CS 170
Efficient Algorithms and Intractable Problems

Lecture 10
Minimum Spanning Trees

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 next Tuesday Feb 25
→ No class on Tuesday!
→ No discussion sections on Tuesday!
→ Midterm 1 Review Sessions: 9-11 Friday @Soda 306
→ Also there are more OHs
→ Scope: Up to and including today's lecture!

HW4 is due this Saturday.
HW 5 is optional and not for grade.
→ Posted with solutions, so review the solutions!

Last Lecture: Minimum Spanning Trees

Minimum Spanning Tree (MST) Problem:
Input: a weighted graph $G = (V, E)$ with non-negative weights.
Output: A tree $T \subseteq E$ connecting all the vertices of the graph with **smallest cost** $\sum_{e \in T} w_e$

Recap: We prove that any algorithm that fits the following meta algorithm correctly returns an MST.

"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 $V \setminus S$
    $X \leftarrow X \cup \{e\}$

Today

Two algorithms for MST.

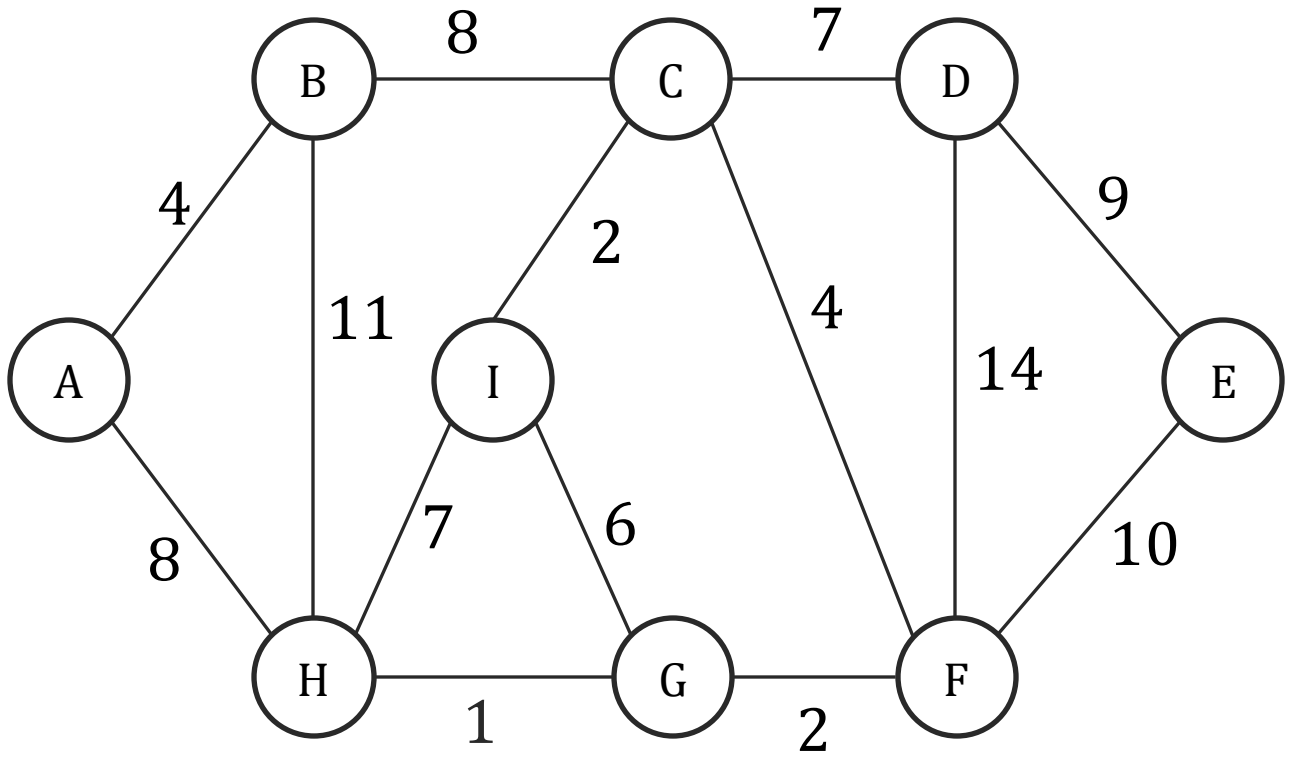
They fit the meta-algorithm recipe!

Based on different choice of cuts.

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

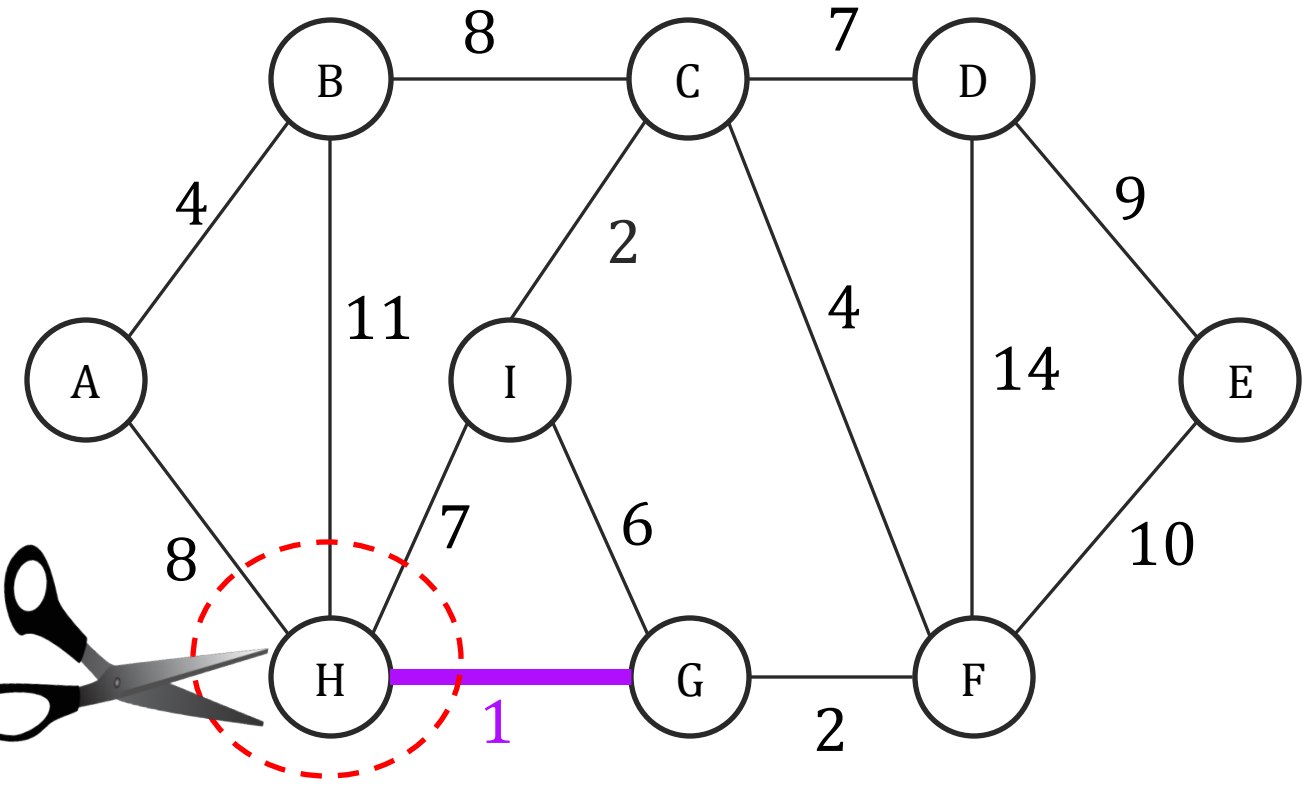
$X \leftarrow X \cup \{e\}$.

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

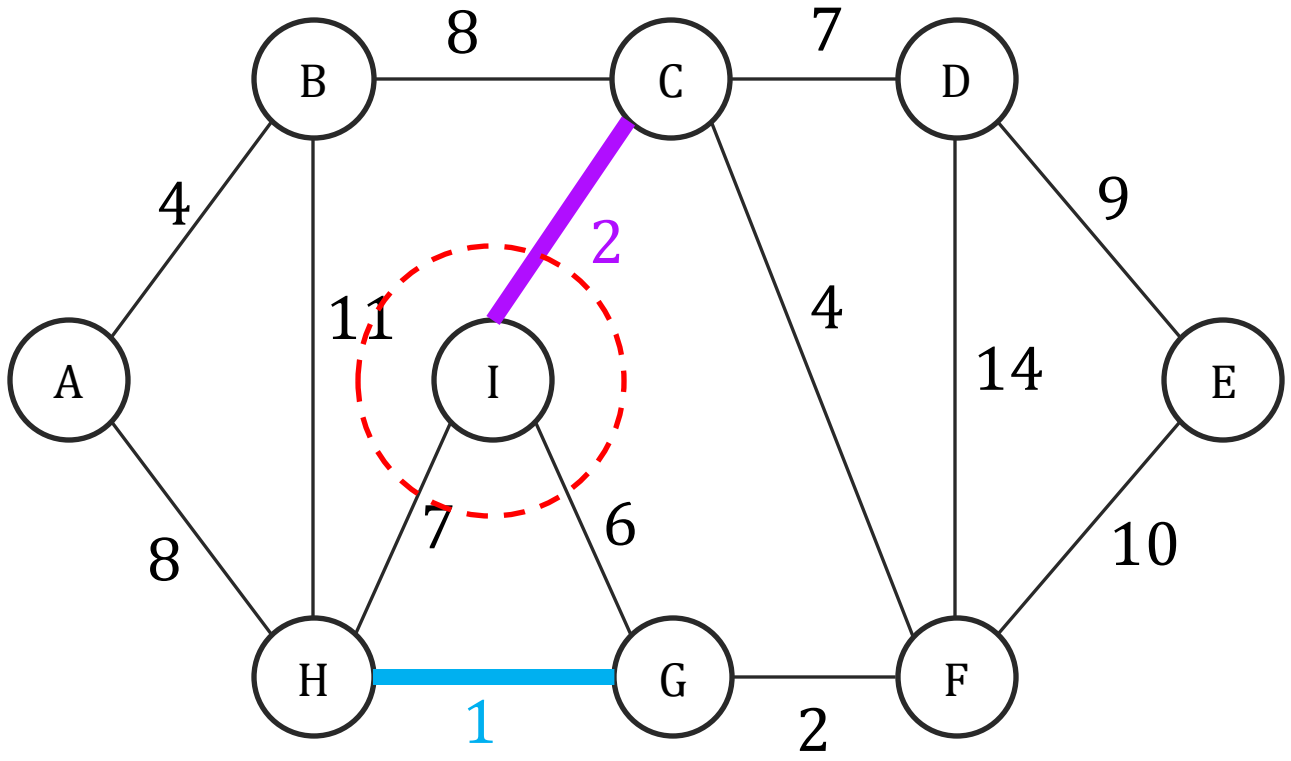
$X \leftarrow X \cup \{e\}$.

