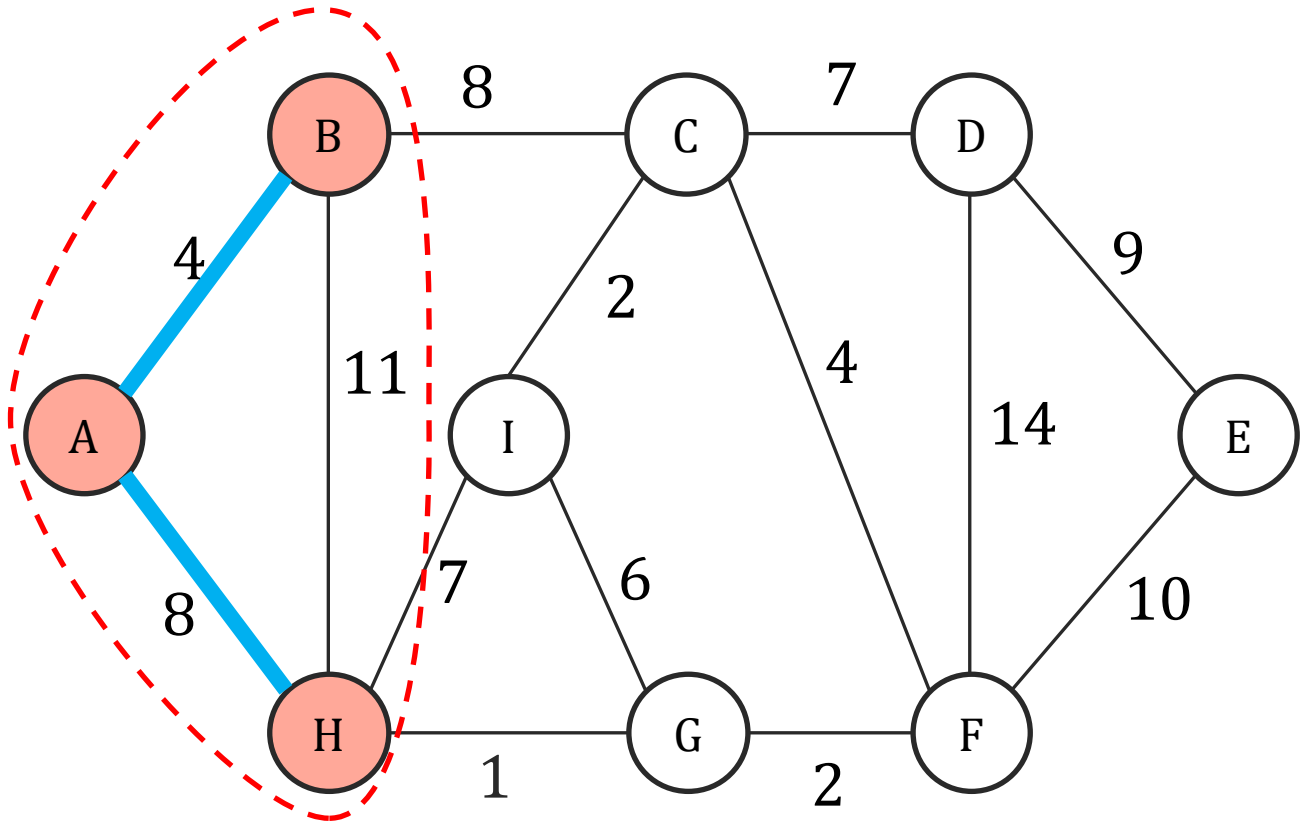
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

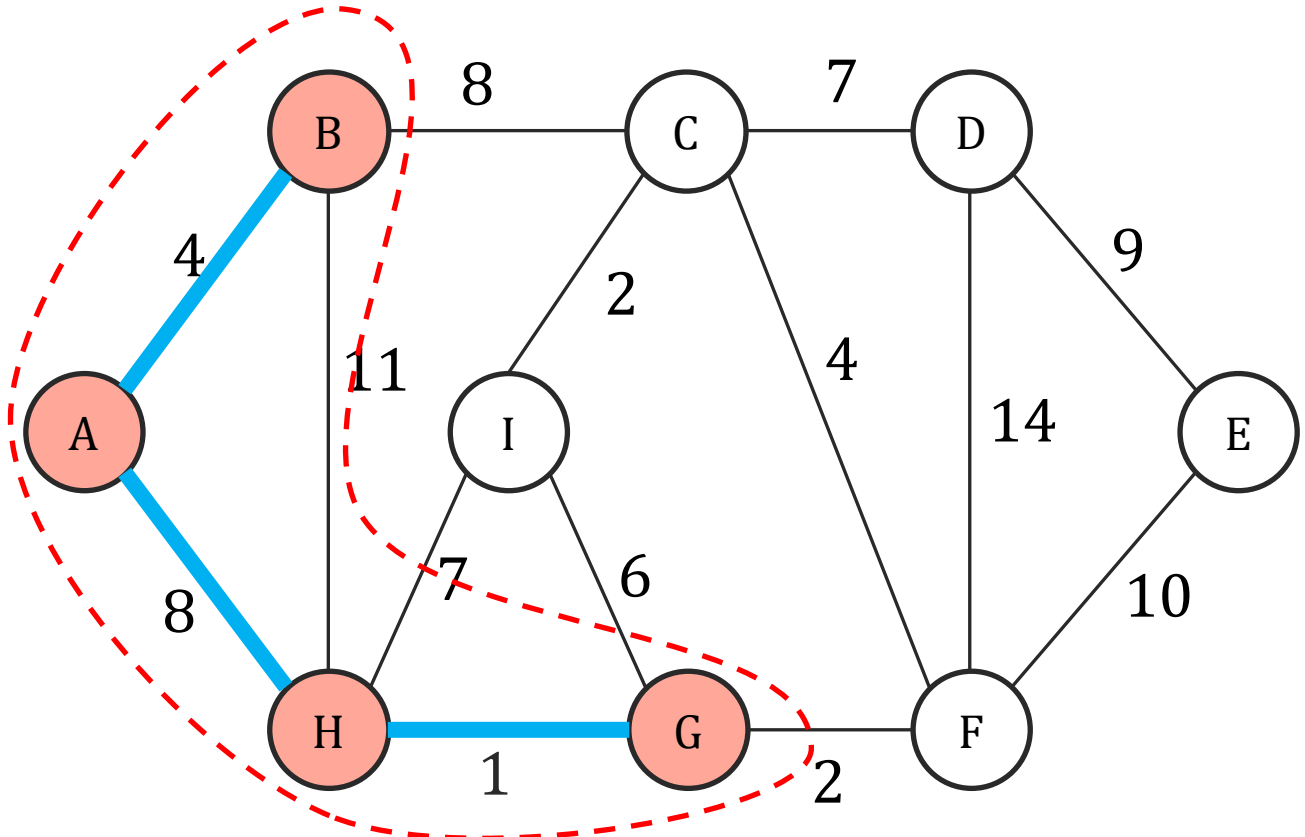
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

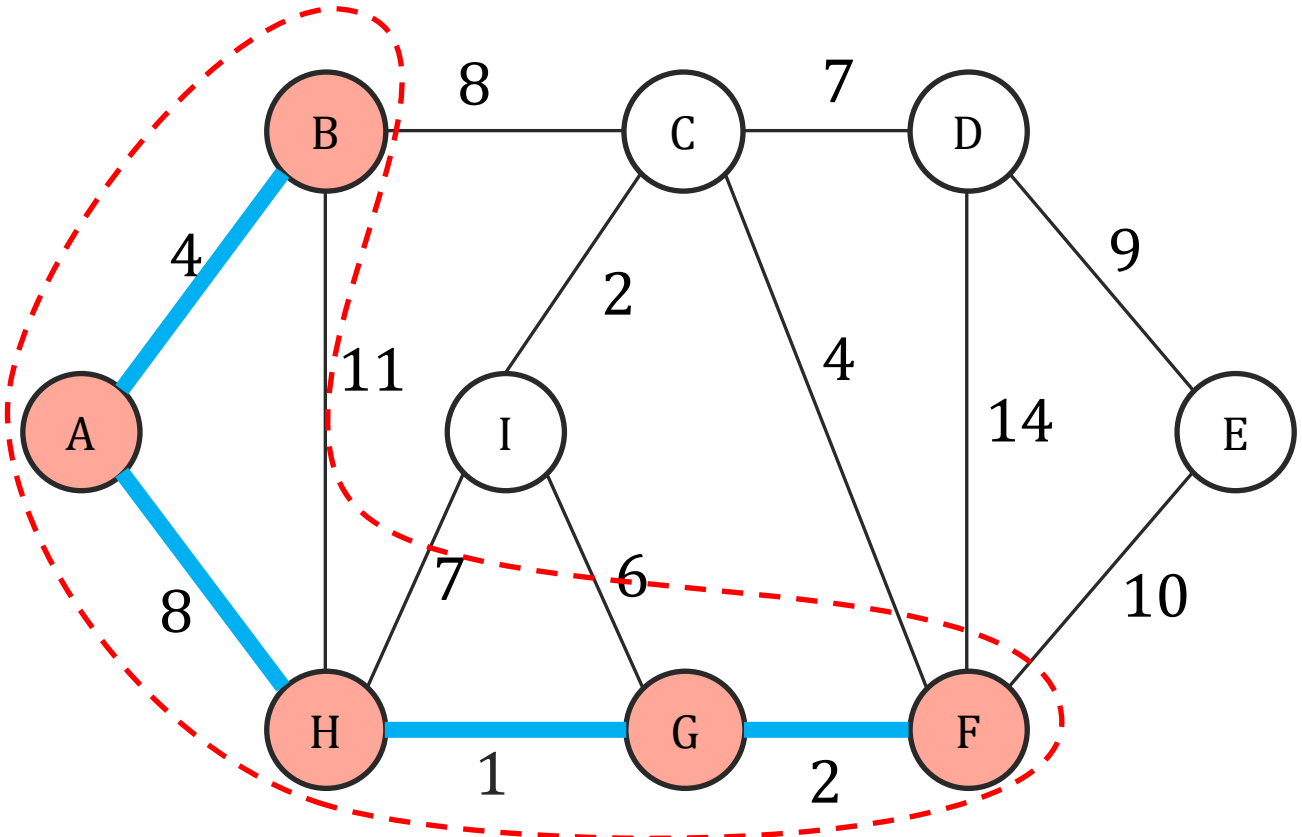
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

    $S \leftarrow S \cup \{v\}$

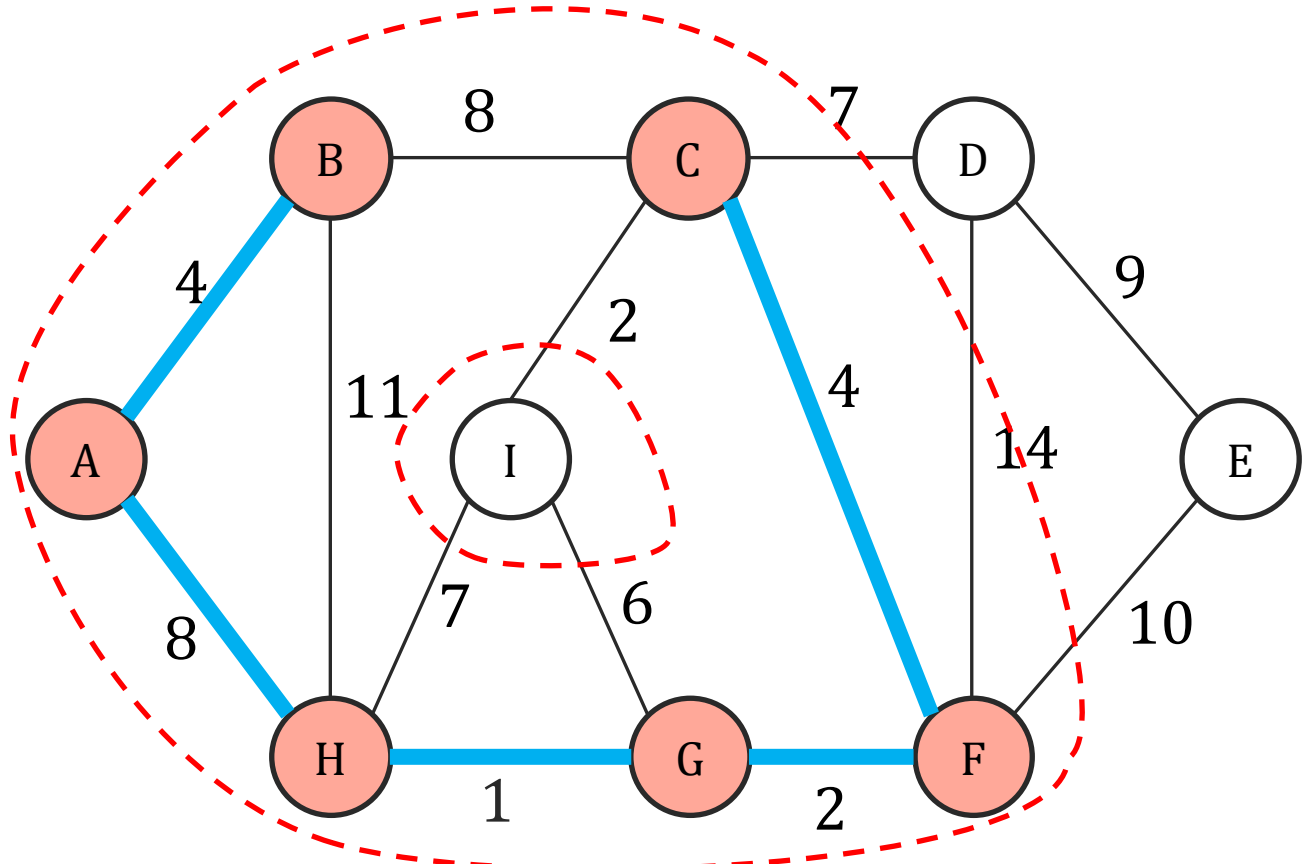
Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.