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

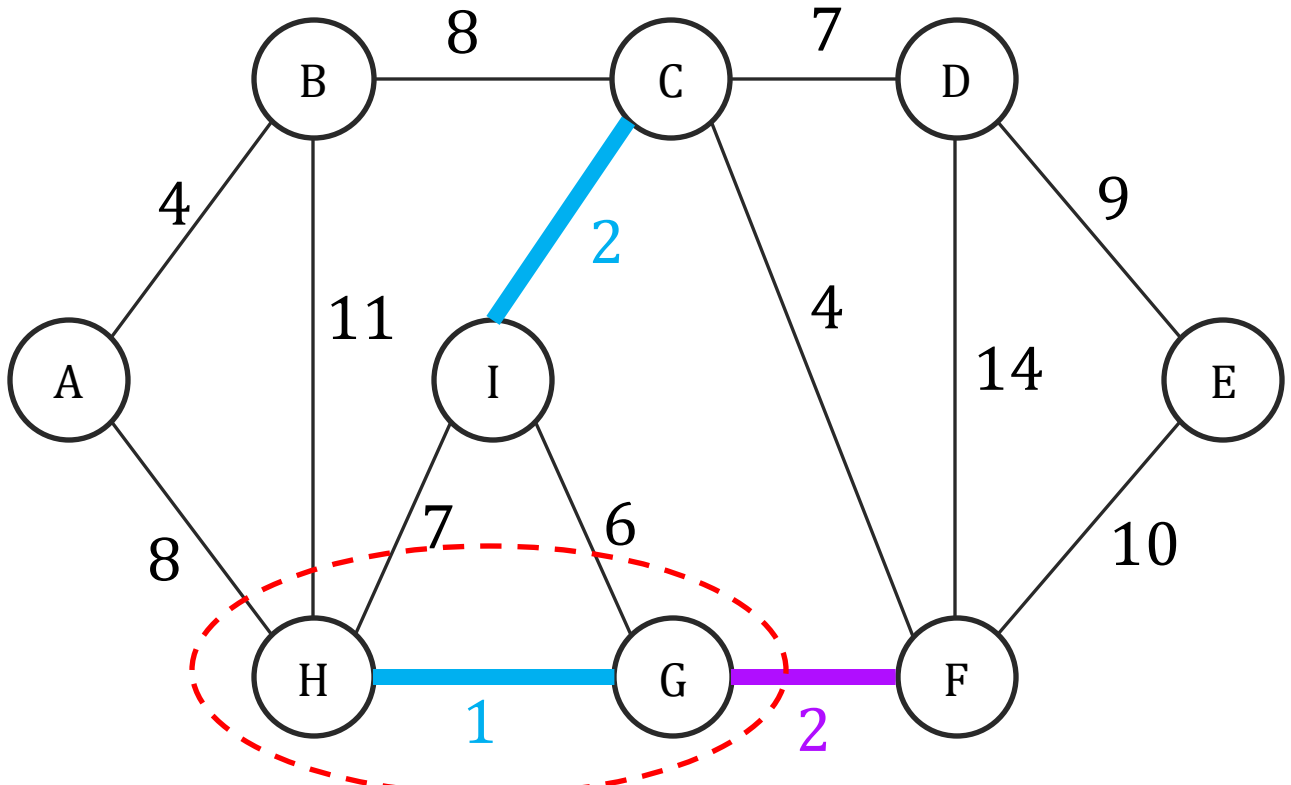
$X \leftarrow X \cup \{e\}$.

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

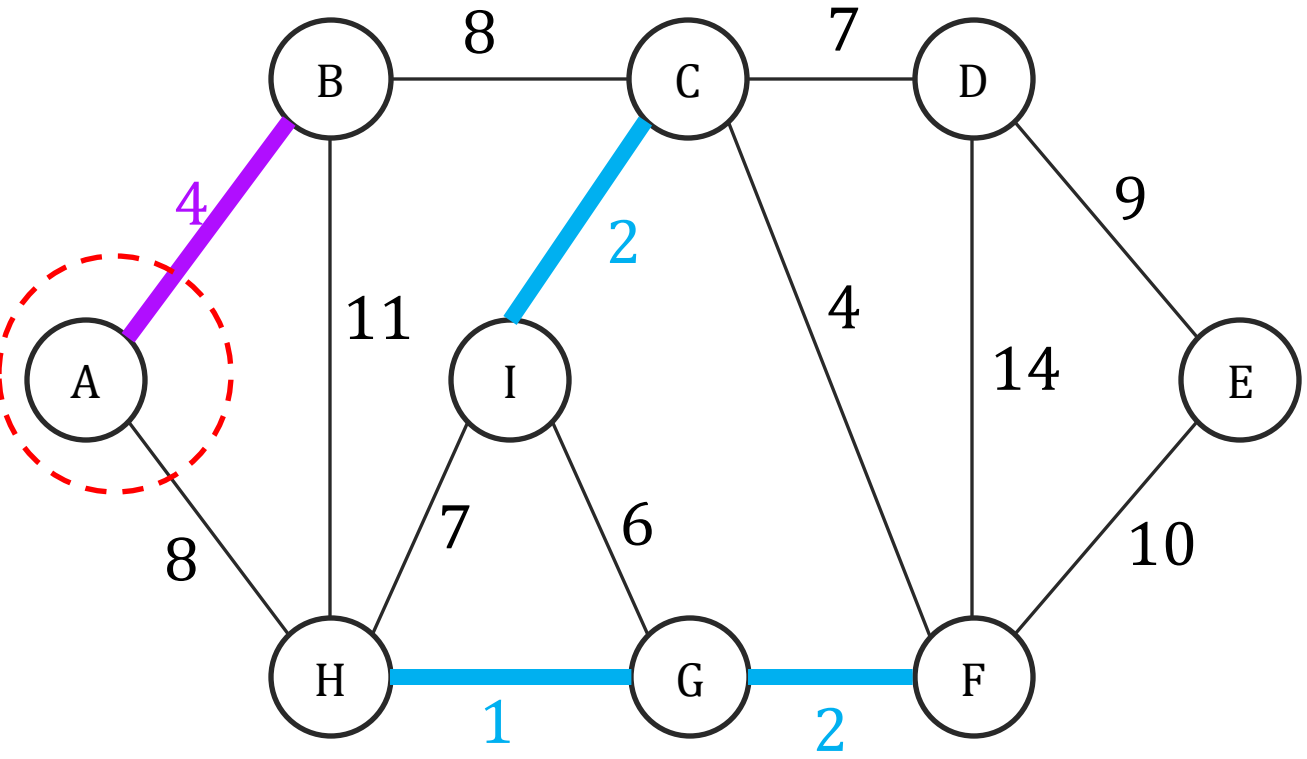
$X \leftarrow X \cup \{e\}$.

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

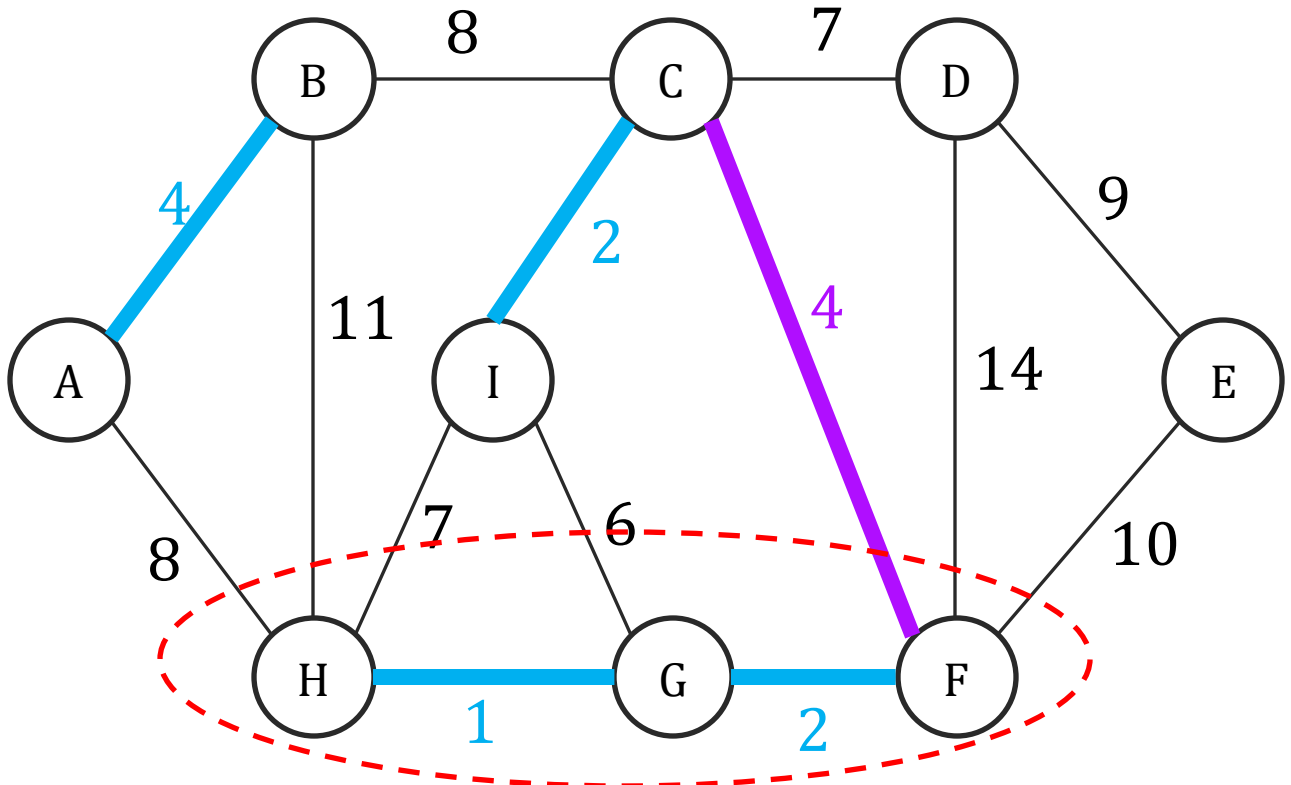
$X \leftarrow X \cup \{e\}$.

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

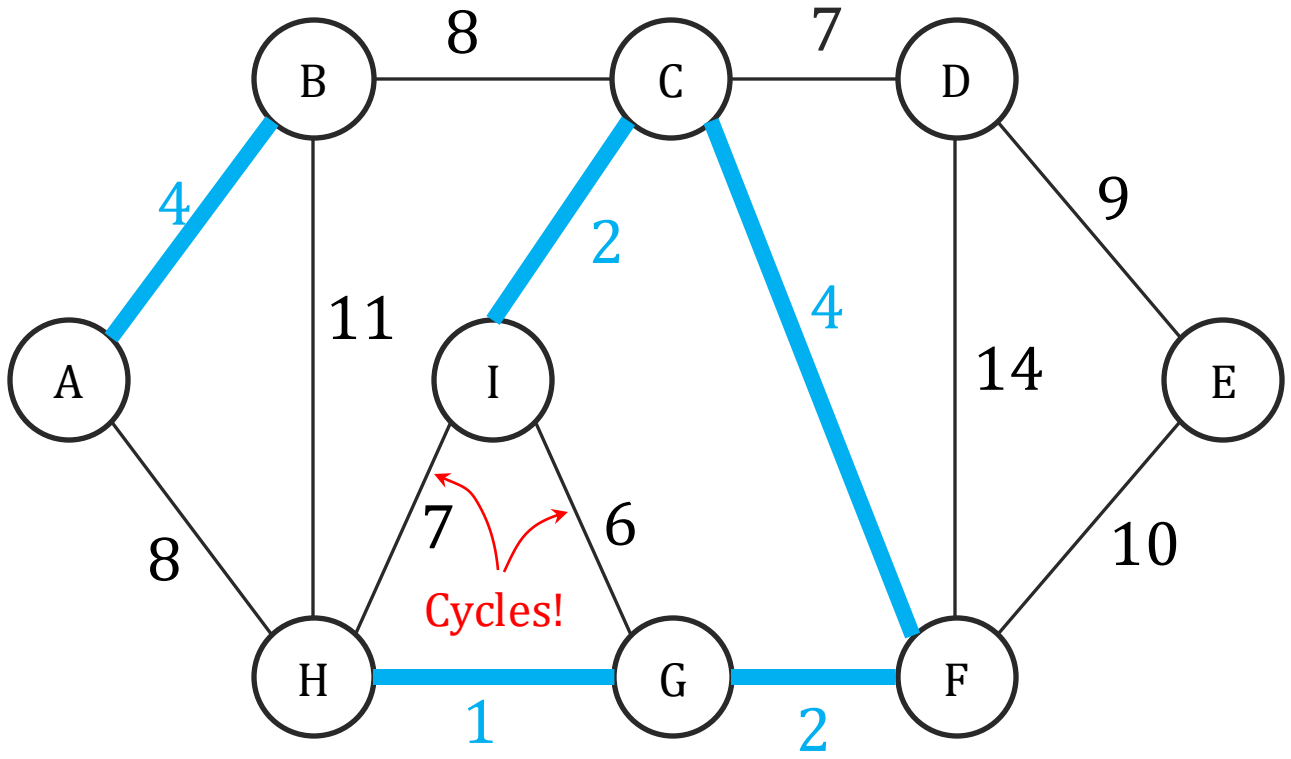
$X \leftarrow X \cup \{e\}$.