Here, we choose a donut to visually represent the set S so only edges crossing from S to $V \setminus S$ visually cross the dotted line.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

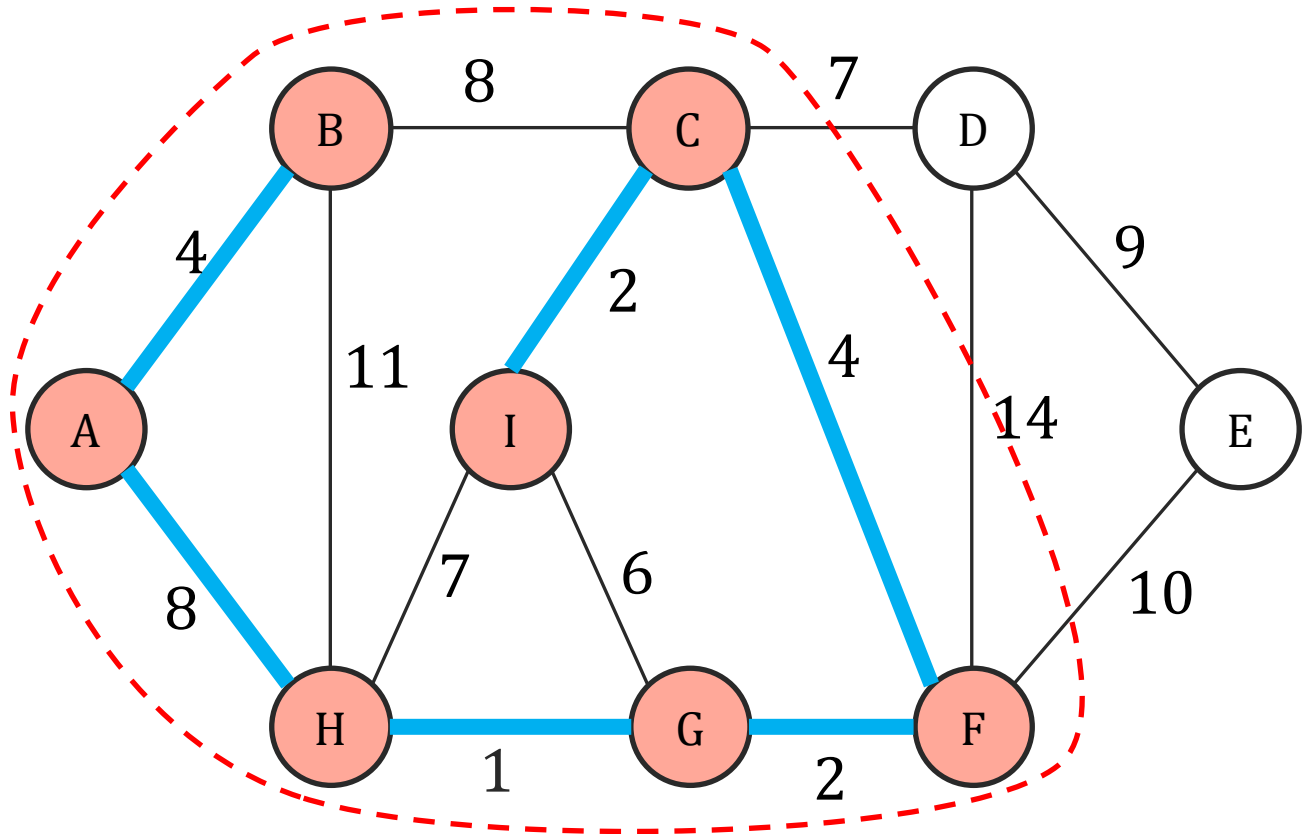
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

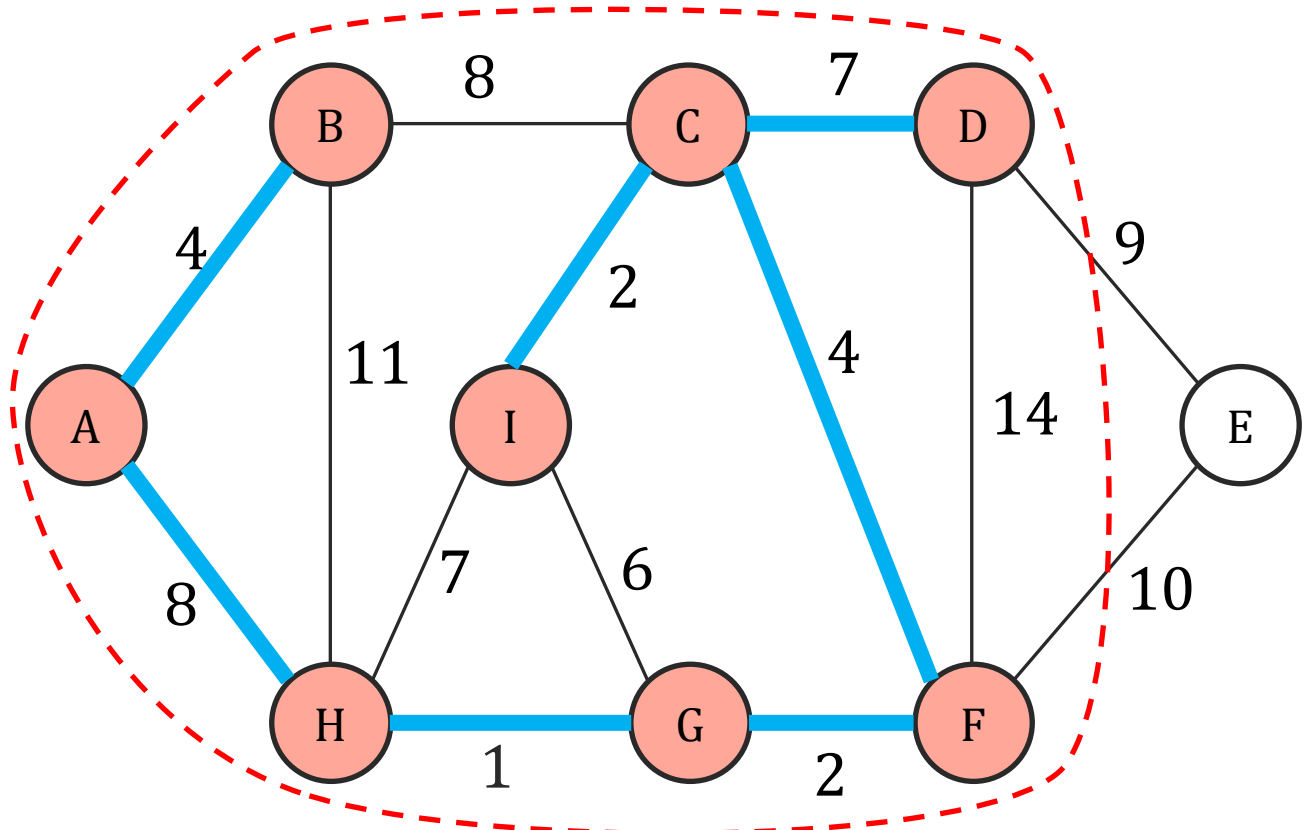
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

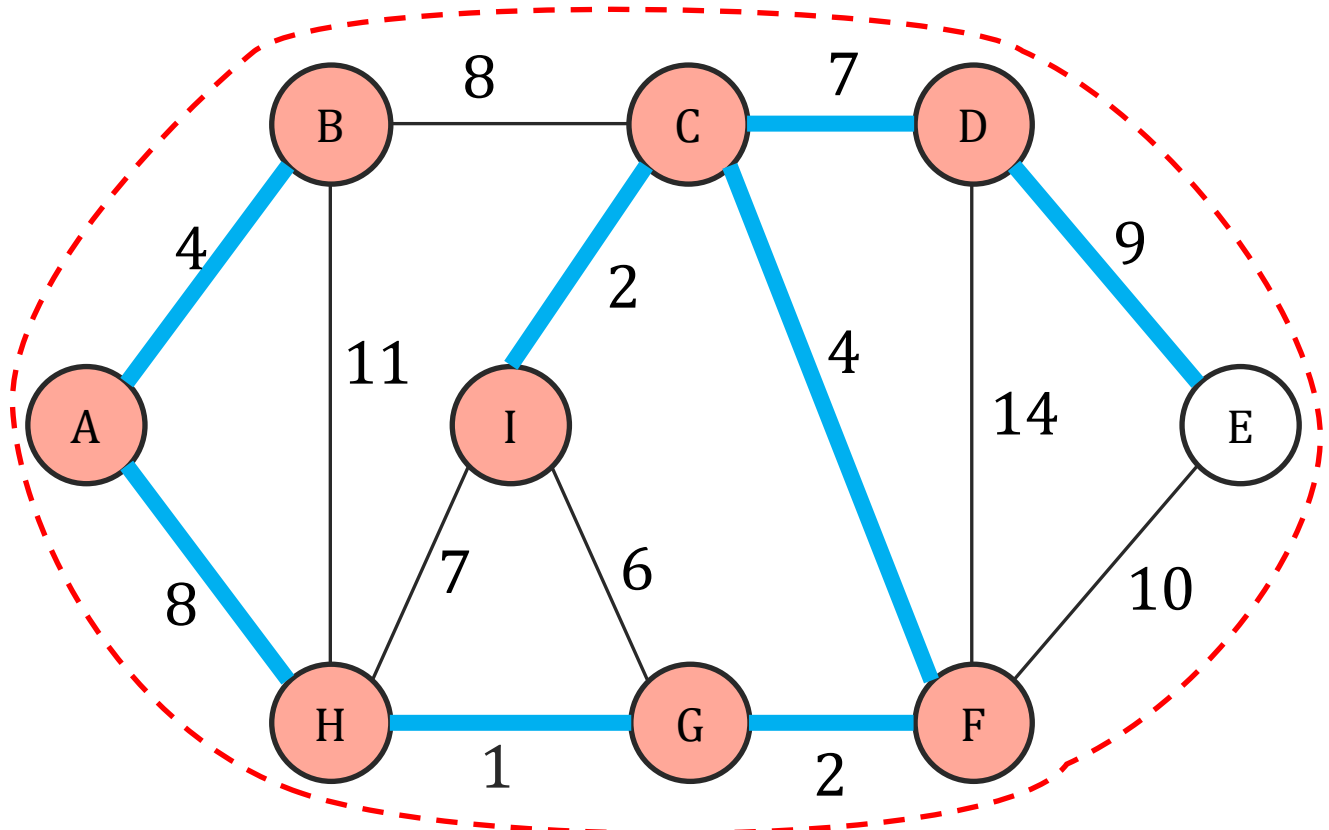
    $S \leftarrow S \cup \{v\}$

Return X

Prim's Algorithm

Grow a tree greedily by adding the cheapest edge that can grow the tree.

Red dotted line indicates the set S.



Prim$(G = (V, E))$

  $S \leftarrow \{A\}$ // an arbitrary node A.

  $X = \{\}$

  while $|X| < |V| - 1$

    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

    $X \leftarrow X \cup \{e\}$

    $S \leftarrow S \cup \{v\}$

Return X

Correctness of Prim's Algorithm

Does Prim's Algorithm return a minimum spanning tree?

- X forms a tree and S refers to the set of vertices connected by this tree.
- Only edges that can "grow" a tree are those that go from S to $V \setminus S$

→At every step, Prim adds the lightest such edge.

So, Prim's algorithm fits the meta algorithm description, so it finds an MST.

Meta Algorithm for MST

$X = \{\}$
Repeat until $|X| = |V| - 1$
    Pick $S \subset V$, s.t. X has no edges from S to $V \setminus S$
    $e \leftarrow$ lightest weight edge from S to $V \setminus S$
    $X \leftarrow X \cup \{e\}$

How to implement Prim's Algorithm

This pseudo-code seems very slow!

At most $n-1$ iterations of this while loop.

Runtime of at most m to go through all the edges and find the lightest.

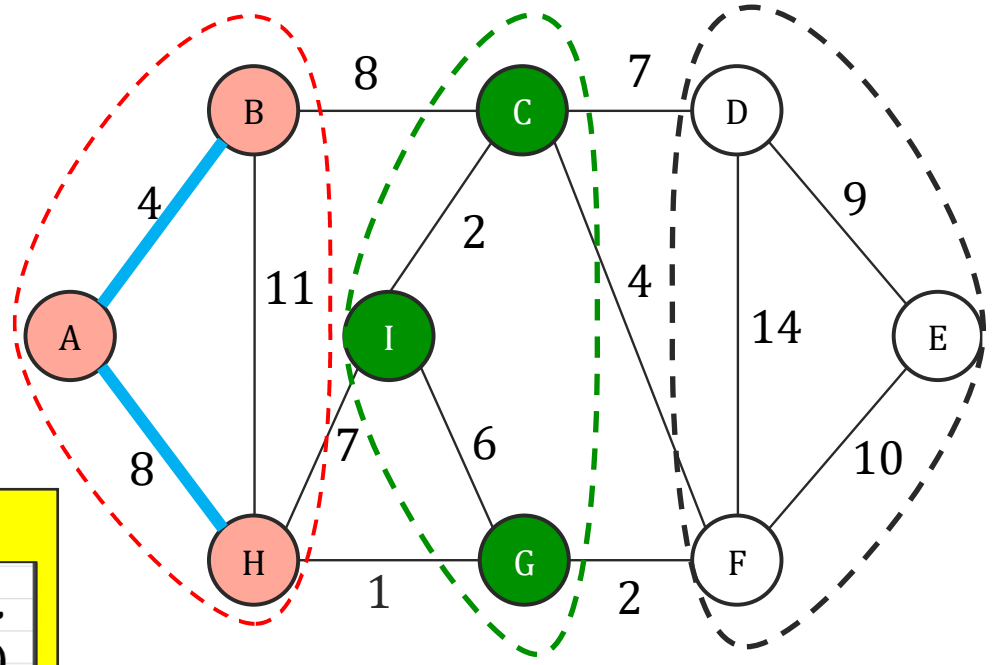
Naively implementing this, take $O(nm)$.