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

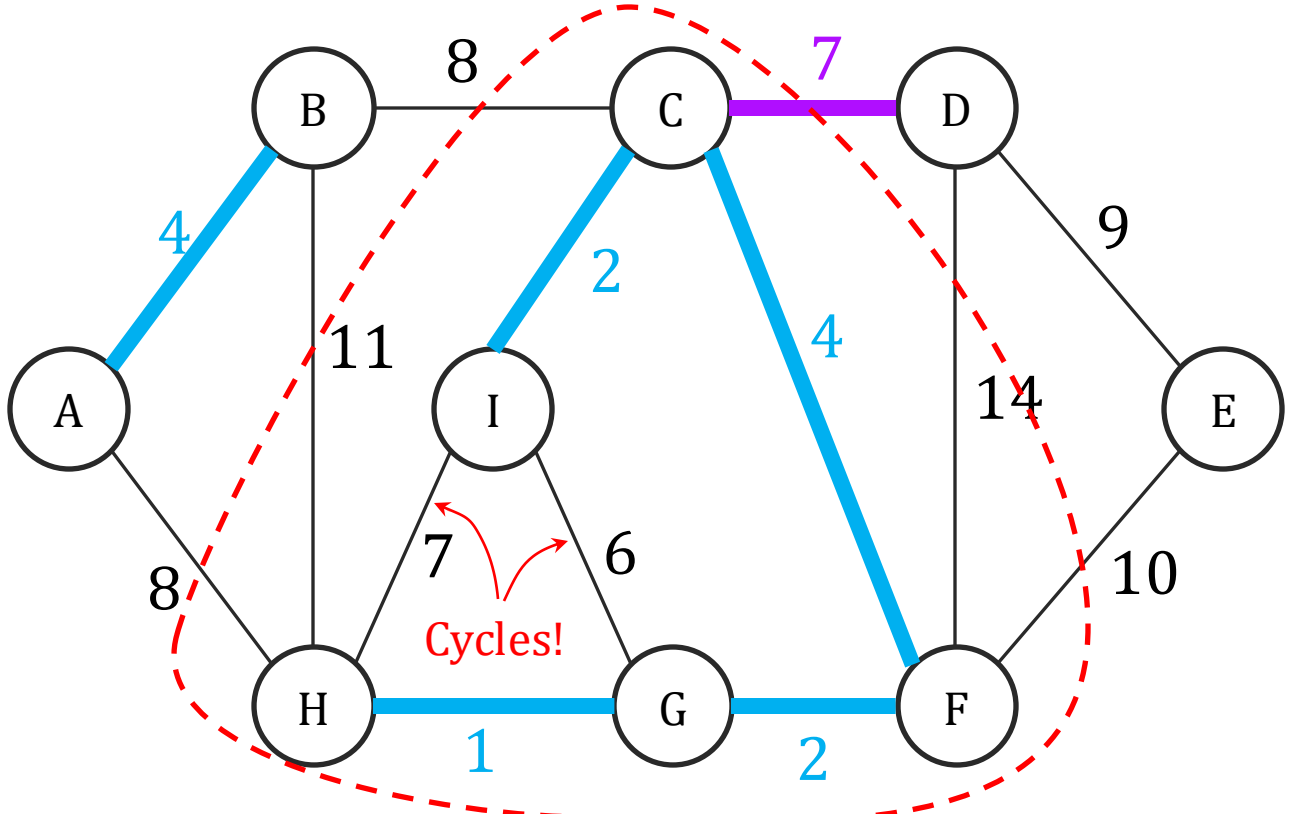
$X \leftarrow X \cup \{e\}$.

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

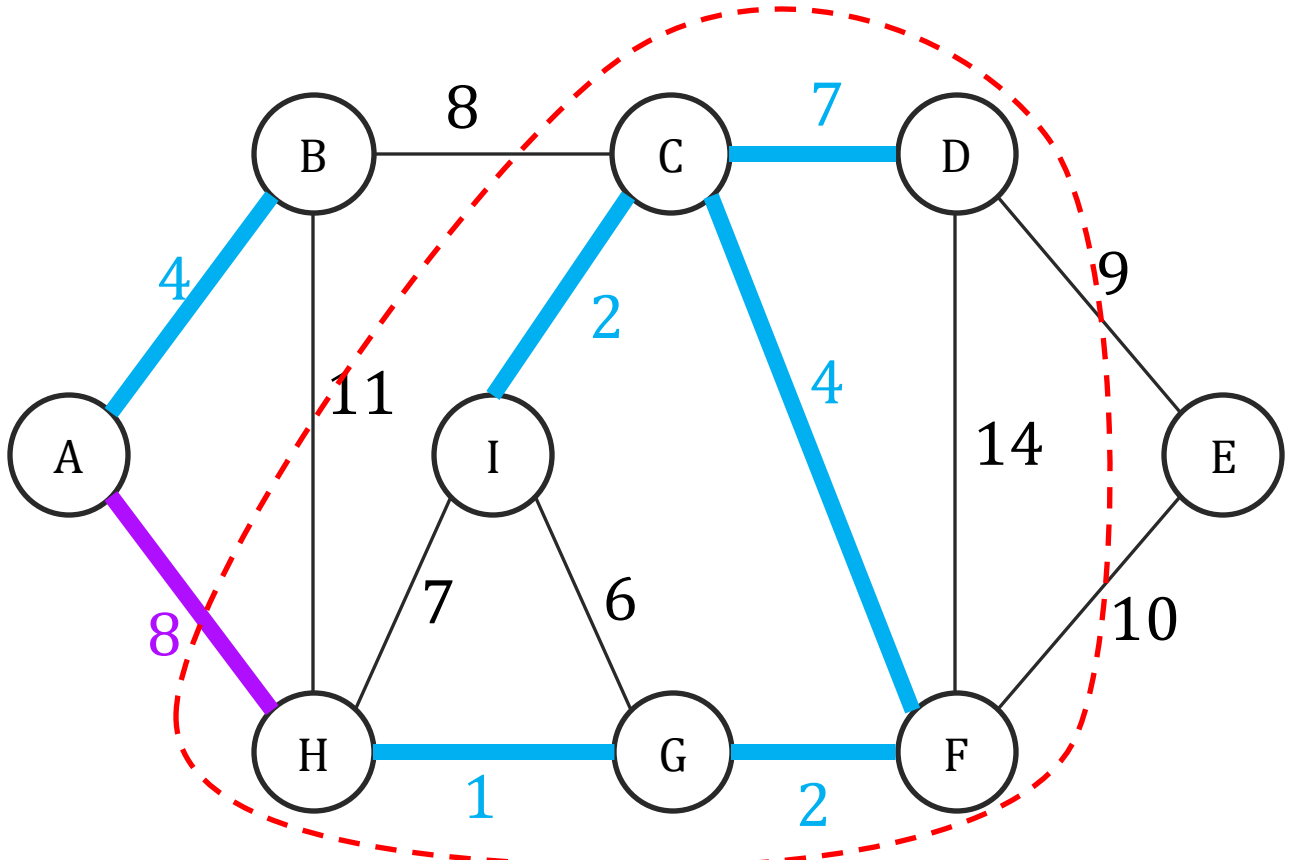
$X \leftarrow X \cup \{e\}$.

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

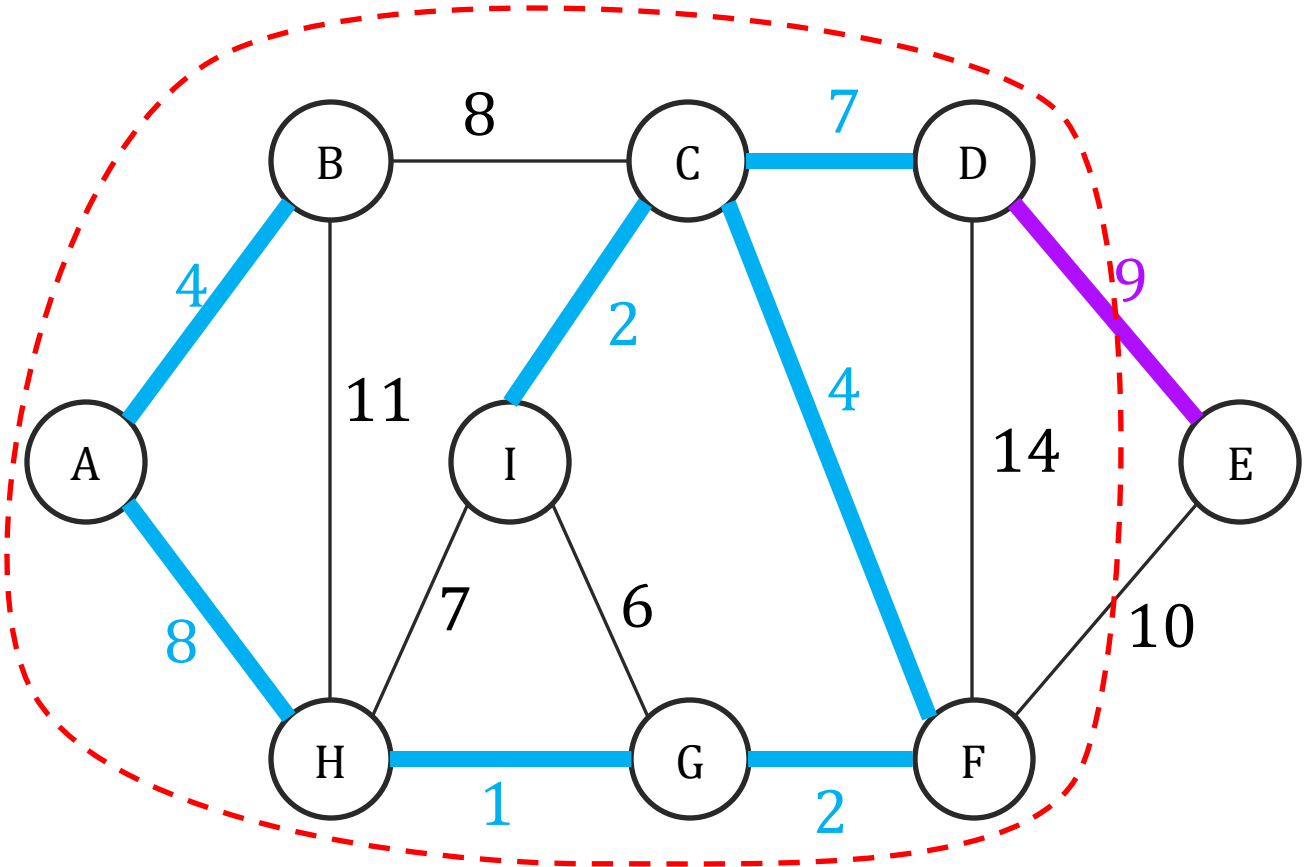
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's 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

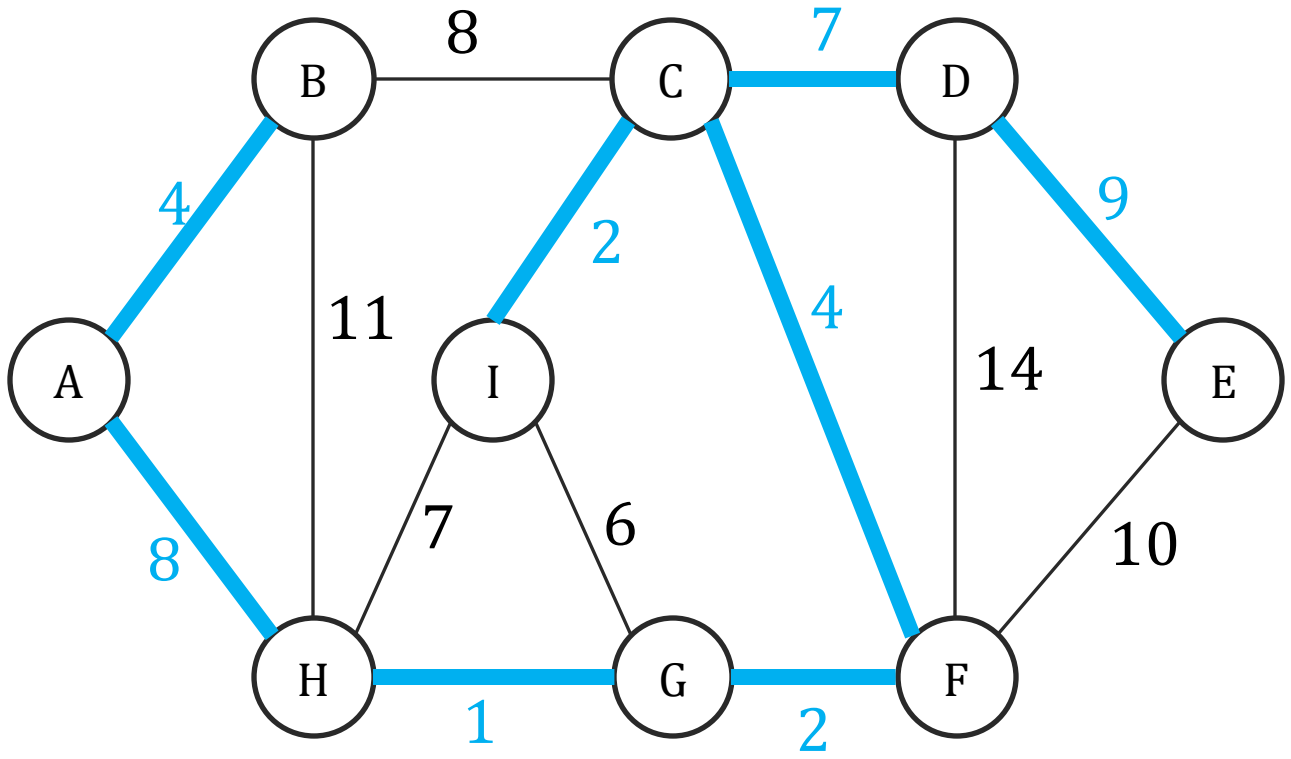
$X \leftarrow X \cup \{e\}$.

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.

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 $V \setminus S$
    $X \leftarrow X \cup \{e\}$

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.

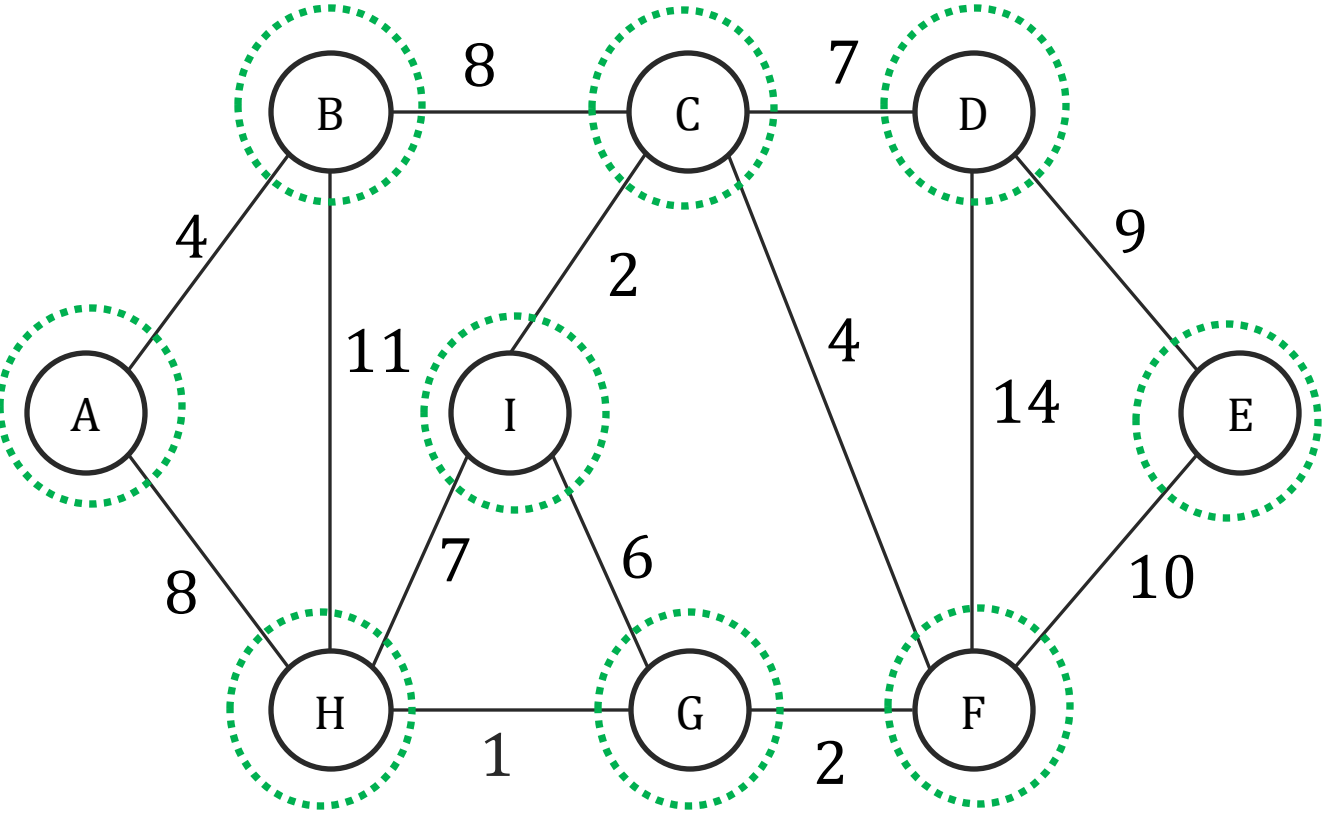
Total: $O((m+n)\log(n))$. For connected graphs = $O(m\log(n))$.

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

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.