Prim$(G = (V, E))$
  $S \leftarrow \{A\}$ // an arbitrary node A.
  $X = \{\}$
  while $|X| < |V| - 1$
    Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.
    $X \leftarrow X \cup \{e\}$
    $S \leftarrow S \cup \{v\}$
Return X

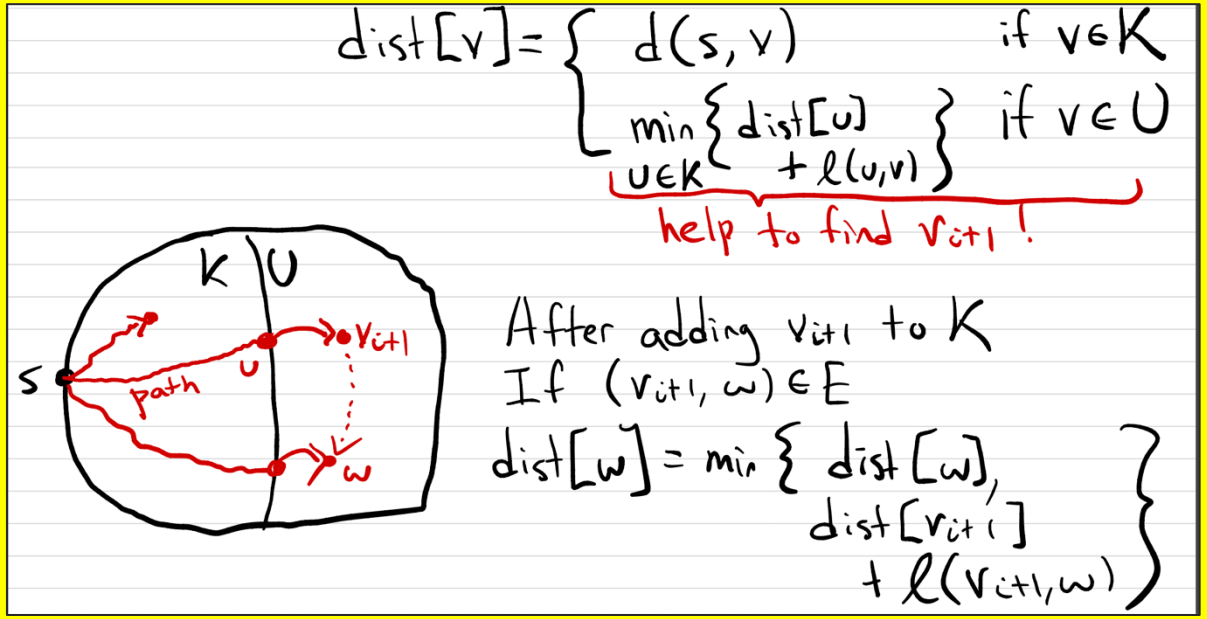
How do we actually implement Prim's Algorithm?

For each vertex $v \in V \setminus S$, we need to keep track of

- Whether v has direct edge the set S of "visited" vertices.
- The cost of the lightest edge connecting v to the set S of "visited" vertices.



Same dilemma in lecture 7!



Implementing Fast Prim's Algorithm

We use the same idea as we did for Dijkstra's, with small changes.

Each vertex has

- cost $dist[v]$ instantiated to ∞ and pointer $prev[v]$ instantiated to null

  →If a neighbor u is added to the visited set S and $dist[v] > w_{(u,v)}$:

  update $dist[v] \leftarrow w_{(u,v)}$.

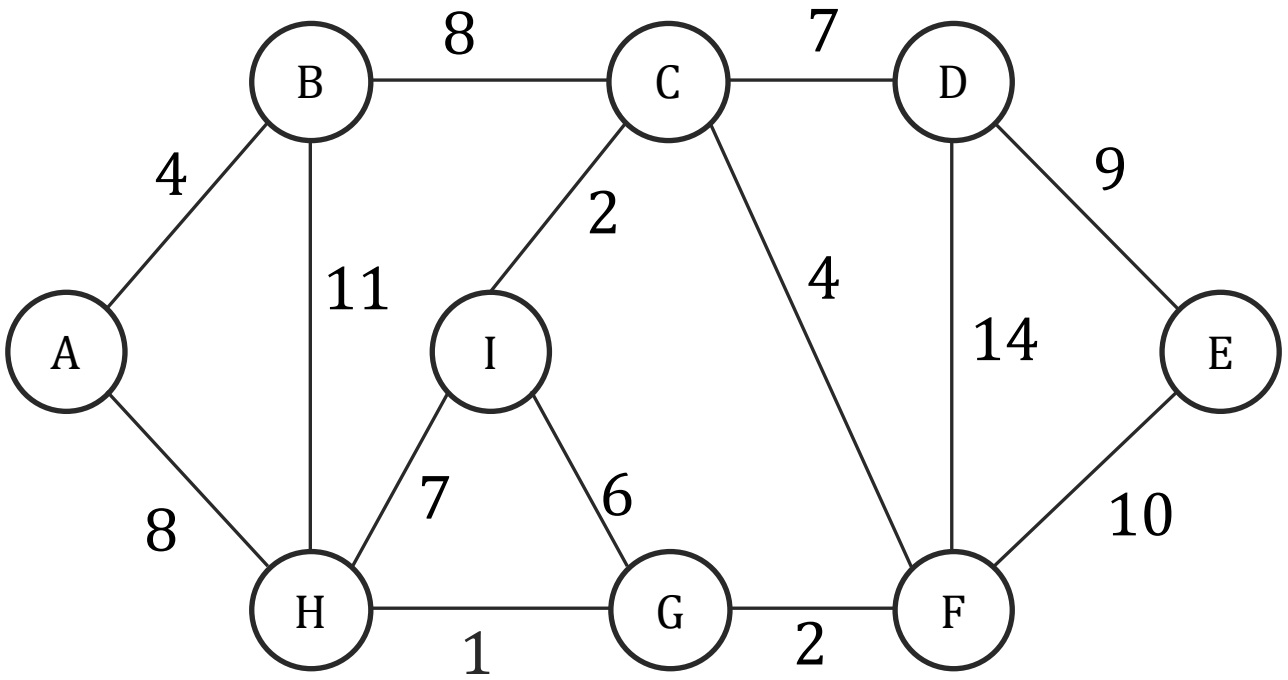
  update $prev[v] \leftarrow u$

How is this different from Dijkstra?

- In Dijkstra, the condition to perform an update and the update accounted for the entire length of s-v path

  → e.g., if $dist[v] > dist[u]+w_{(u,v)}$, then update $dist[v] \leftarrow dist[u]+w_{(u,v)}$
- Here, we only care about distance to the closest visited node, not the entire path.

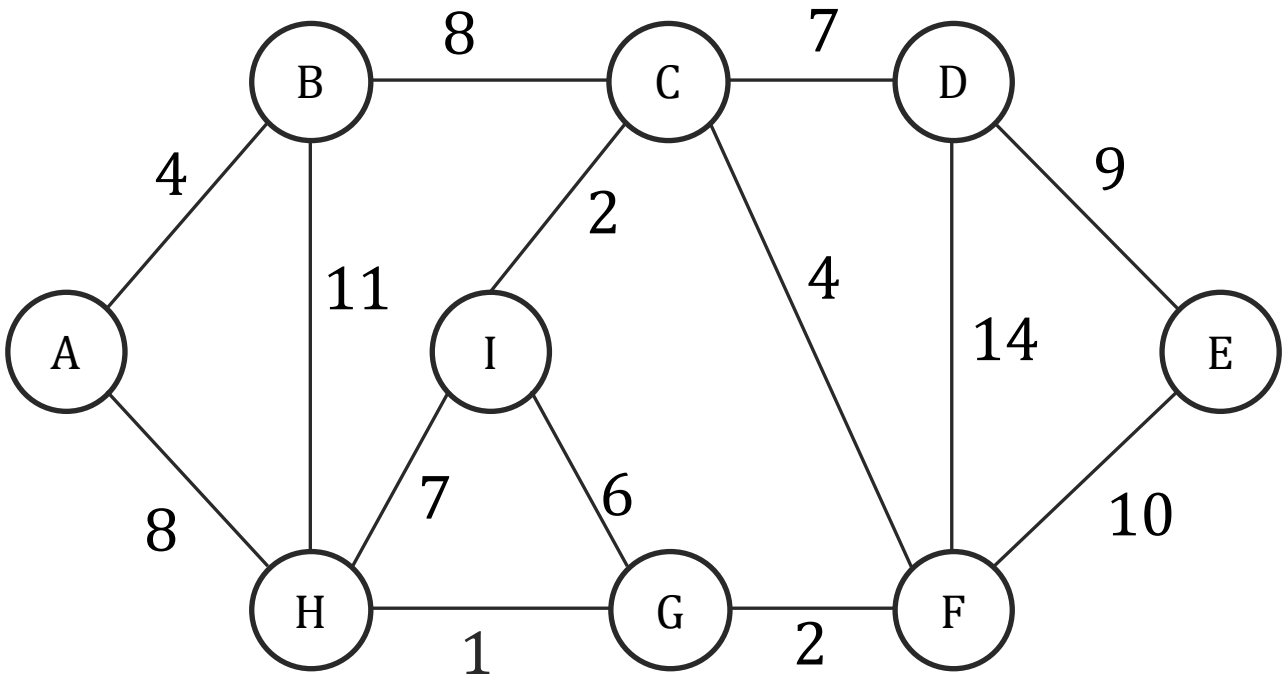
Prim's Algorithm: Efficient Implementation



	A	B	C	D	E	F	G	H	I
$dist$									
$prev$									

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

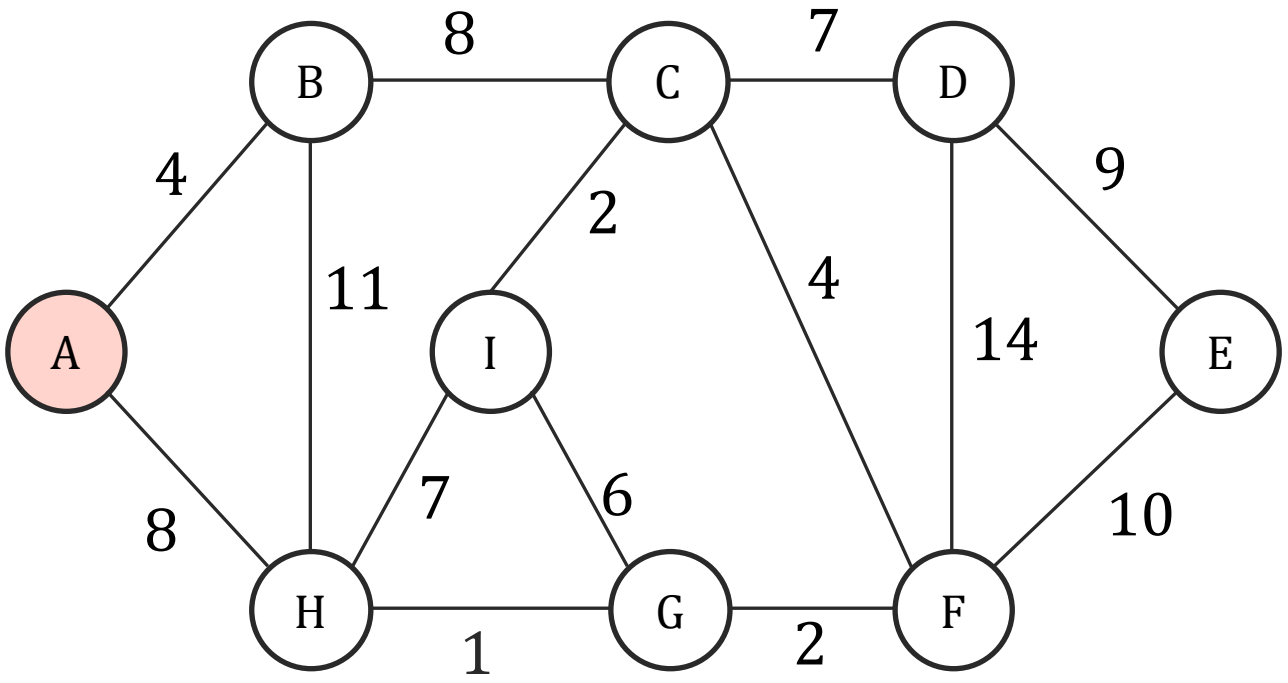


	A	B	C	D	E	F	G	H	I
$dist$	0	∞	∞	∞	∞	∞	∞	∞	∞
$prev$	∅	∅	∅	∅	∅	∅	∅	∅	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q

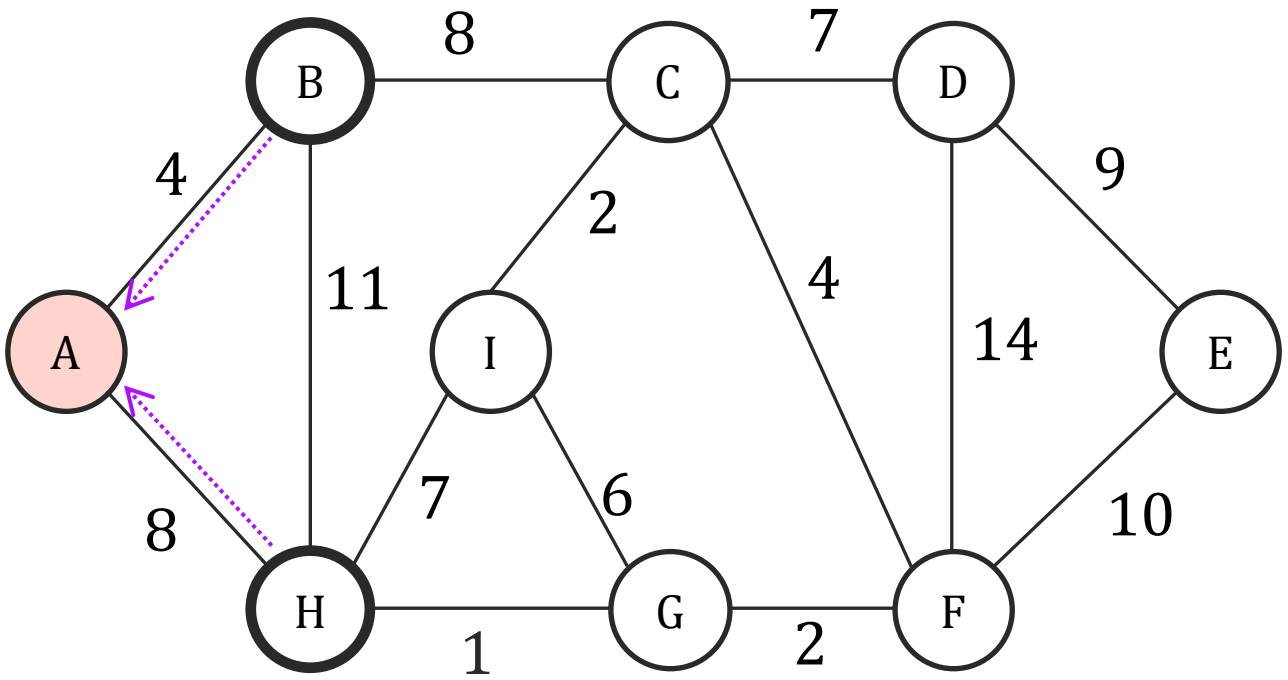


	A	B	C	D	E	F	G	H	I
$dist$	0	∞	∞	∞	∞	∞	∞	∞	∞
$prev$	∅	∅	∅	∅	∅	∅	∅	∅	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