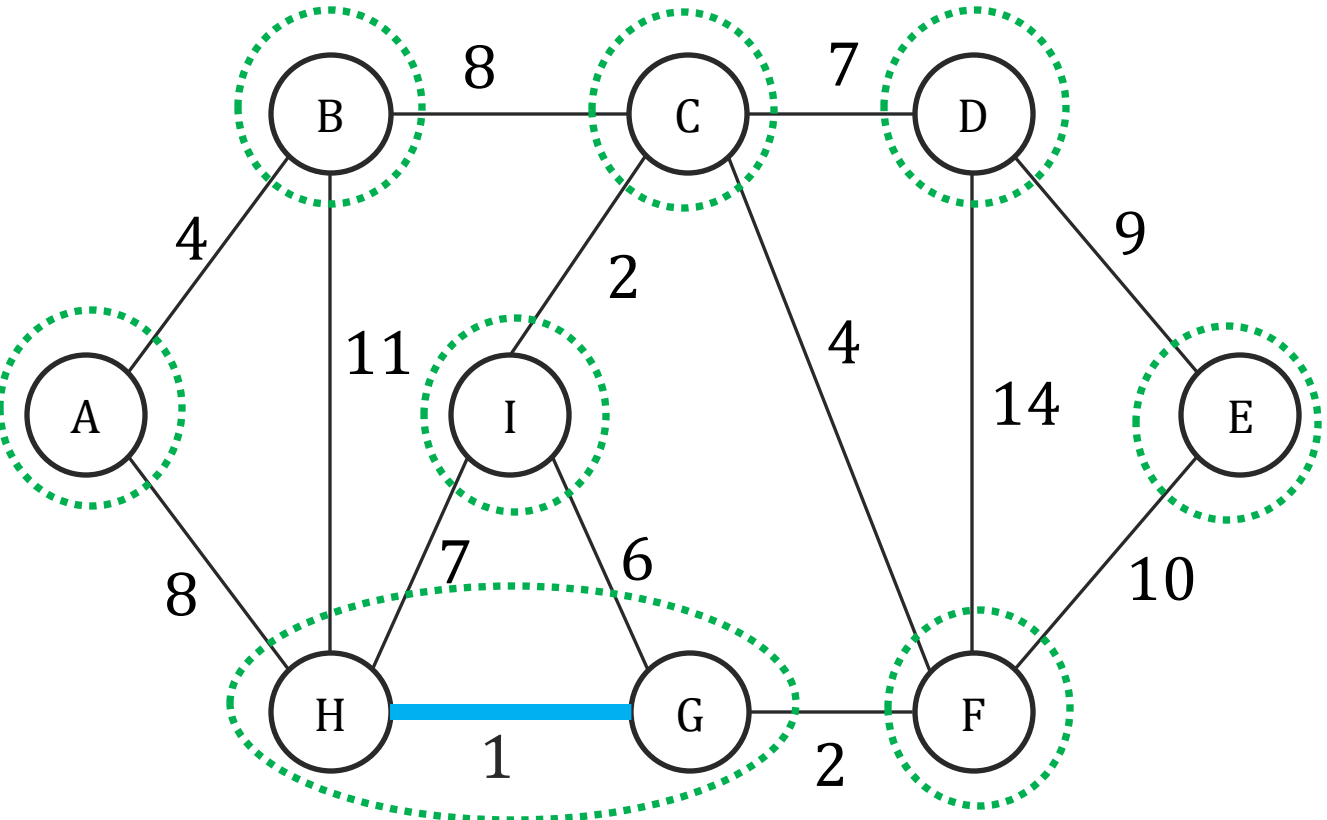


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

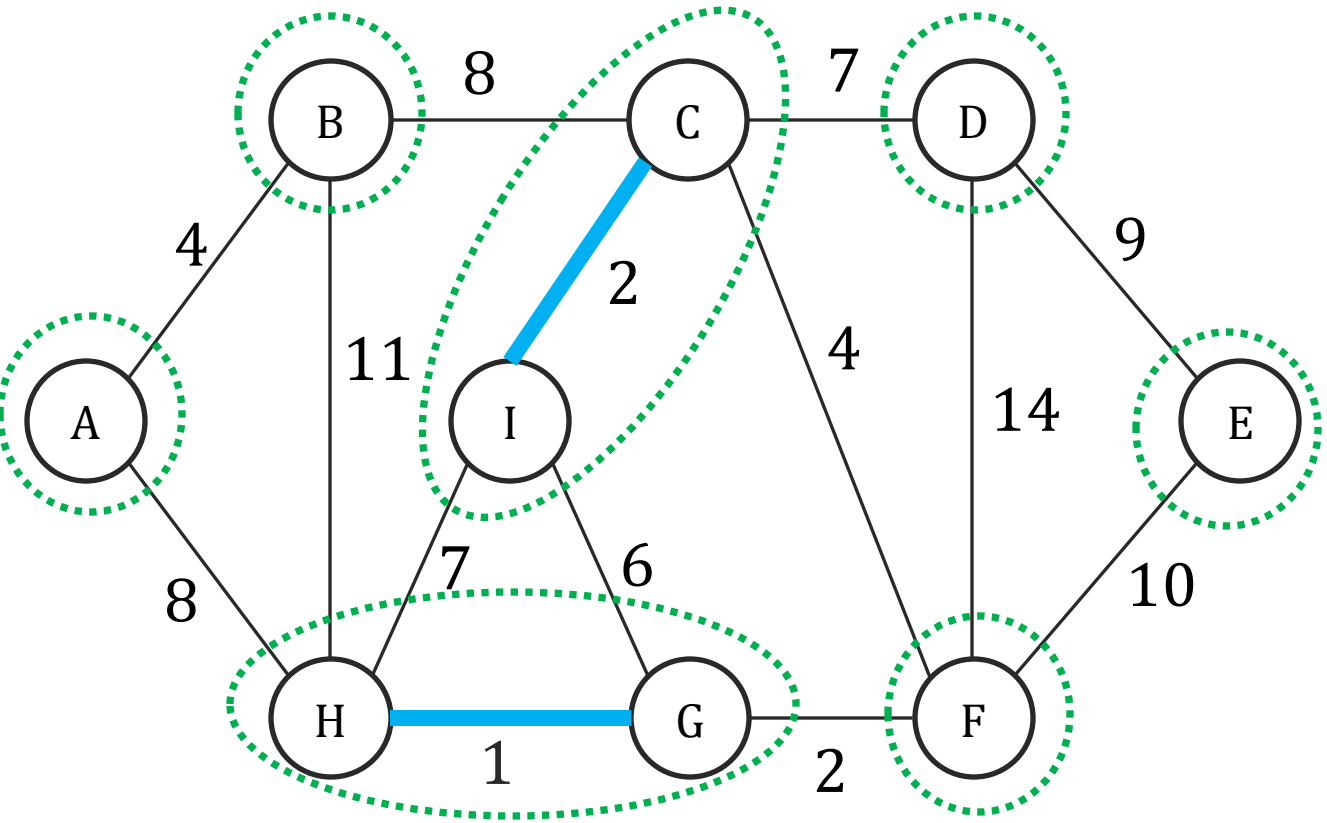


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

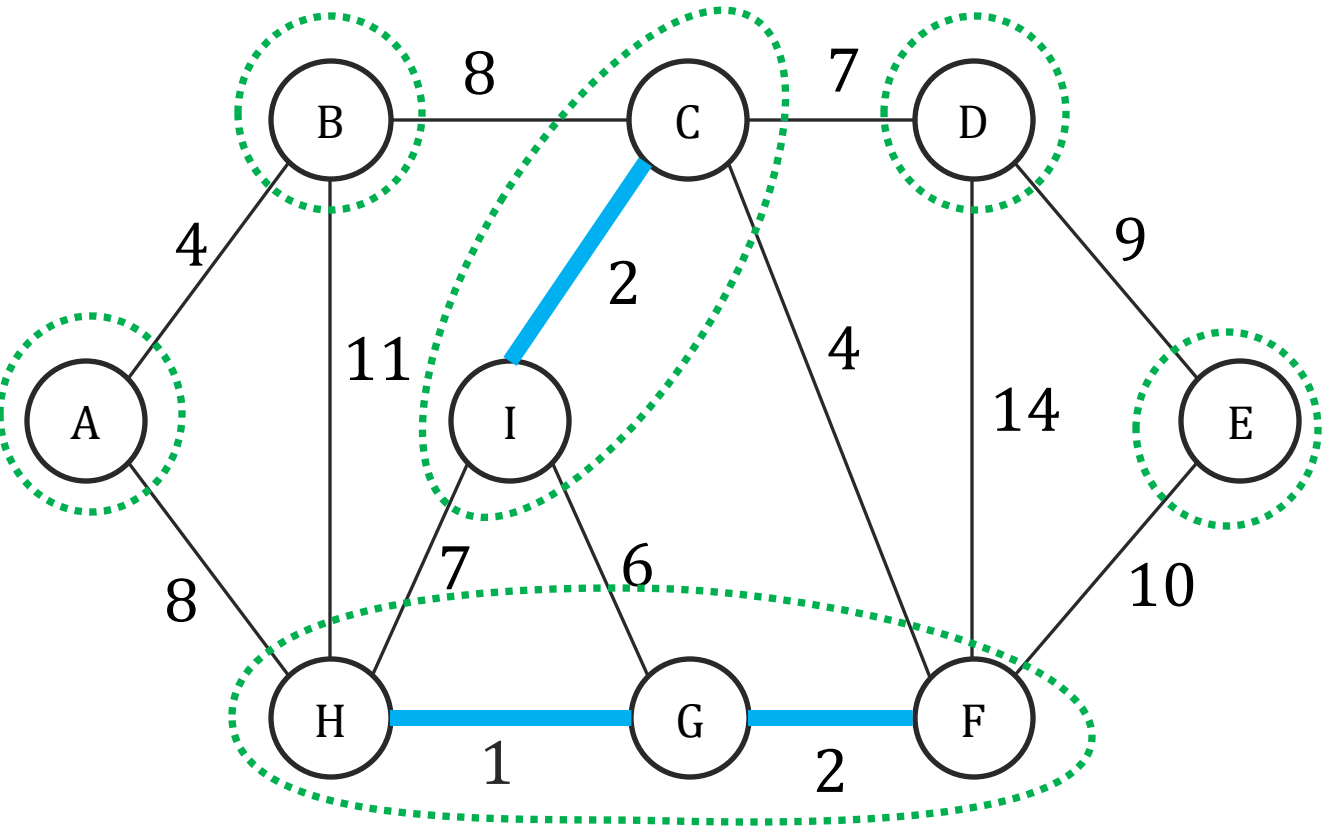


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

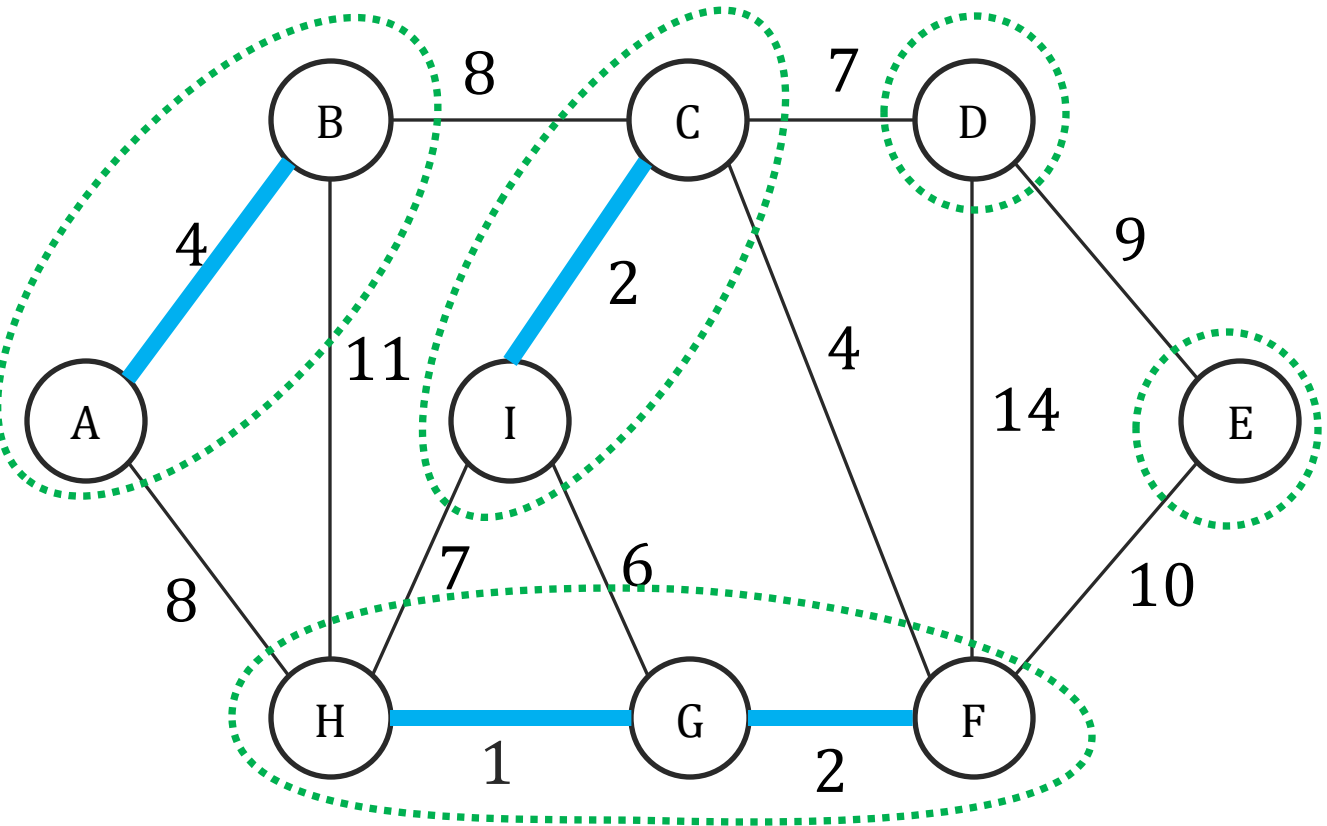


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

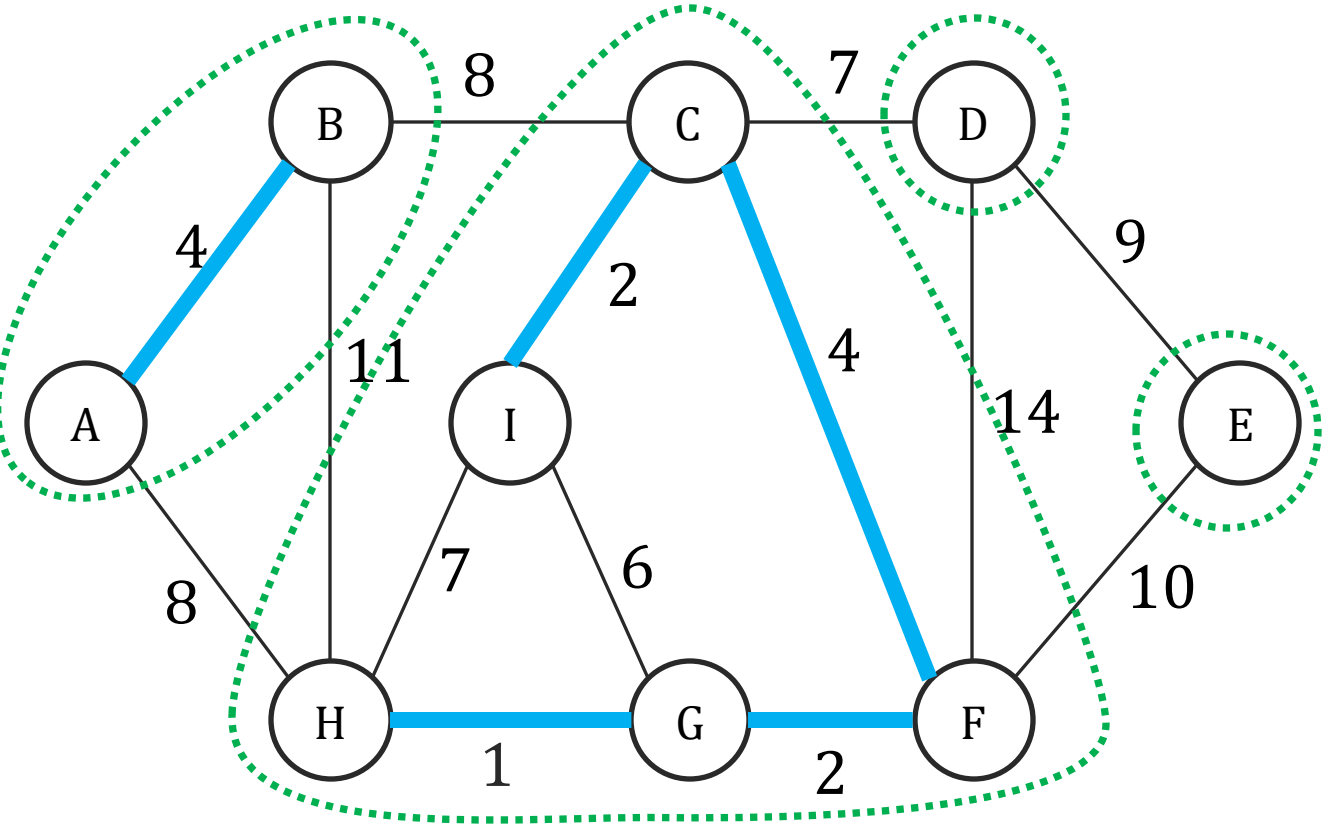


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

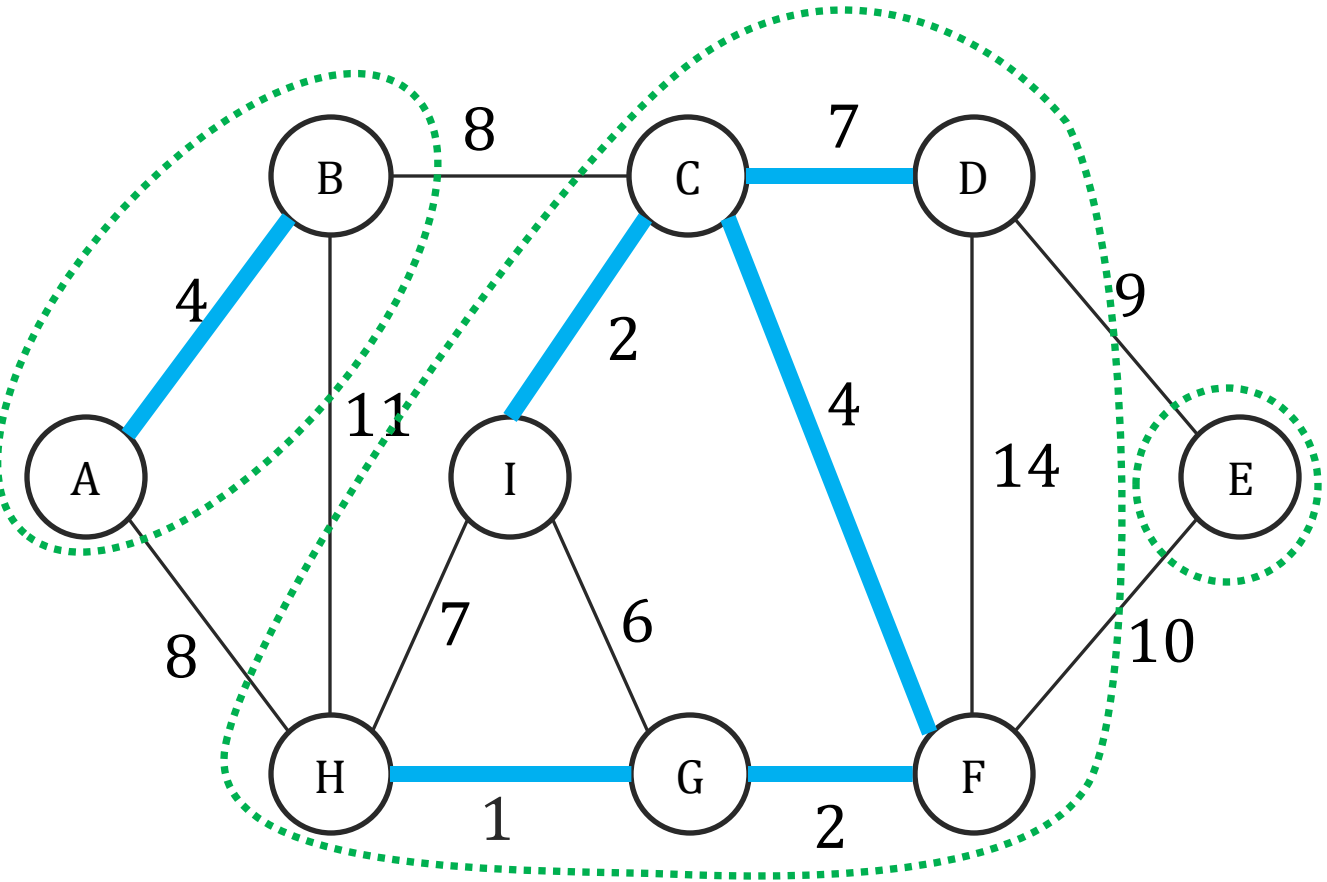


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

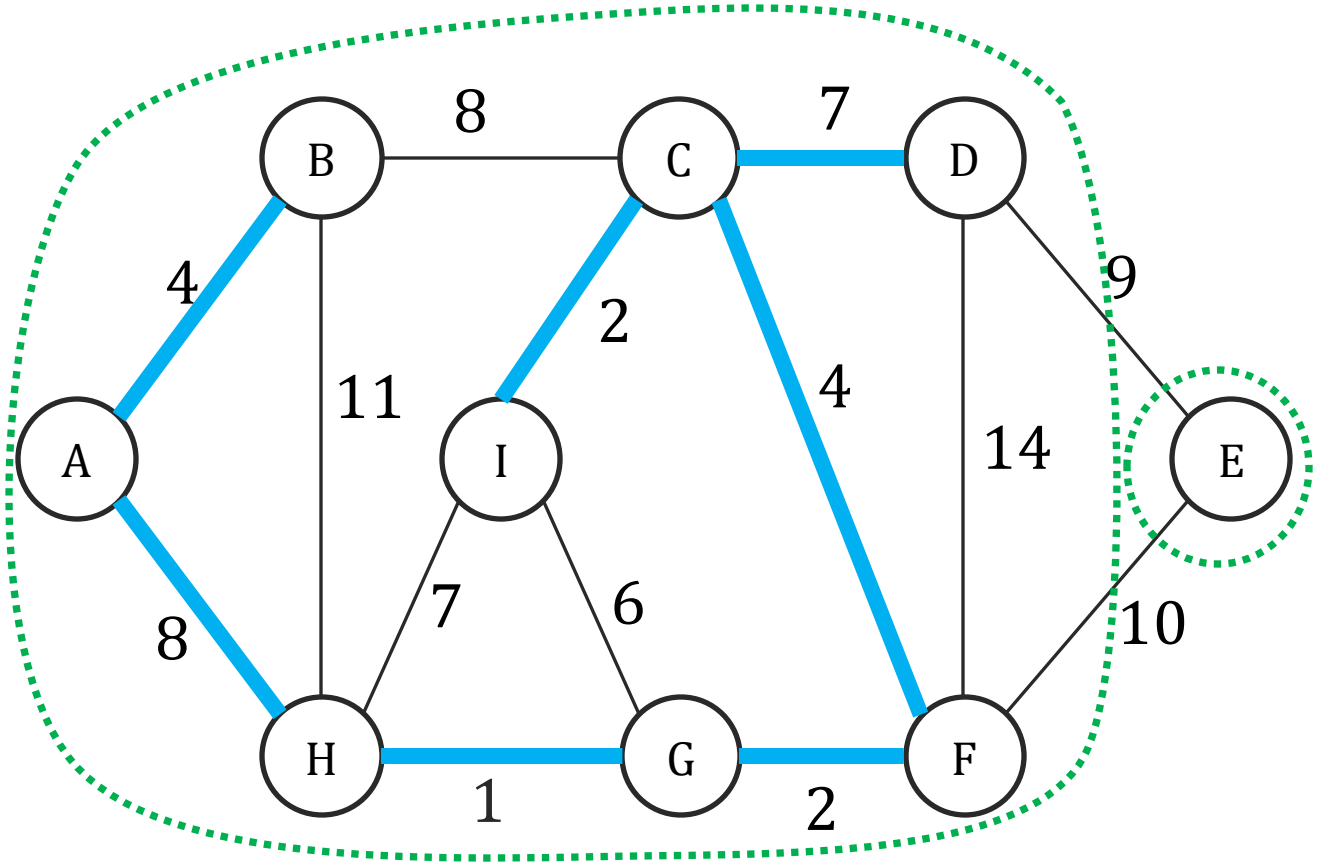


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

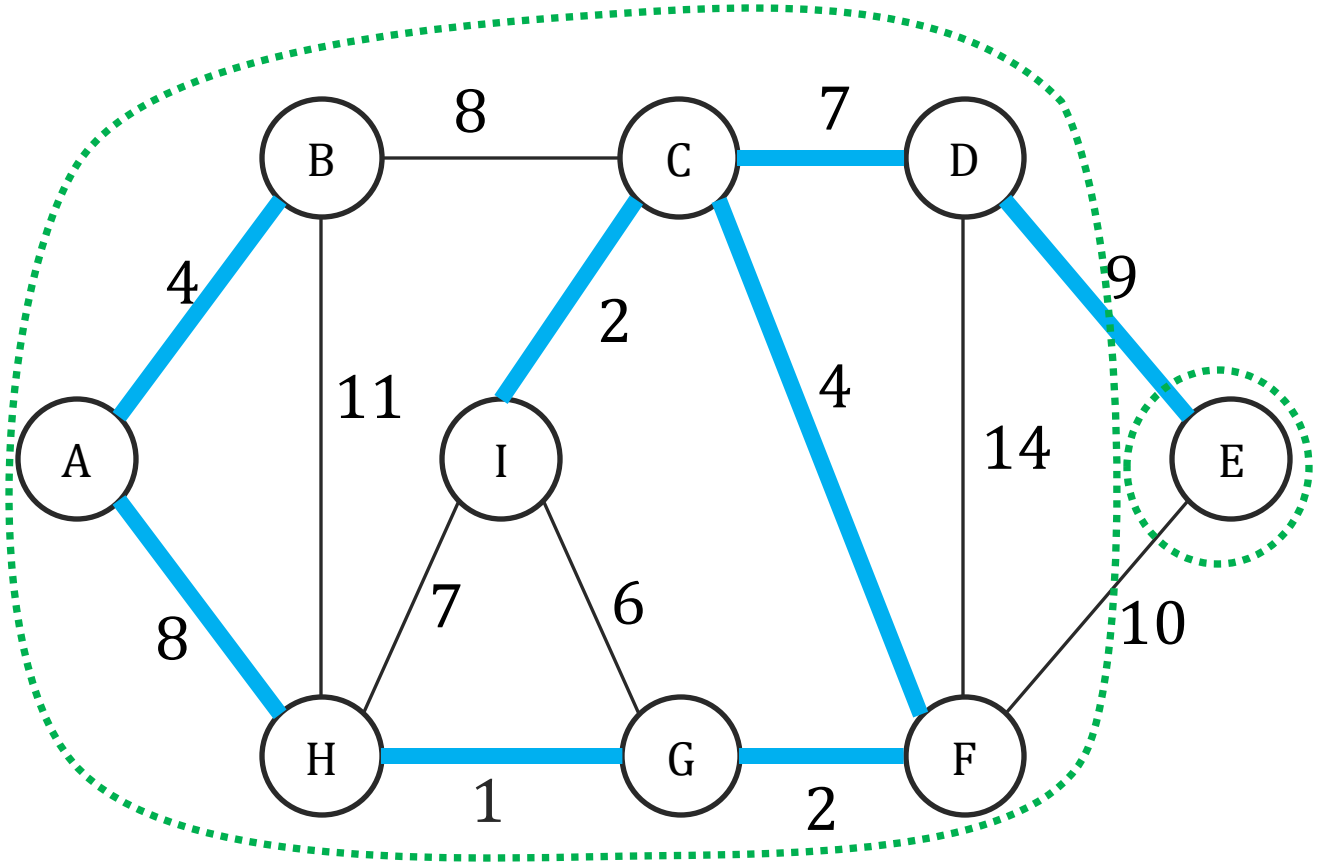


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

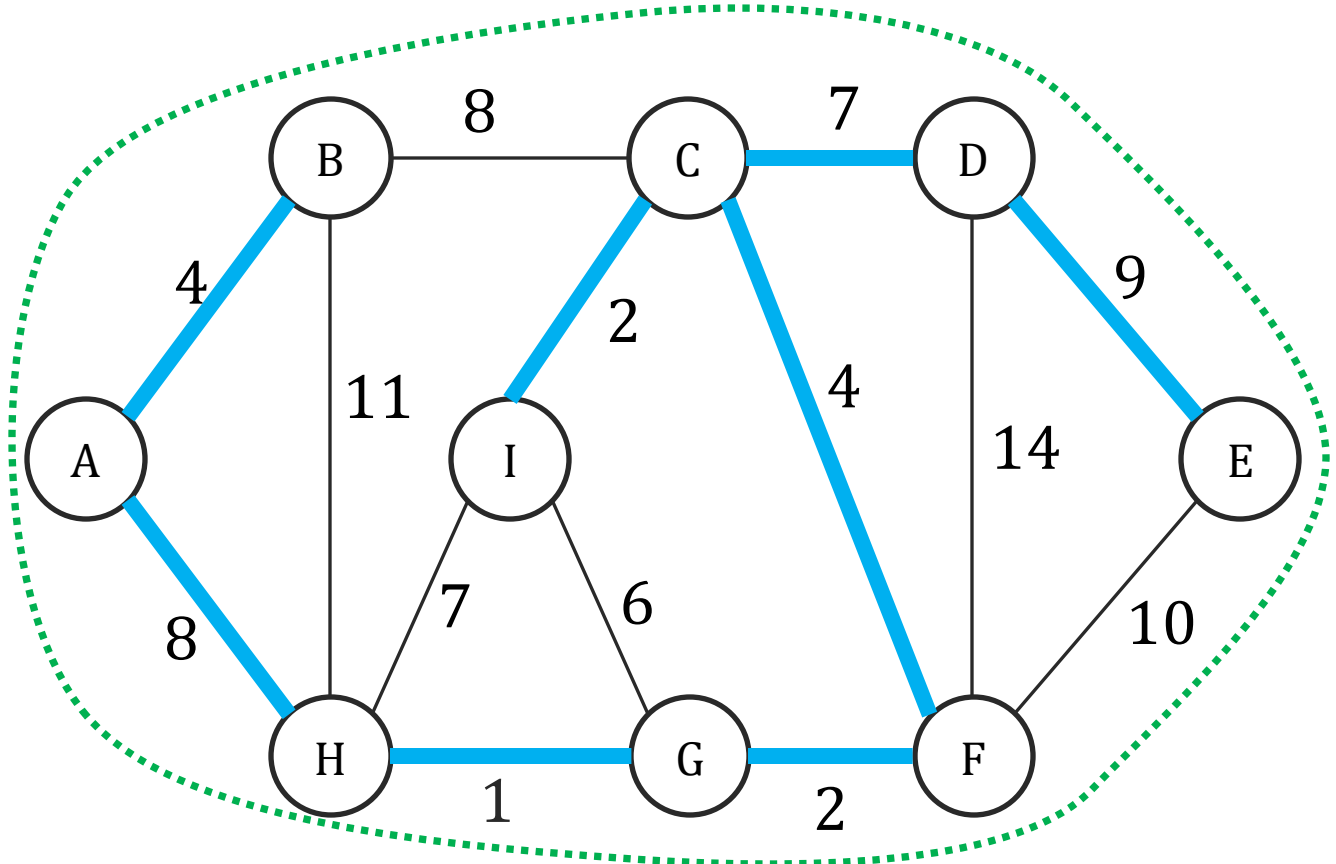


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

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.

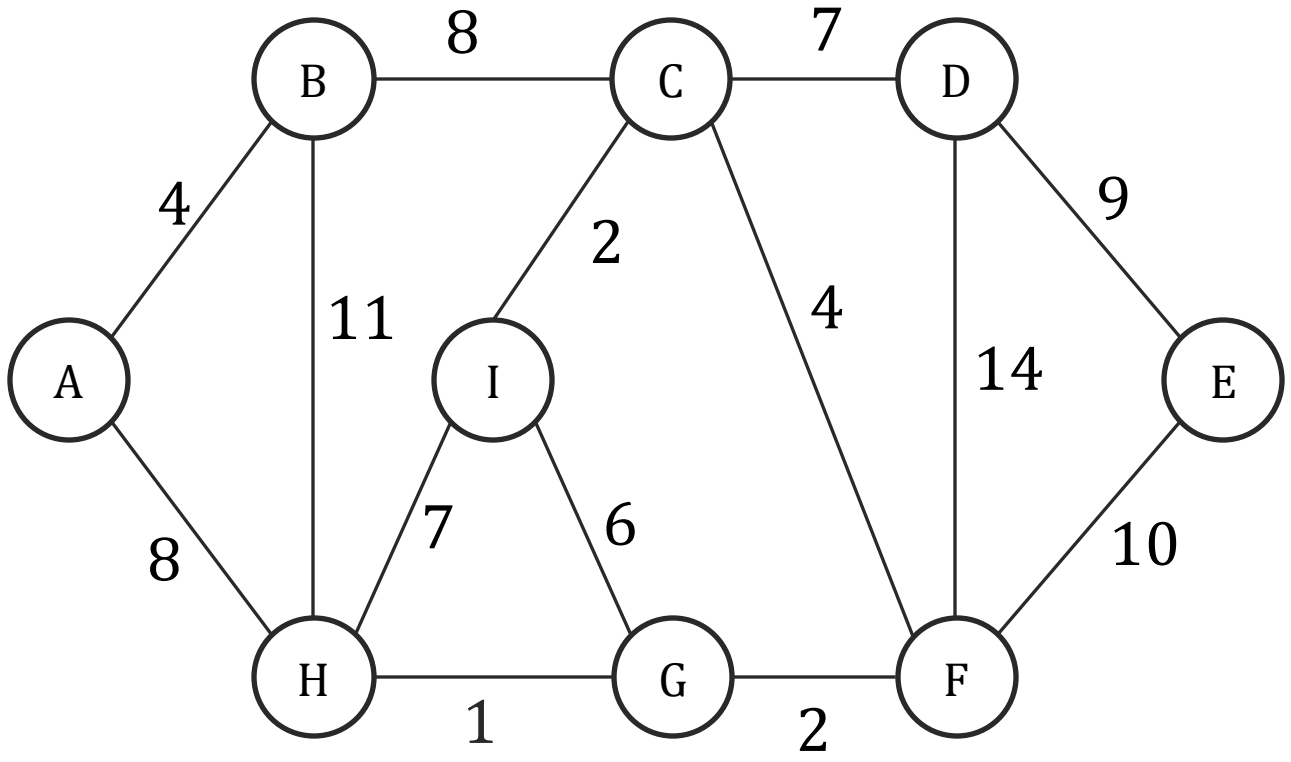