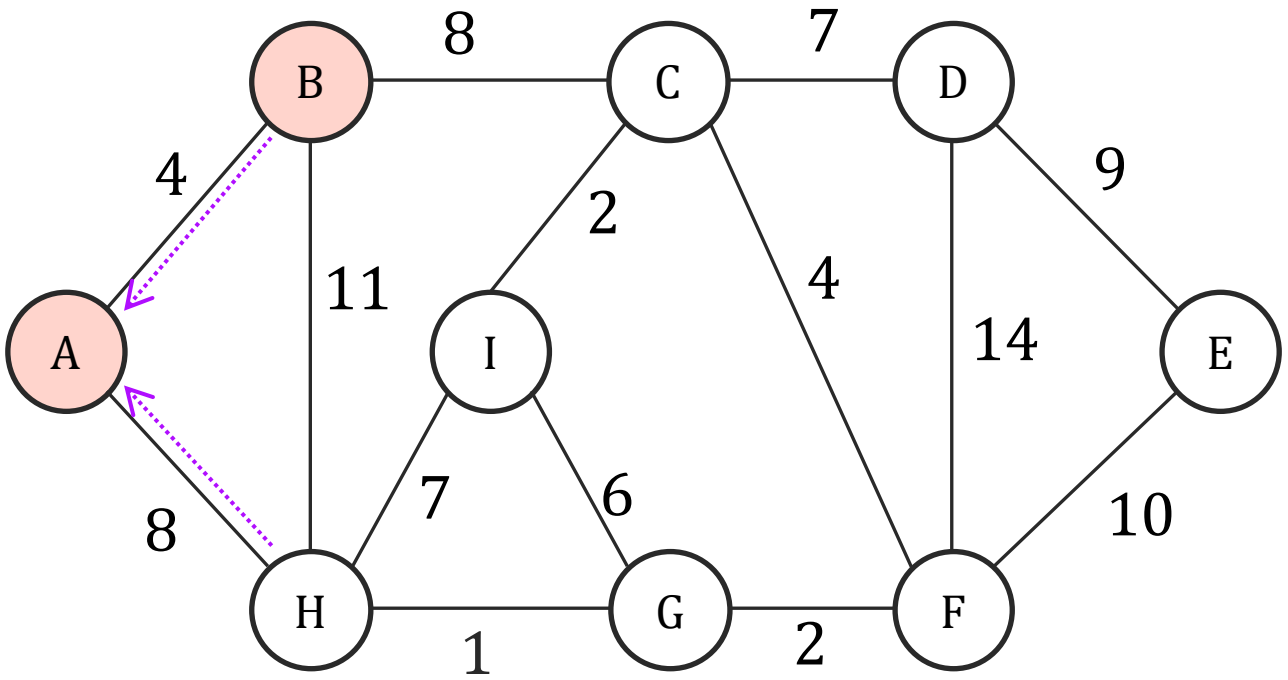


	A	**B**	C	D	E	F	G	**H**	I
dist	0	**4**	∞	∞	∞	∞	∞	**8**	∞
prev	∅	***A***	∅	∅	∅	∅	∅	***A***	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

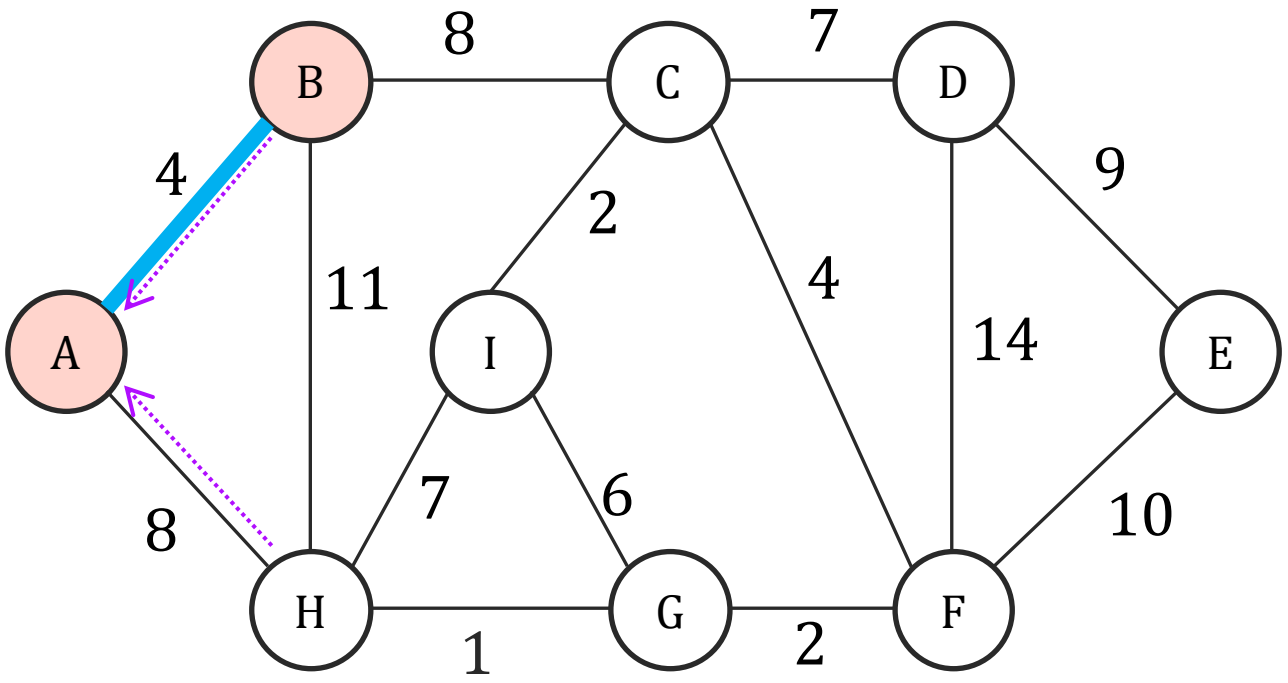


	A	B	C	D	E	F	G	H	I
dist	0	4	∞	∞	∞	∞	∞	8	∞
prev	∅	*A*	∅	∅	∅	∅	∅	*A*	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

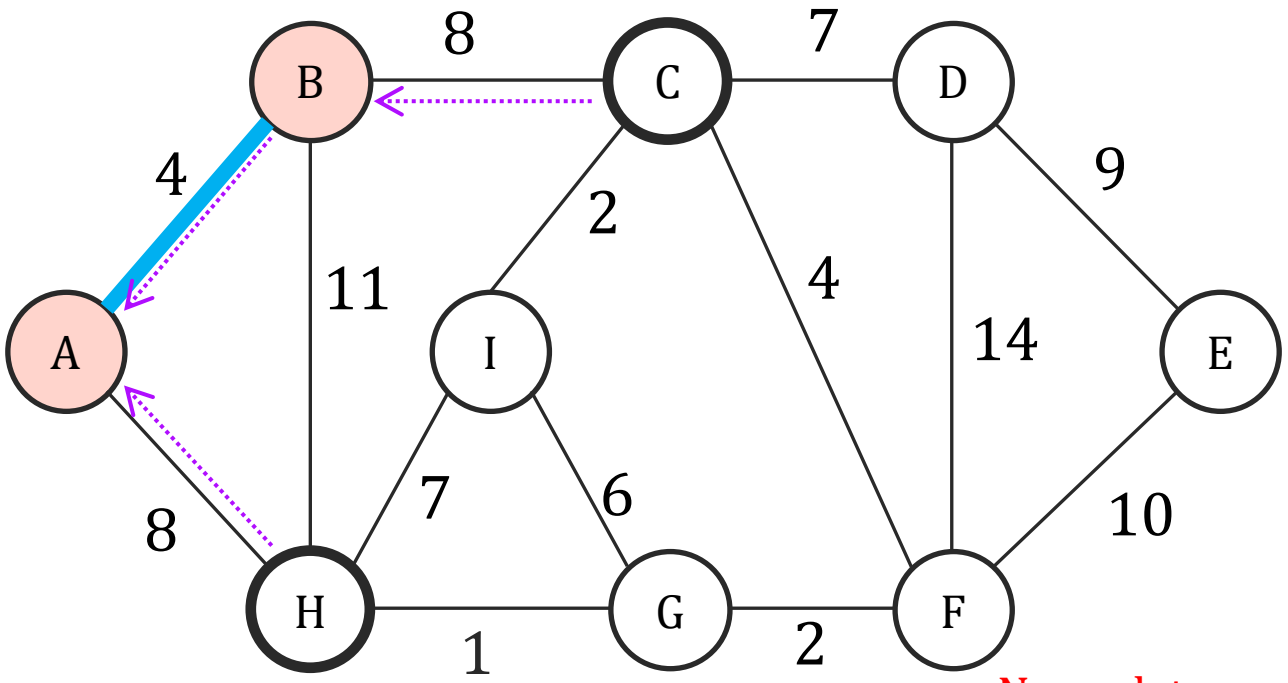


	A	B	C	D	E	F	G	H	I
$dist$	0	4	∞	∞	∞	∞	∞	8	∞
$prev$	∅	A	∅	∅	∅	∅	∅	A	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*



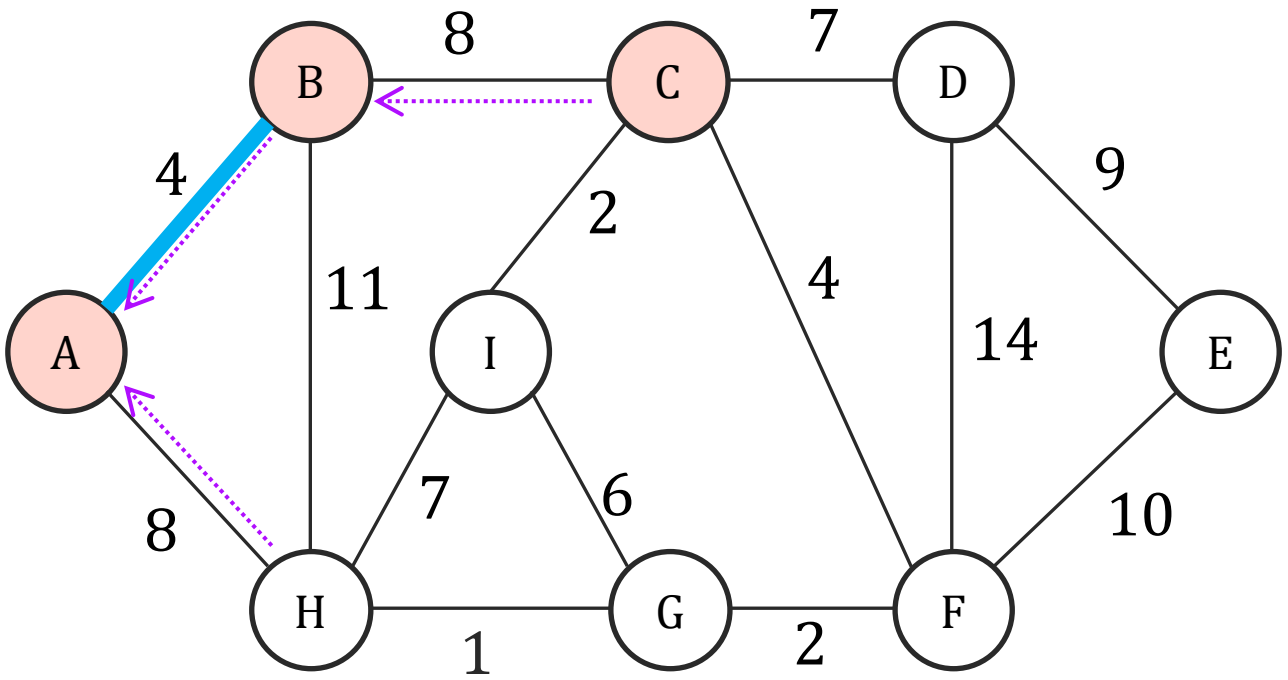
No update

	A	B	**C**	D	E	F	G	**H**	I
dist	0	4	**8**	∞	∞	∞	∞	8	∞
prev	∅	*A*	***B***	∅	∅	∅	∅	*A*	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

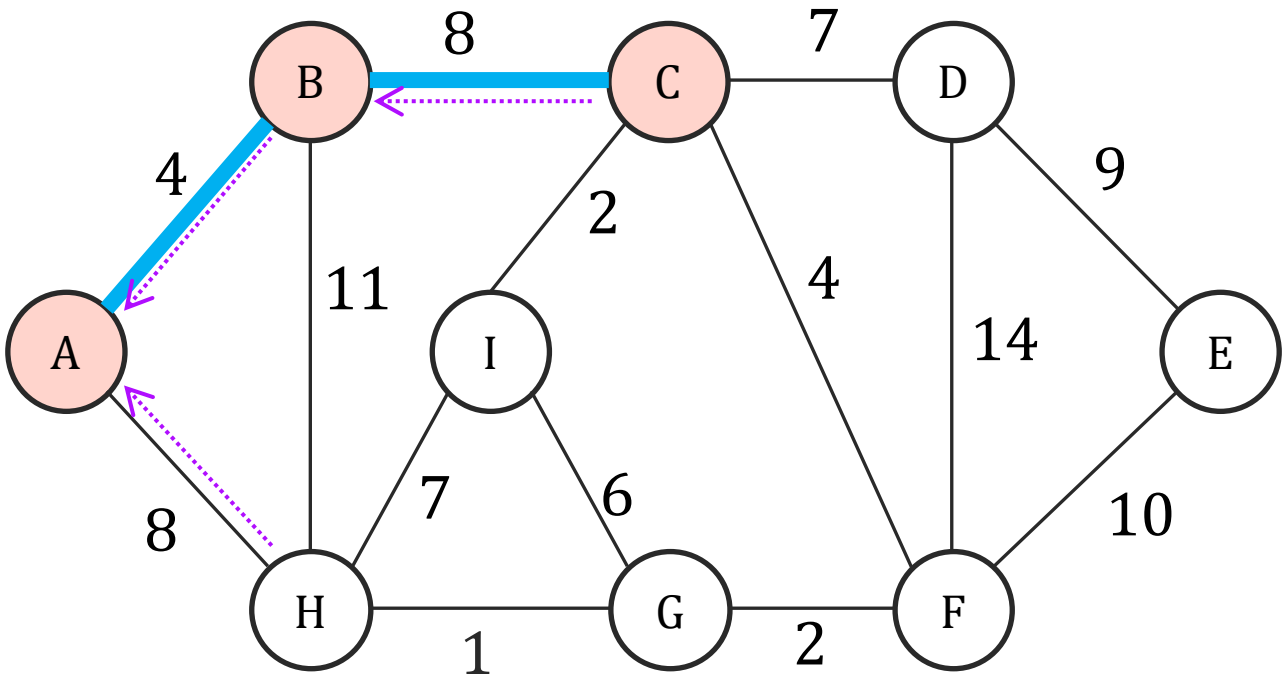


	A	B	C	D	E	F	G	H	I
dist	0	4	8	∞	∞	∞	∞	8	∞
prev	∅	*A*	*B*	∅	∅	∅	∅	*A*	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