"Cut Property":
If X is a subset of an MST and has no edges from S to $V \setminus S$, then $X \cup \{e\}$ is also a subset of an MST.

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 $v \in V \setminus S$.

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

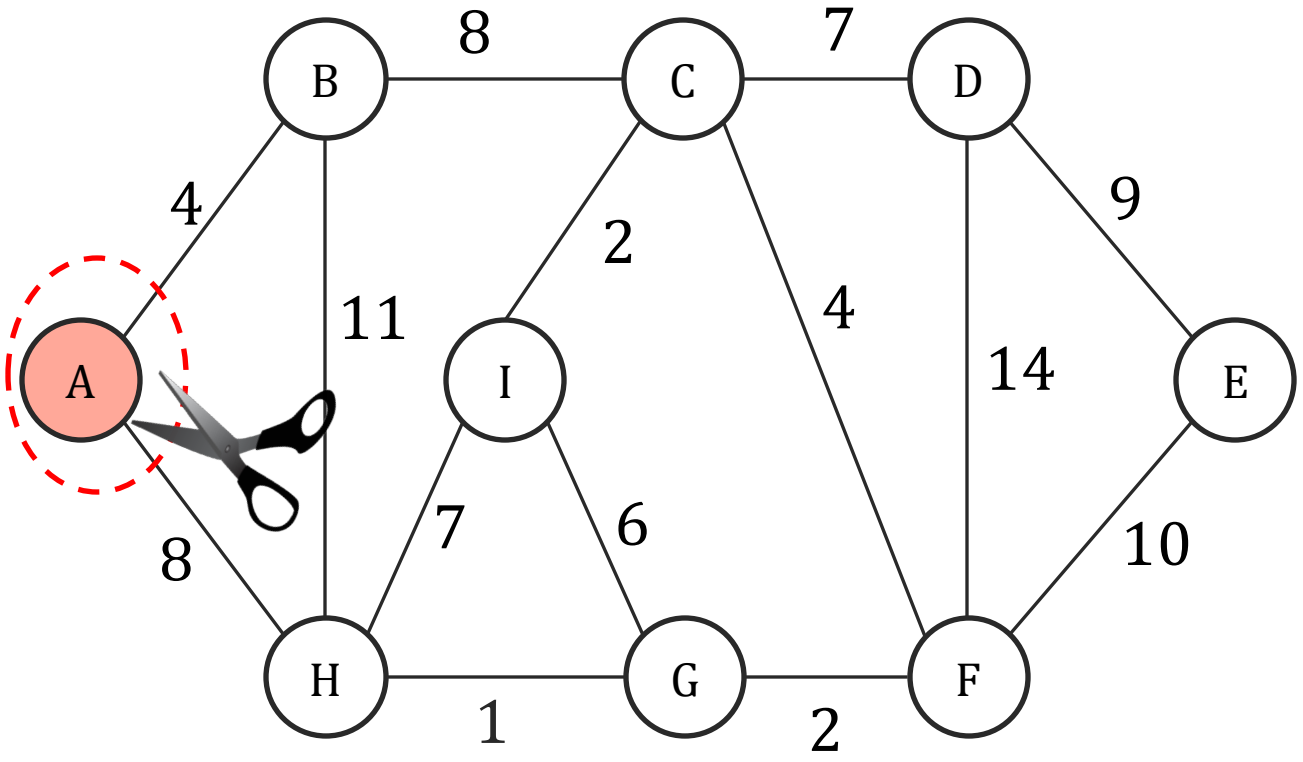
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

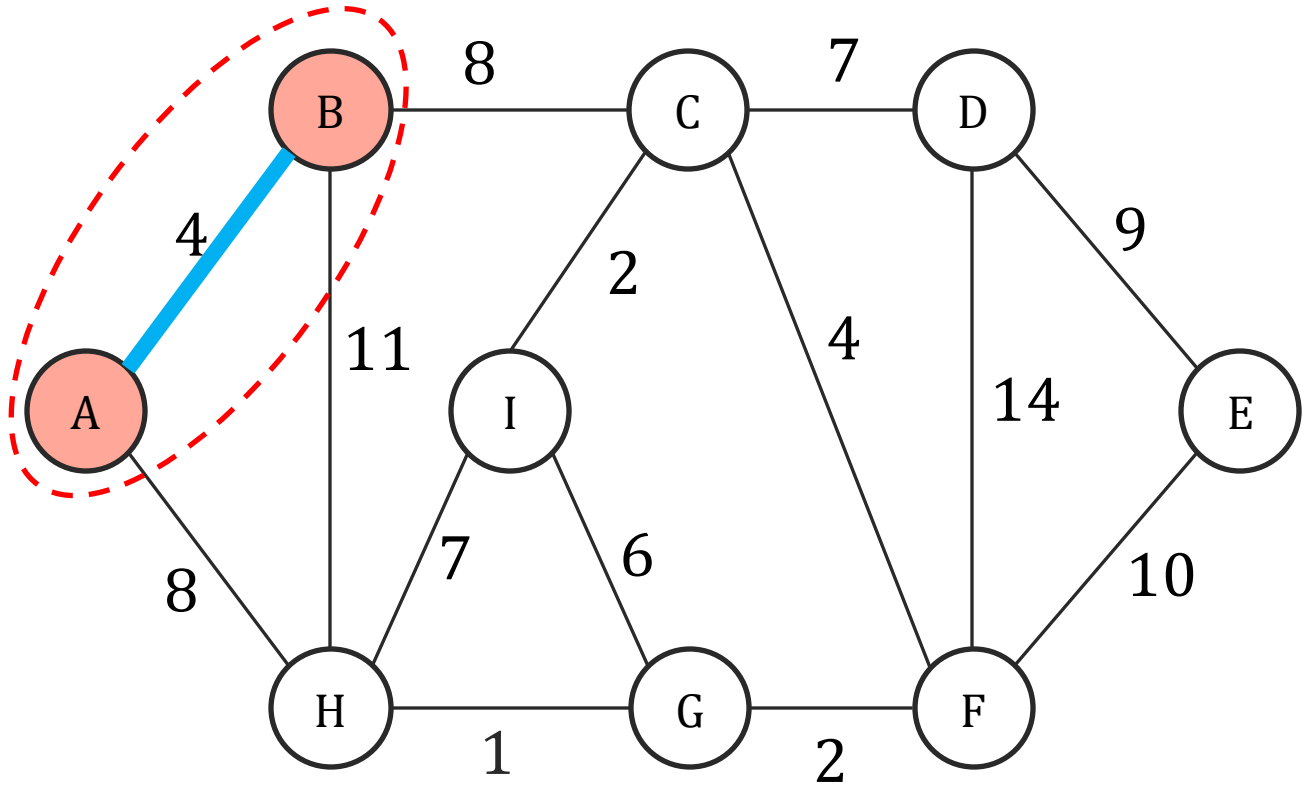
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