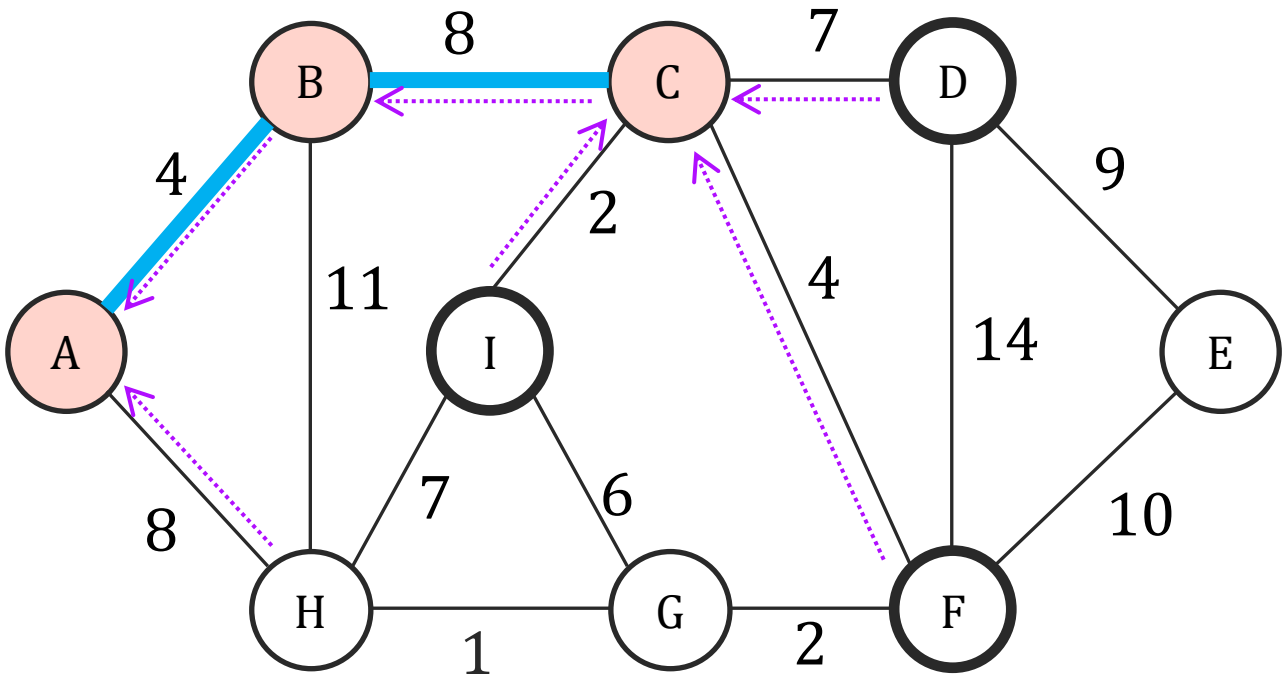


	A	B	C	D	E	F	G	H	I
dist	0	4	8	∞	∞	∞	∞	8	∞
prev	∅	*A*	*B*	∅	∅	∅	∅	*A*	∅

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

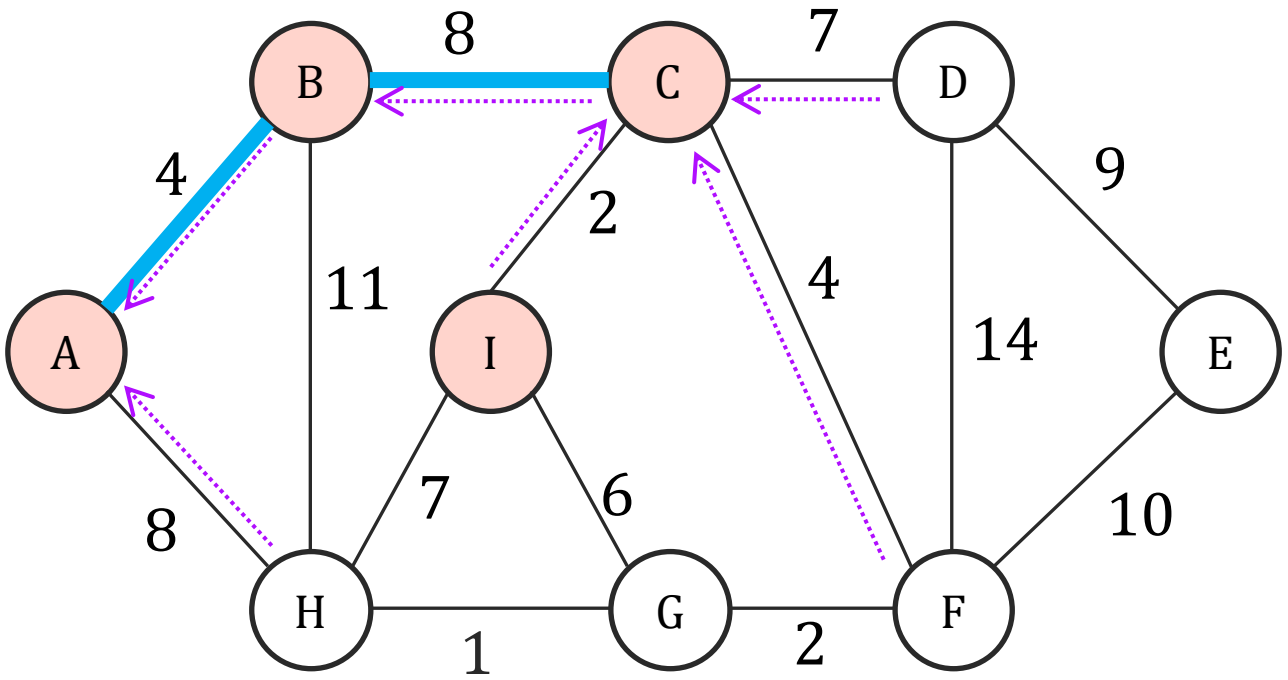


	A	B	C	**D**	E	**F**	G	H	**I**
dist	0	4	8	**7**	∞	**4**	∞	8	**2**
prev	∅	*A*	*B*	***C***	∅	***C***	∅	*A*	***C***

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

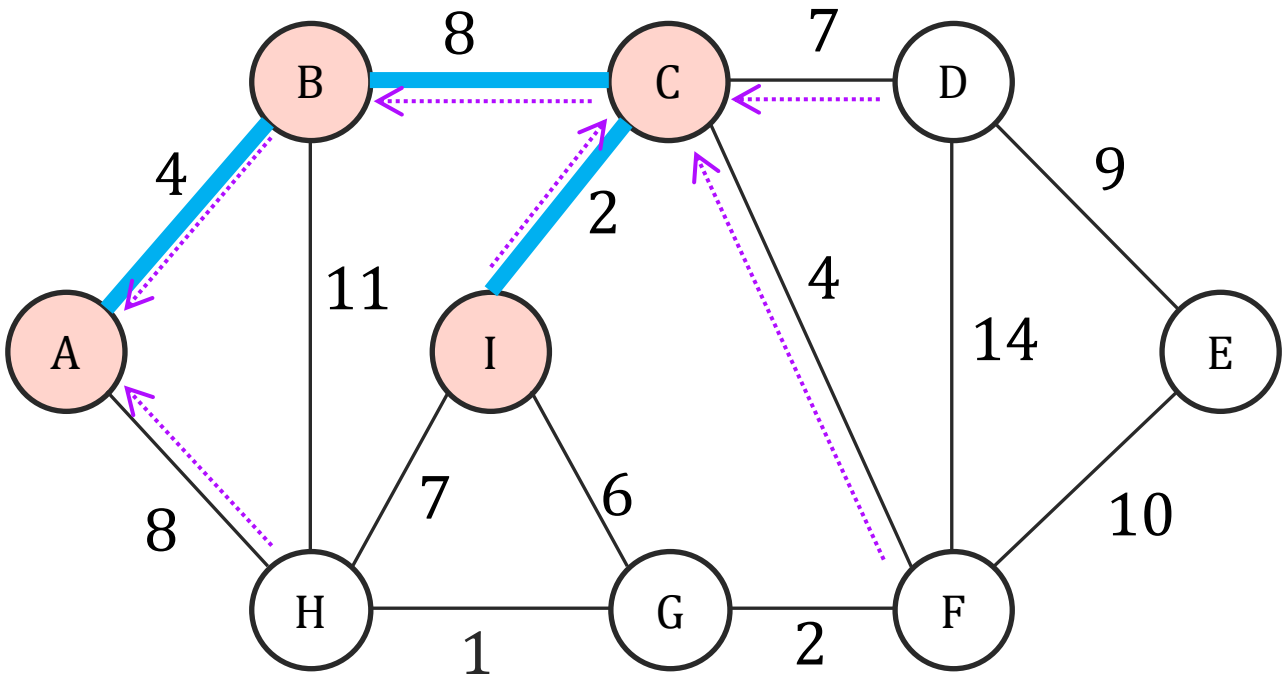


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	∞	4	∞	8	2
$prev$	∅	A	B	C	∅	C	∅	A	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

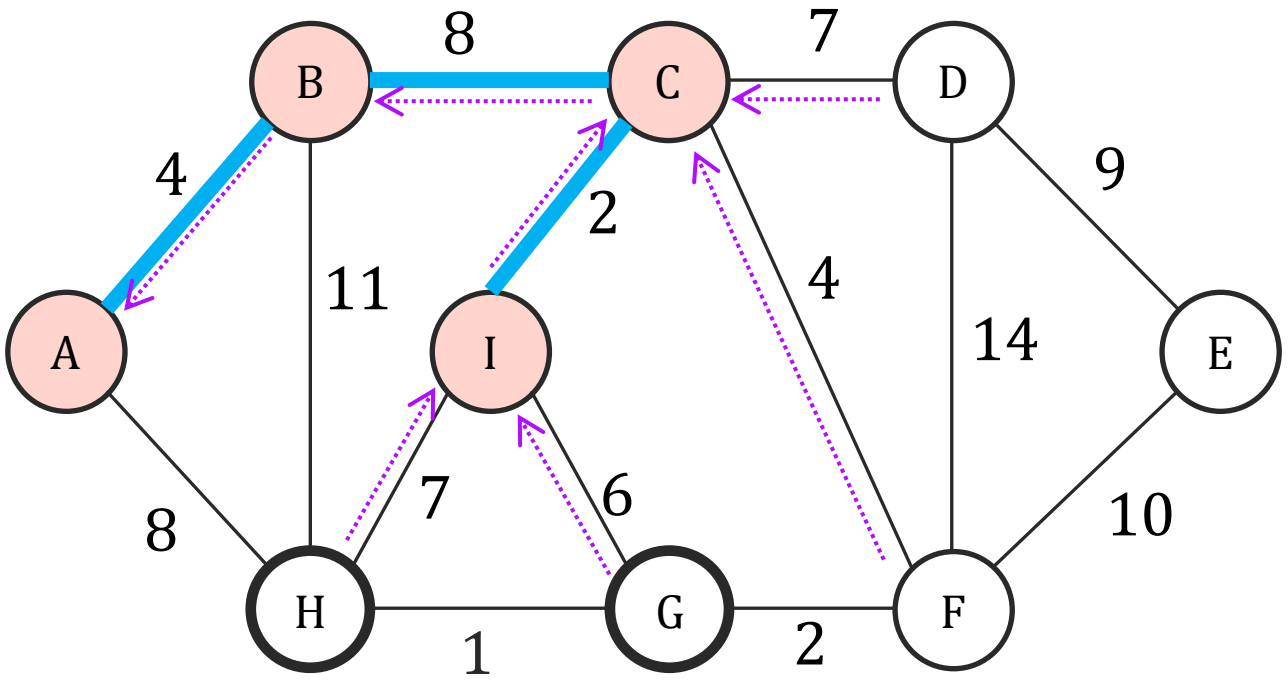


	A	B	C	D	E	F	G	H	I
dist	0	4	8	7	∞	4	∞	8	2
prev	∅	*A*	*B*	*C*	∅	*C*	∅	*A*	*C*

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

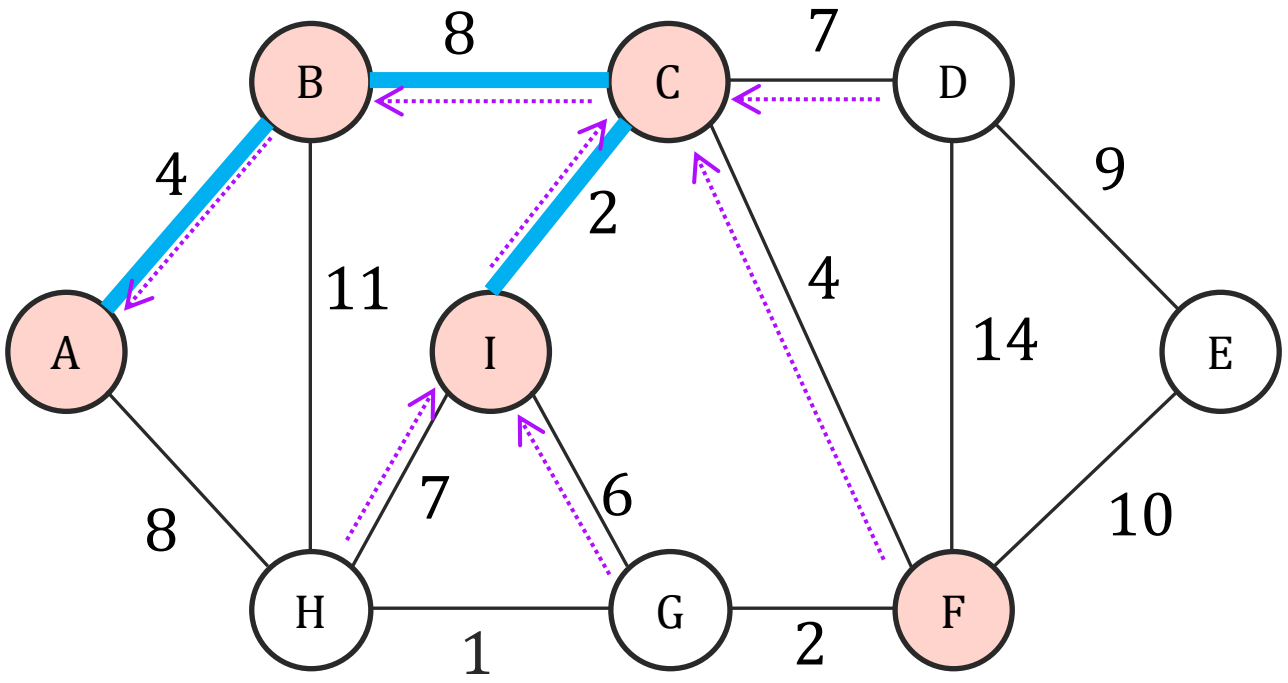


	A	B	C	D	E	F	**G**	**H**	I
dist	0	4	8	7	∞	4	**6**	**7**	2
prev	∅	*A*	*B*	*C*	∅	*C*	***I***	***I***	*C*

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

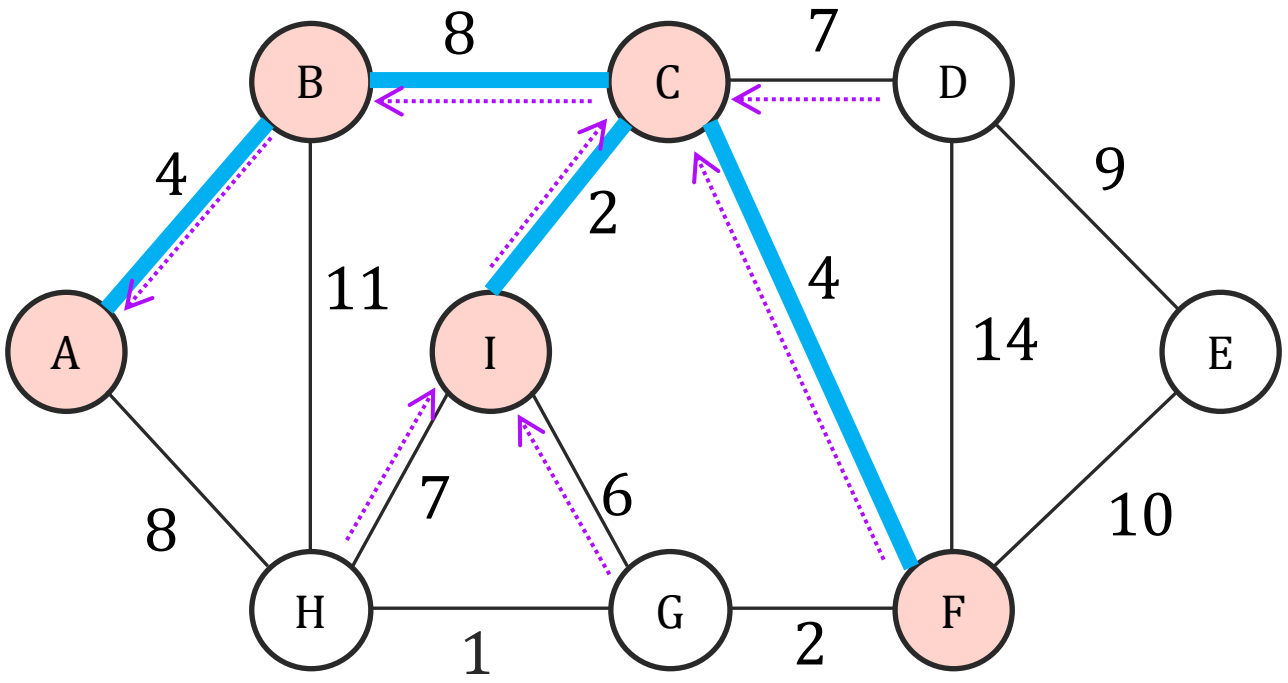


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	∞	4	6	7	2
$prev$	∅	A	B	C	∅	C	I	I	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

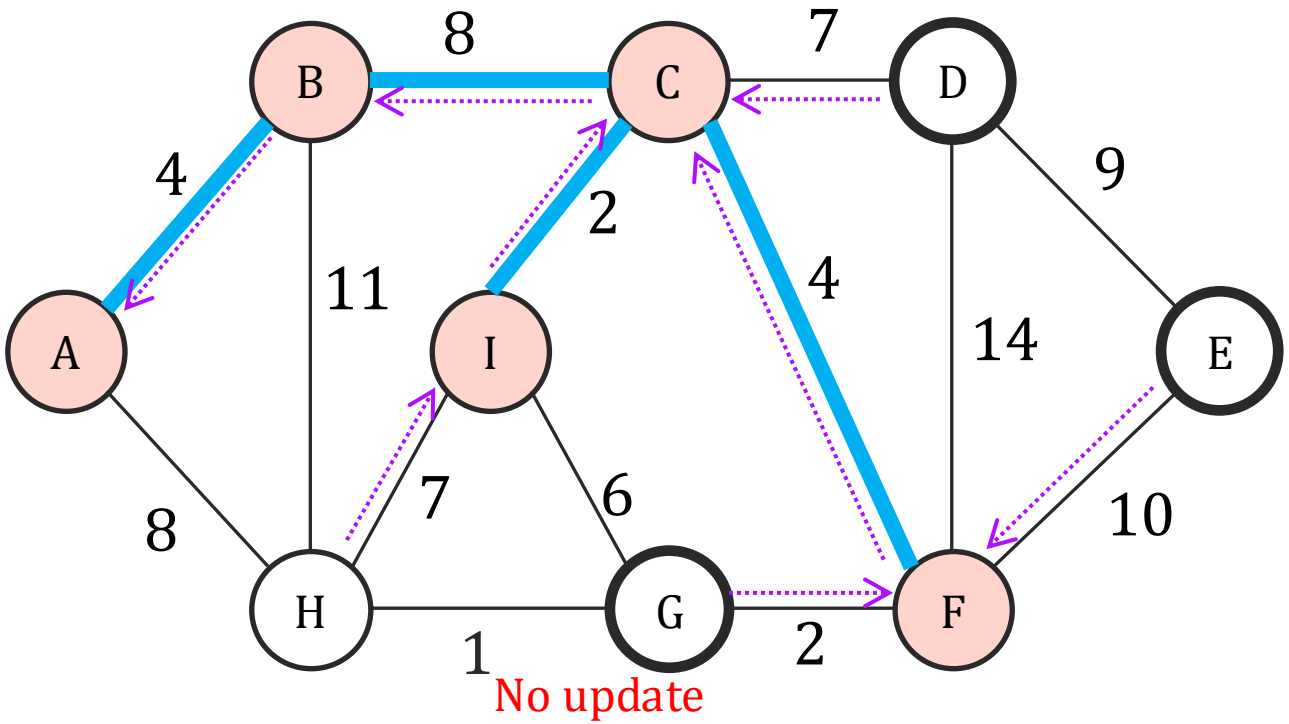


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	∞	4	6	7	2
$prev$	$\emptyset$	A	B	C	$\emptyset$	C	I	I	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*



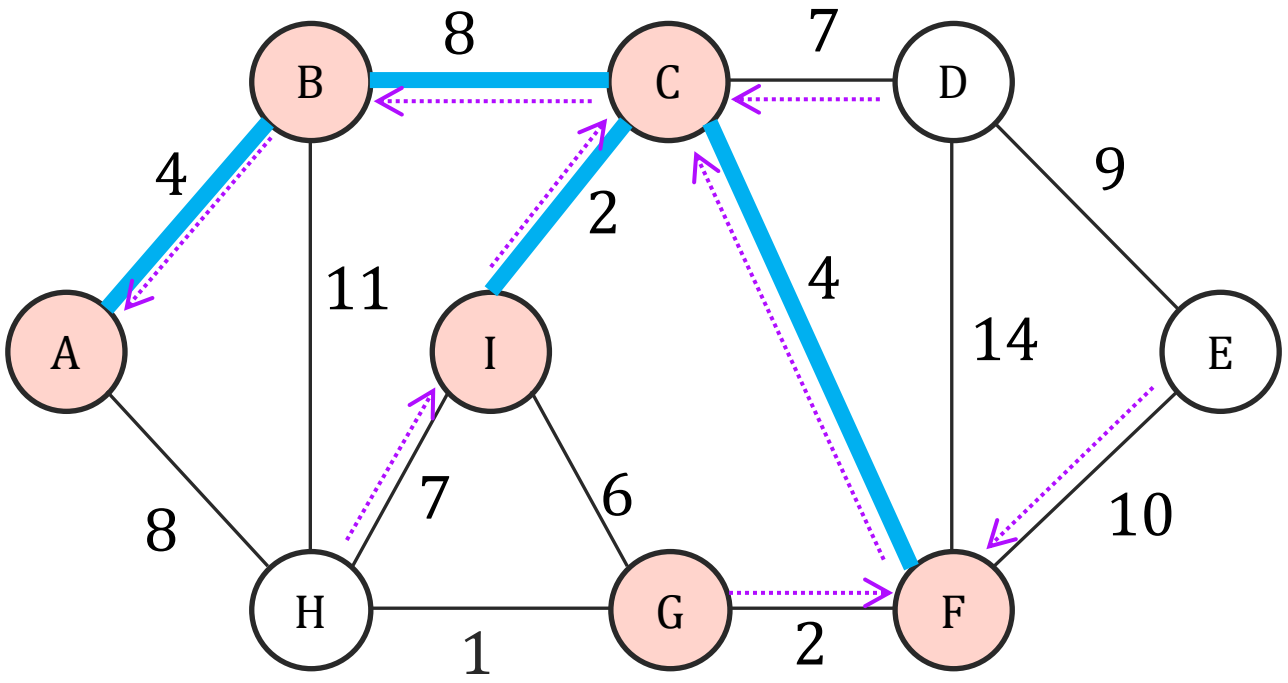
No update

	A	B	C	**D**	**E**	F	**G**	H	I
dist	0	4	8	7	**10**	4	**2**	7	2
prev	∅	*A*	*B*	*C*	***F***	*C*	***F***	*I*	*C*

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

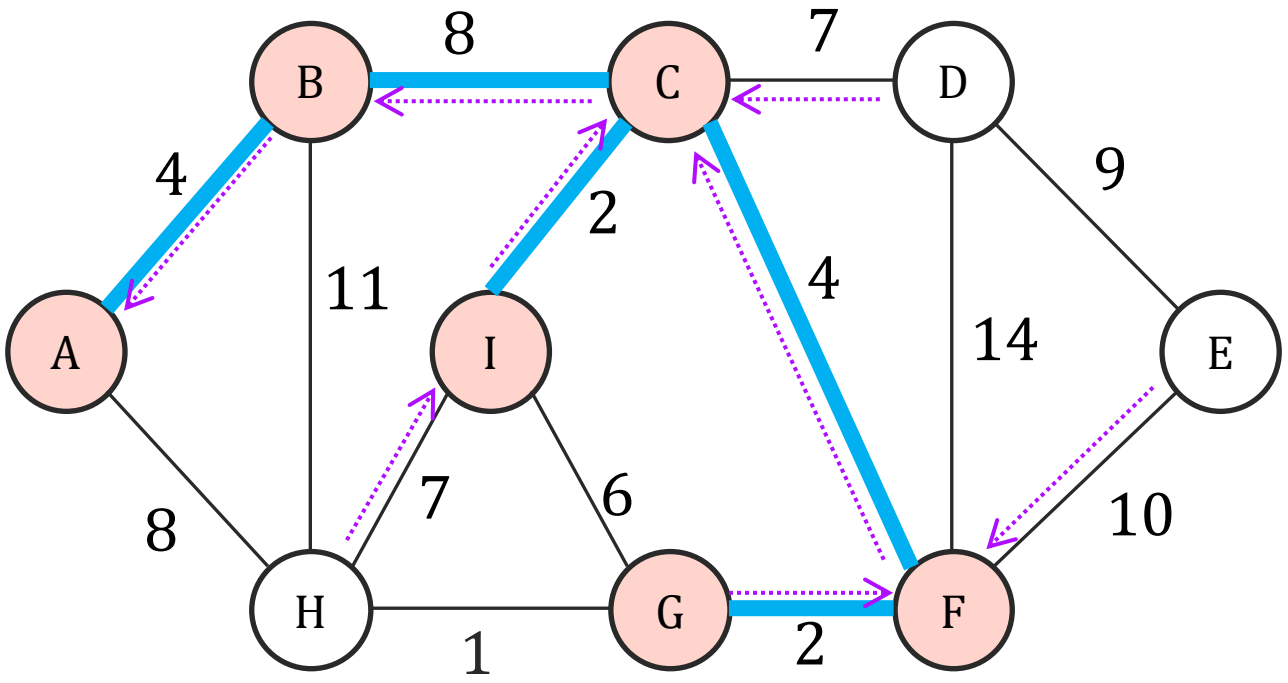


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	7	2
$prev$	∅	A	B	C	F	C	F	I	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

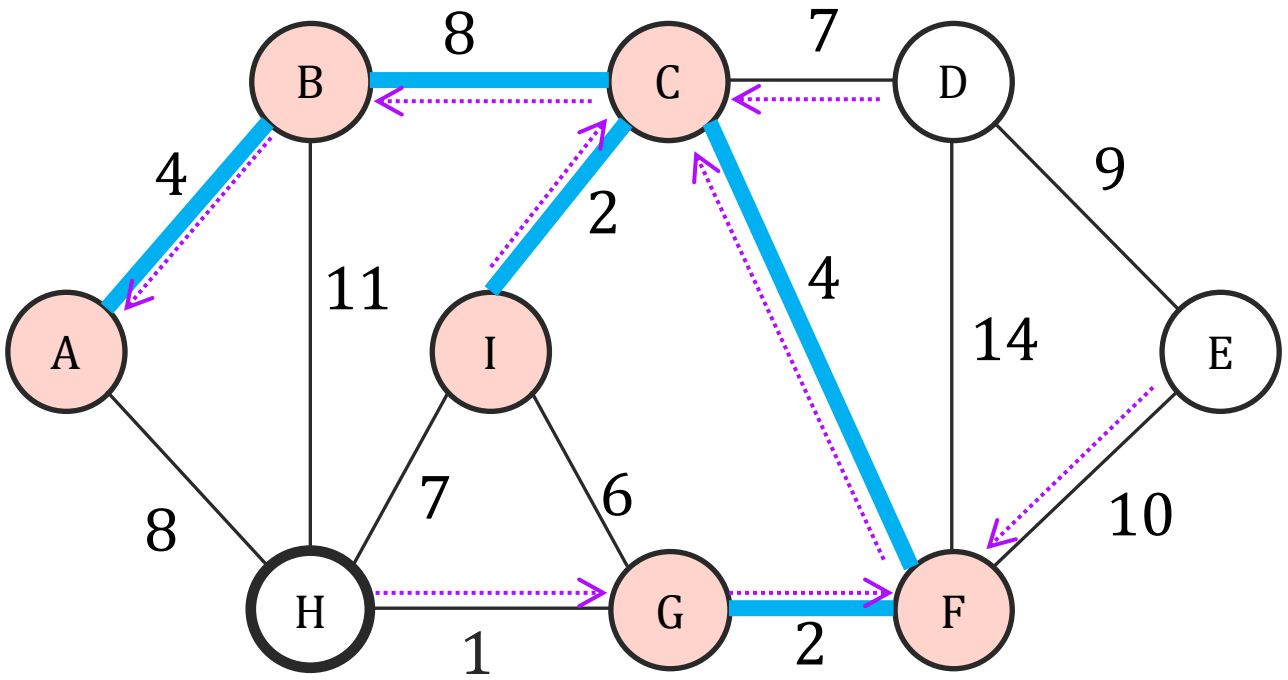


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	7	2
$prev$	∅	A	B	C	F	C	F	I	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

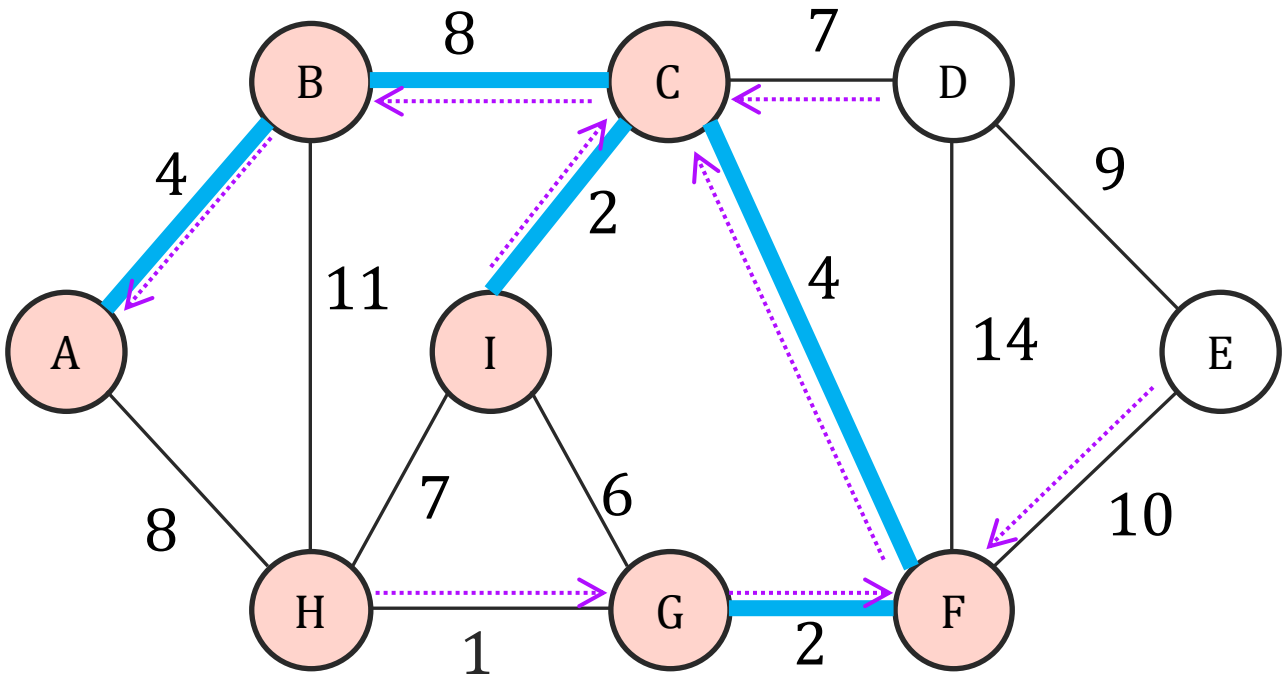


	A	B	C	D	E	F	G	**H**	I
$dist$	0	4	8	7	10	4	2	**1**	2
$prev$	∅	A	B	C	F	C	F	***G***	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

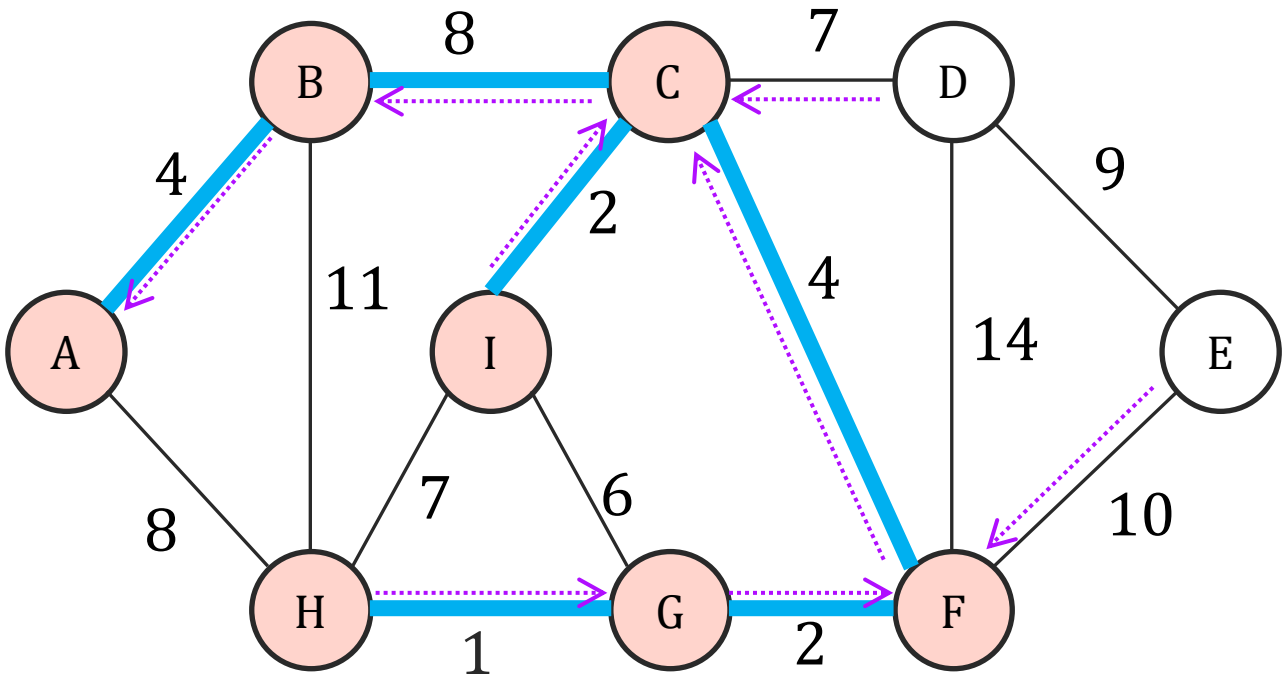


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	∅	A	B	C	F	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

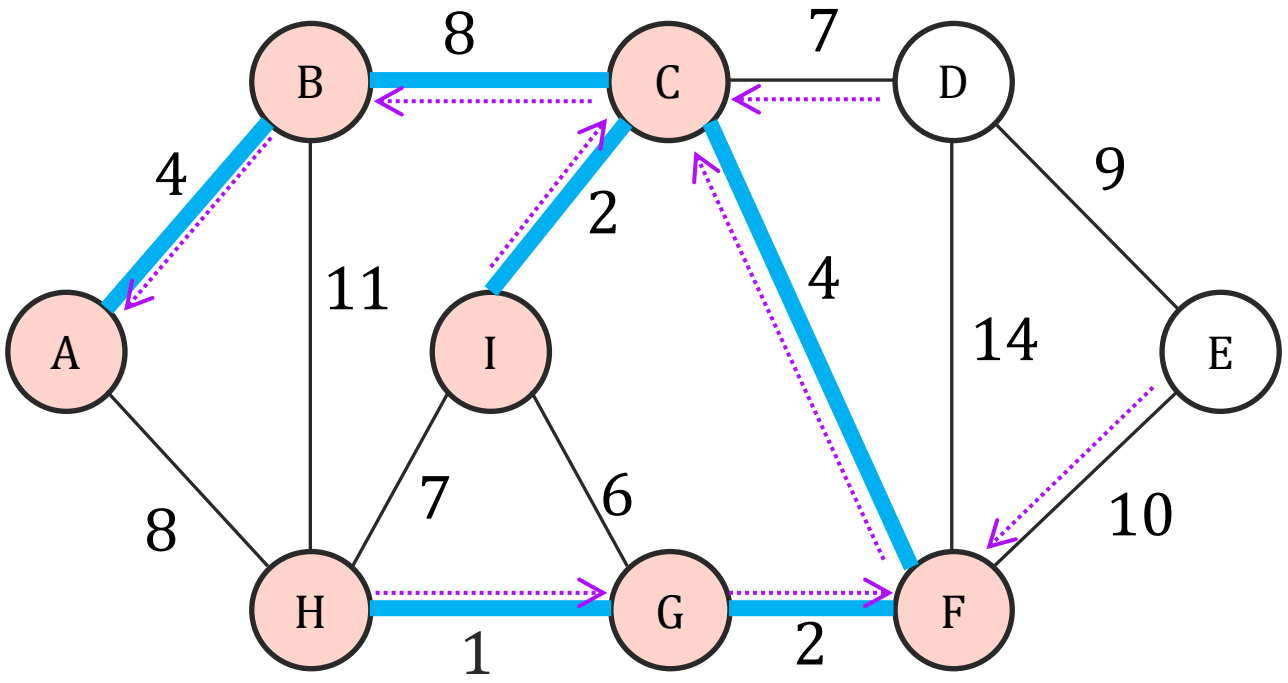


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	∅	A	B	C	F	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$



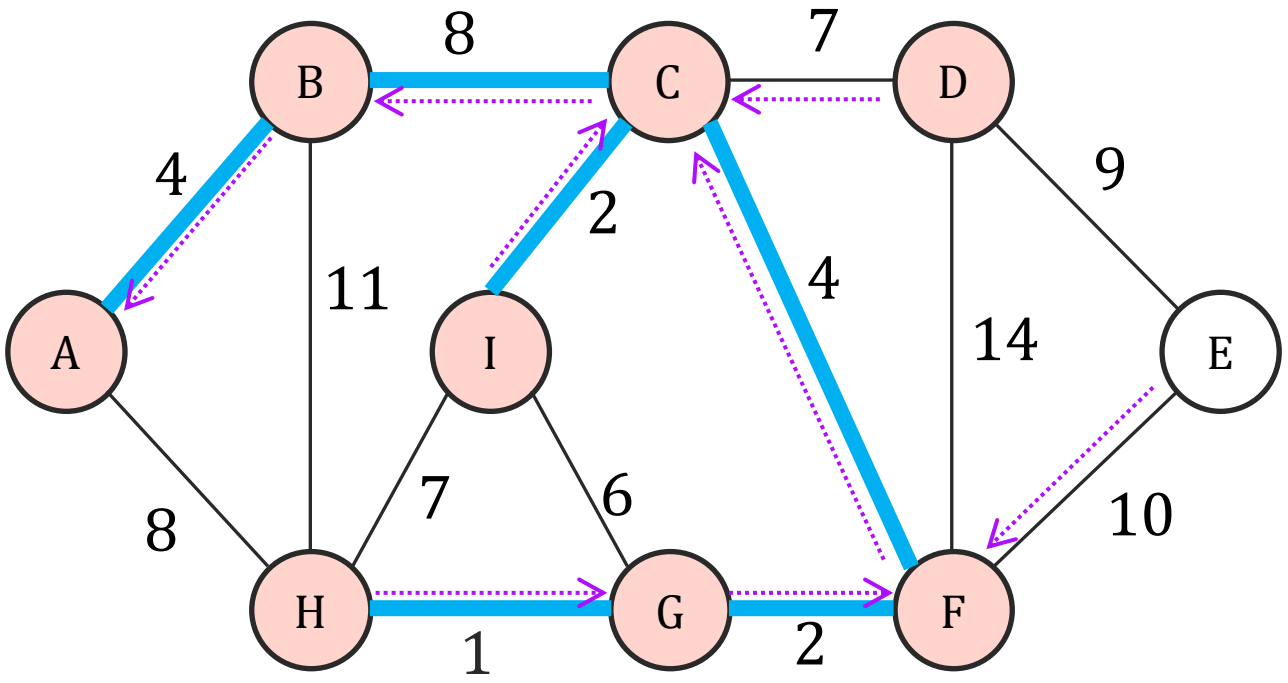
Nothing to update

	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	∅	A	B	C	F	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

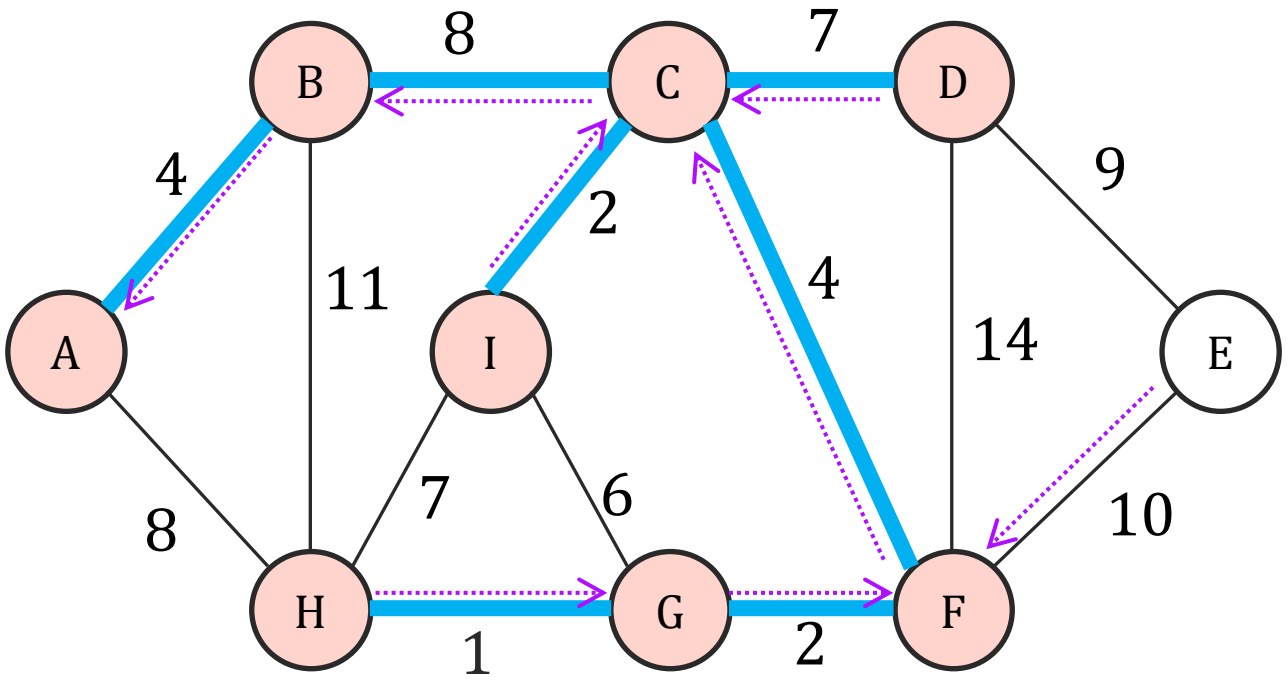


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	$\emptyset$	A	B	C	F	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

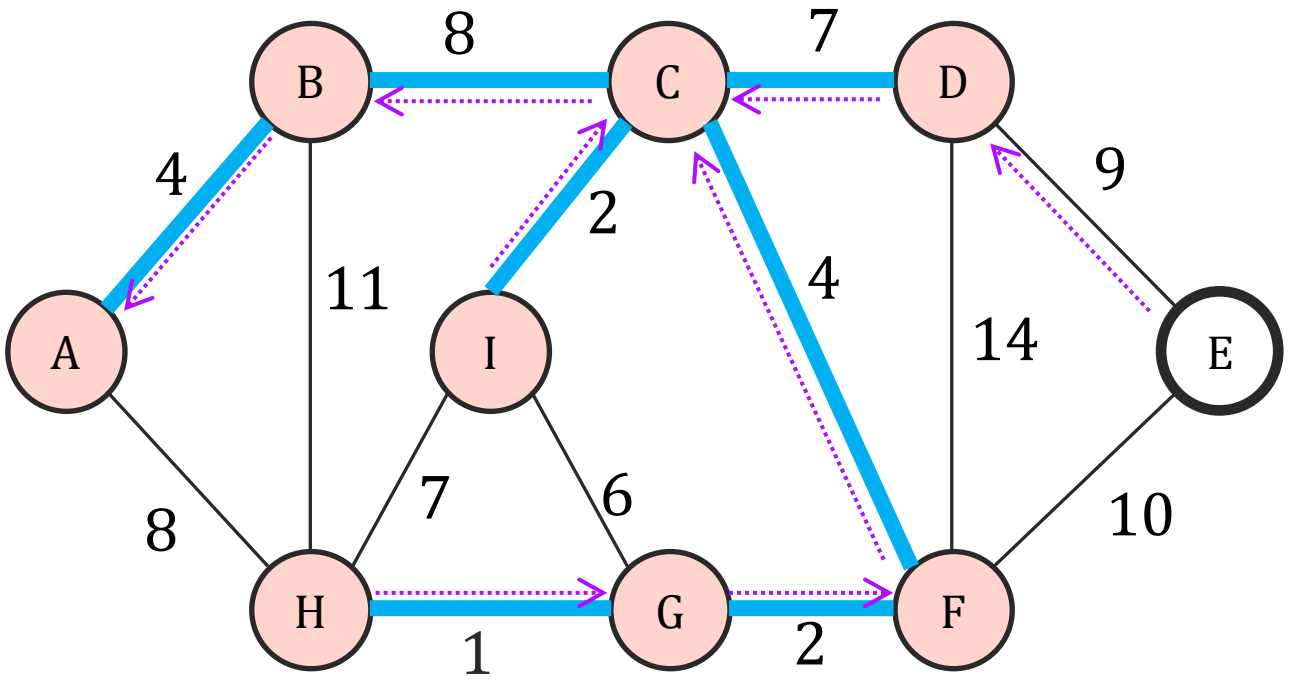


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	$\emptyset$	A	B	C	F	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to *prev*

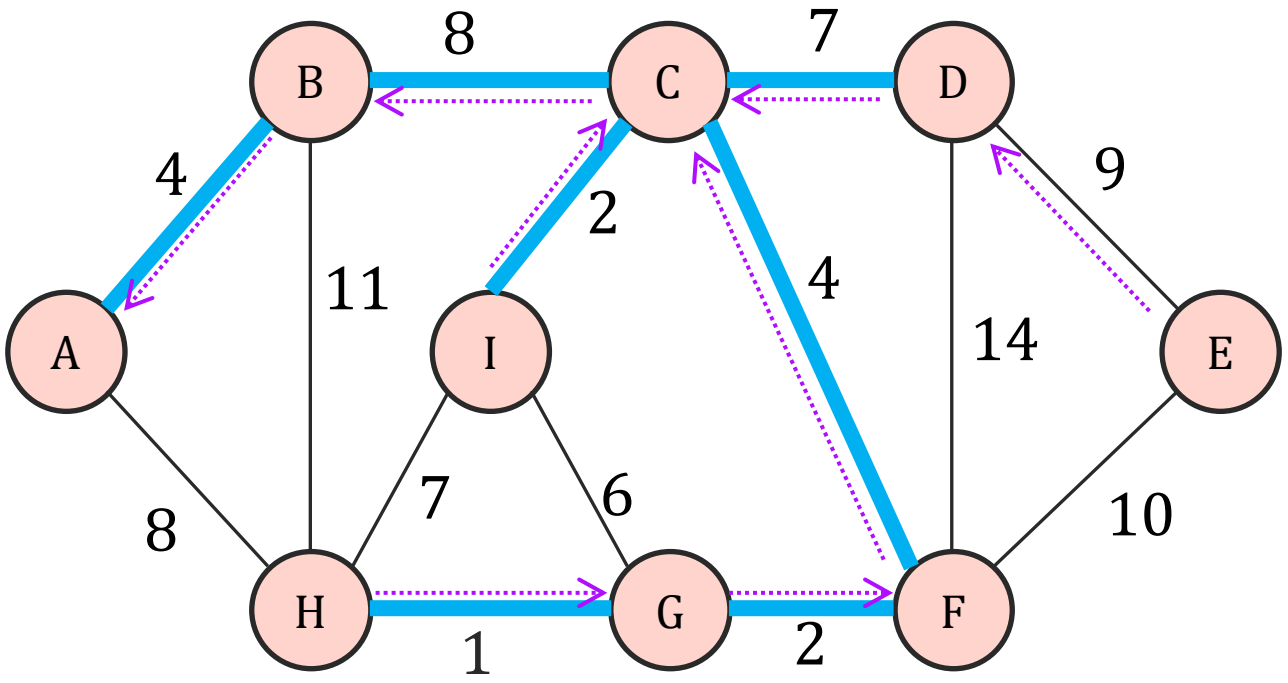


	A	B	C	D	**E**	F	G	H	I
dist	0	4	8	7	**9**	4	2	1	2
prev	Ø	*A*	*B*	*C*	***D***	*C*	*F*	*G*	*C*

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w(v,z) and z ∈ Q.
                Q.decreaseKey(z, w(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$

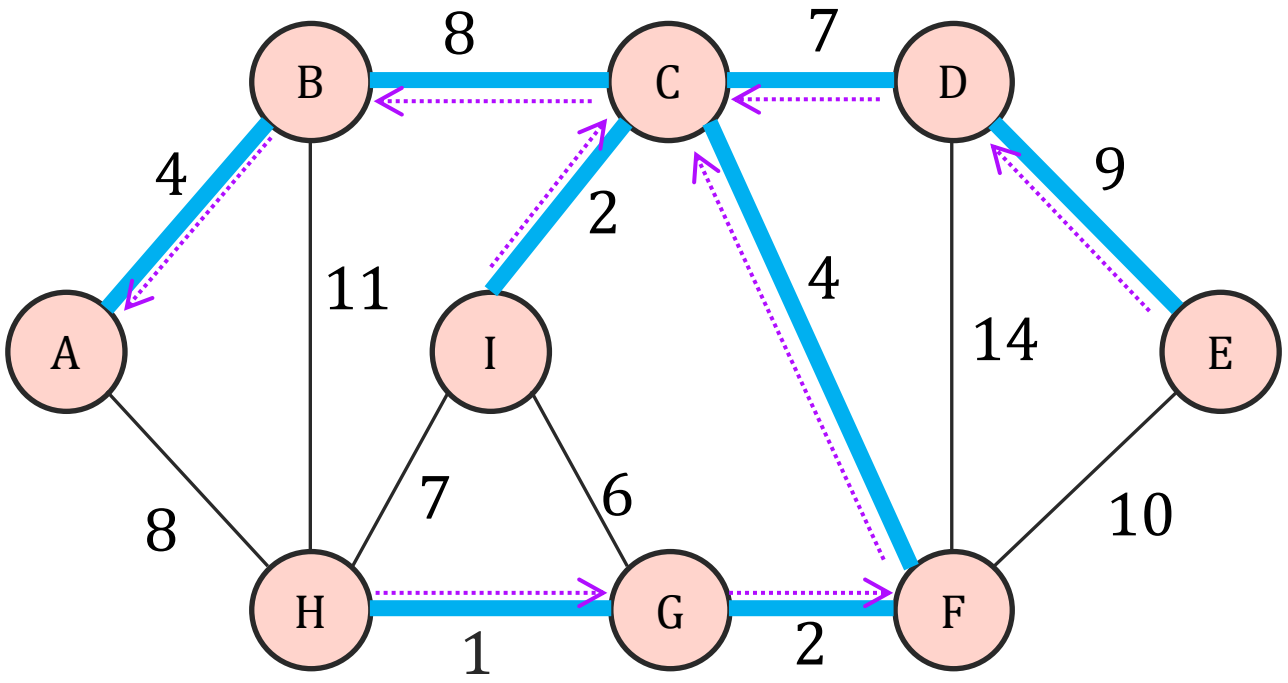


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	9	4	2	1	2
$prev$	∅	A	B	C	D	C	F	G	C

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Prim's Algorithm: Efficient Implementation

Red nodes: those deleted from Q Purple dotted line points to $prev$



	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	9	4	2	1	2
$prev$	$\emptyset$	A	B	C	D	C	F	G	C

Fast-Prim$(G = (V, E))$

array $dist(n)$ // initialize to all ∞

array $prev(n)$ // initialized to null

$X = \{\}$ and Q empty priority queue

$dist[A] = 0$ // an arbitrary node A

for $v \in V$, Q. insert$(v, dist[v])$

while $|X| < |V| - 1$

$v \leftarrow$ Q.deleteMin

if $v \neq A, X \leftarrow X \cup \{(prev[v], v)\}$

for $(v, z) \in E$

if $dist[z] > w_{(v,z)}$ and $z \in Q$.

Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Runtime of Prim's Algorithm

Recall Priority Queue implementations

→Binary heap: $\log(n)$ per operation.

→ Fibonacci Heap: $\log(n)$ for deleteMin, $O(1)$ for insert and decreaseKey.

Runtime of Prim's:

- n Q.inserts
- n Q.deleteMin
- m Q.decreaseKey

With binary heap: $O\big((m+n)\log(n)\big)$.

With Fibonacci heap: $O(m + n\log(n))$

```
Fast-Prim(G = (V, E))
    array dist(n)  // initialize to all ∞
    array prev(n)  // initialized to null
    X = {} and Q empty priority queue
    dist[A] = 0 // an arbitrary node A
    for v ∈ V, Q.insert(v, dist[v])
    while |X| < |V| − 1
        v ← Q.deleteMin
        if v ≠ A, X ← X ∪ {(prev[v], v)}
        for (v, z) ∈ E
            if dist[z] > w_(v,z) and z ∈ Q.
                Q.decreaseKey(z, w_(v,z))
                prev[z] ← v
    return X
```

Comparing MST algorithm's runtimes

- Kruskal's runtime: $O((m+n)\log(n))$
- Prim's runtime: $O(m+n\log(n))$

→ For sparse graphs (m $= O(n)$), both equally good.

→For dense graphs, $(m \gg (n\log(n))$, Prim is much faster than Kruskal.

Other fun facts (no need to memorize):

- $O(m+n)$ expected runtime of a randomized algorithm: Karger, Klein, Tarjan 1995.
- Deterministic $O\big(m\,\alpha(m,n)\big)$: Chazelle 2000

→$\alpha(m,n)$ is called "inverse Ackerman" function and $\alpha(m,n) \leq 5$ for m,n being # of particles in the universe!

- A deterministic algorithm with $O(optimal)$: Pettie, Ramachandran 2002

→ What's "optimal"? No idea!

Wrap up

We saw a meta algorithm for MSTs

- → Variant one: Kruskal's Algorithm
  - → Greedily add the lightest edge that doesn't create a cycle
  - → Union-Find: Useful data structure for keeping track of sets and trees.
- → Variant two: Prim's Algorithm
  - → Greedily add the lightest edge that expands the tree
  - → Dijkstra-like update.

→ We are (almost done) with Greedy!

Next time

Dynamic Programming