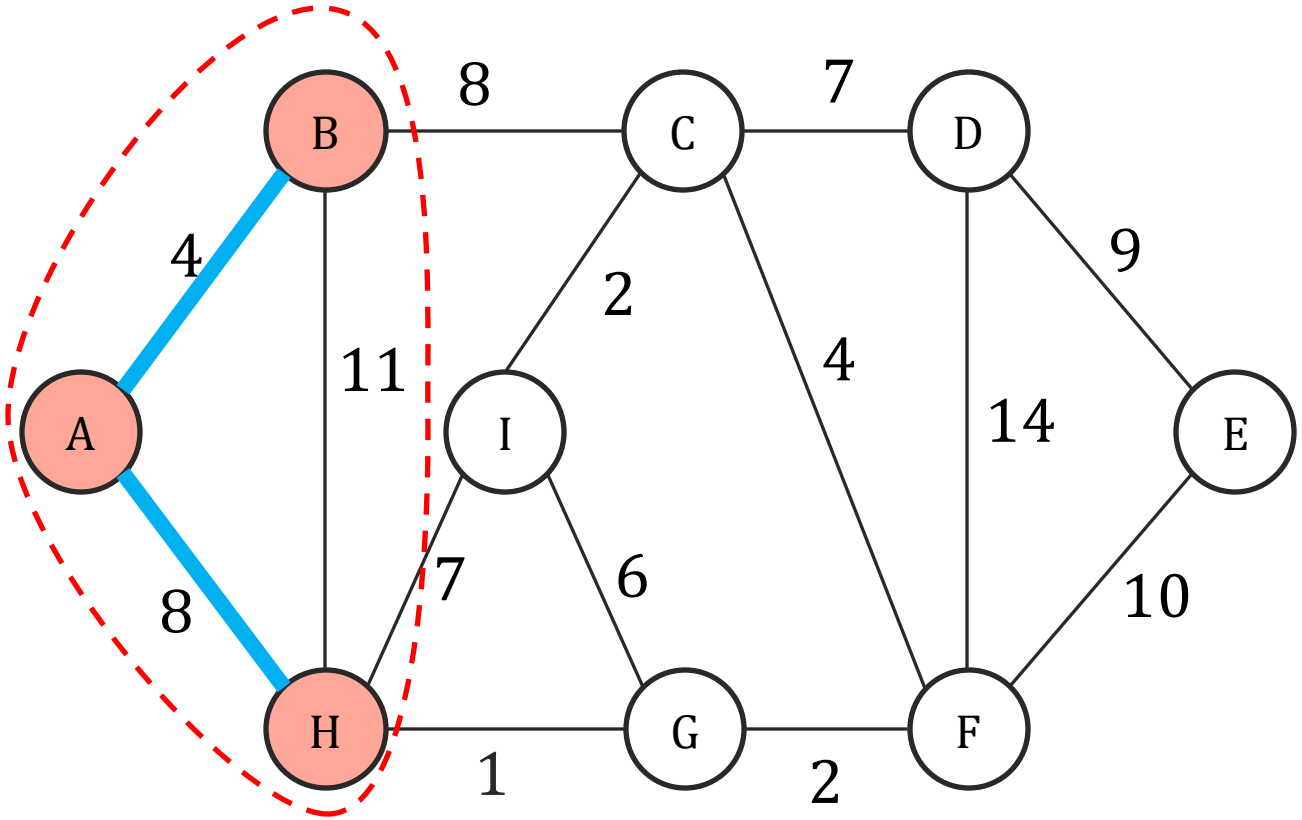
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

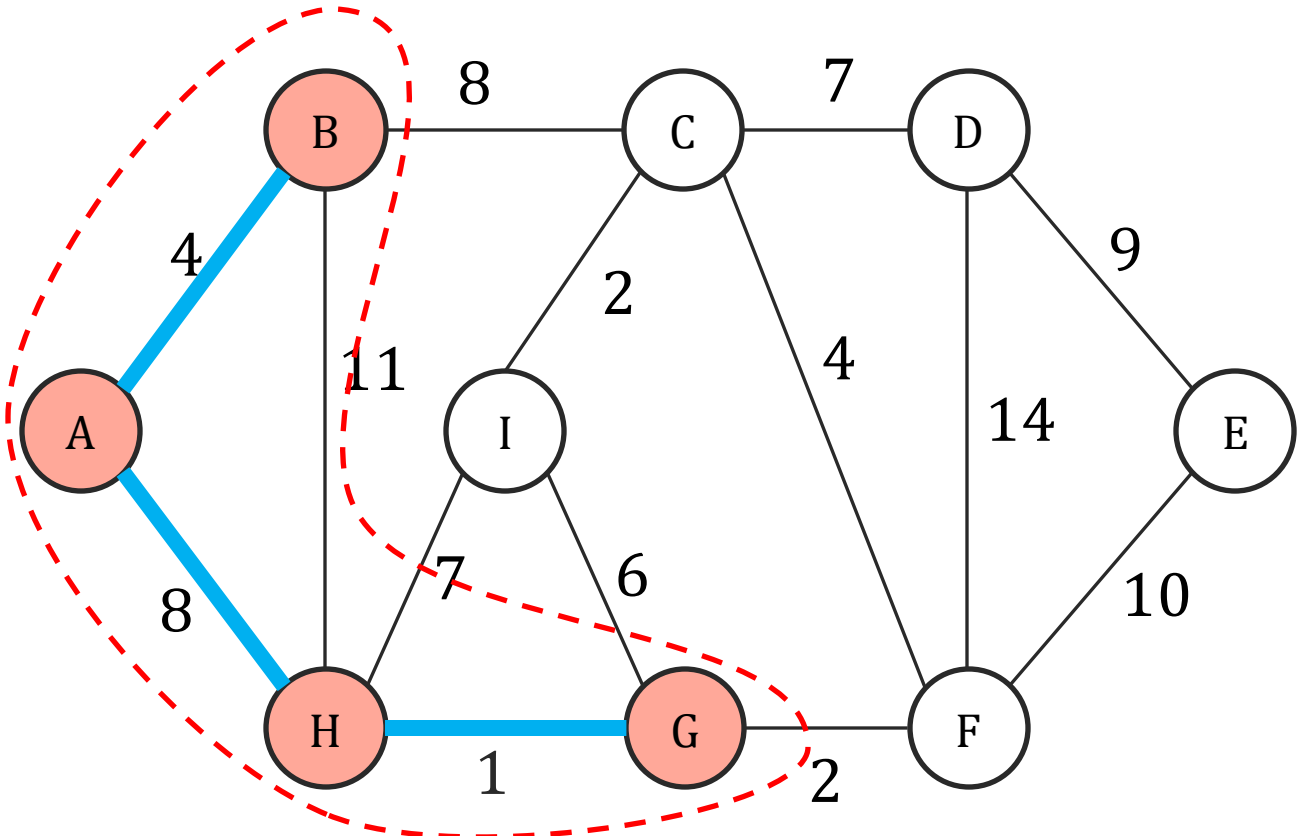
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

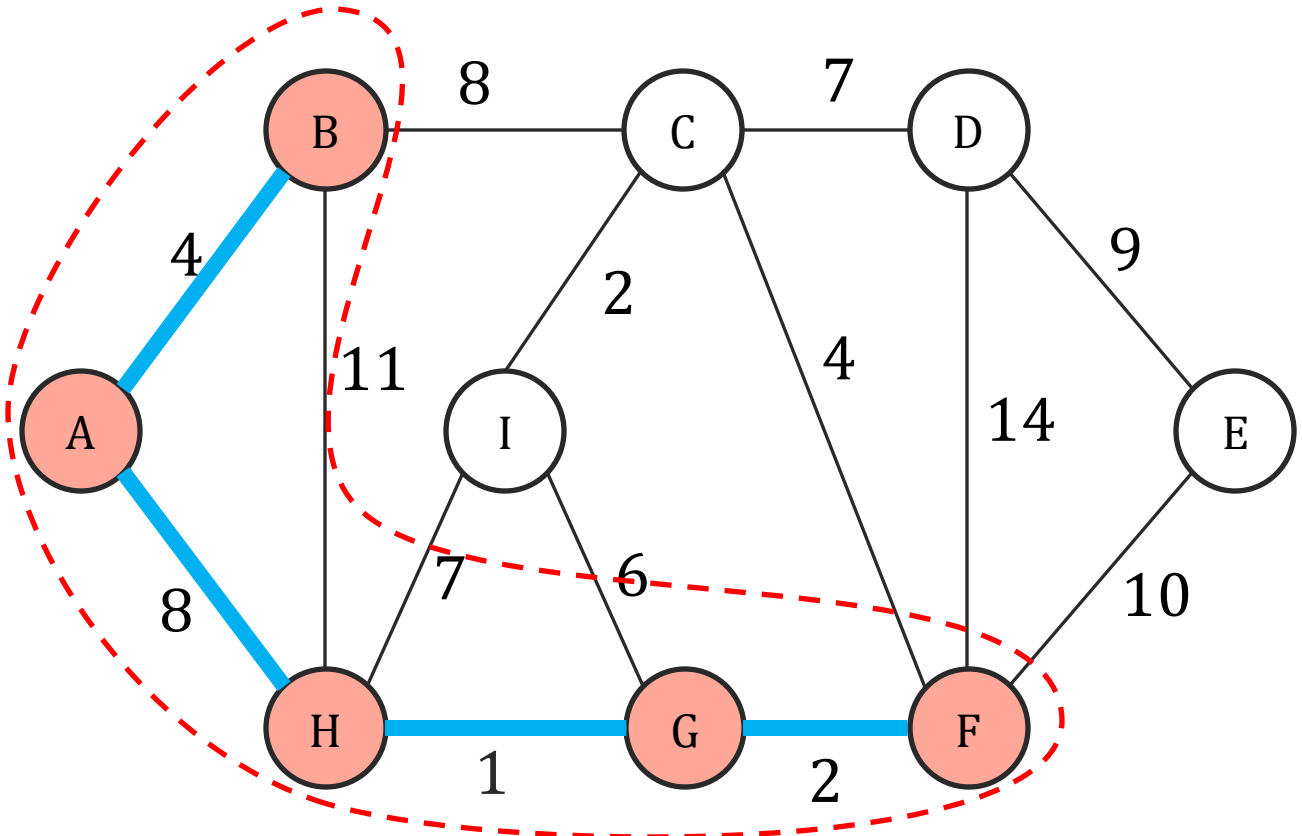
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

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

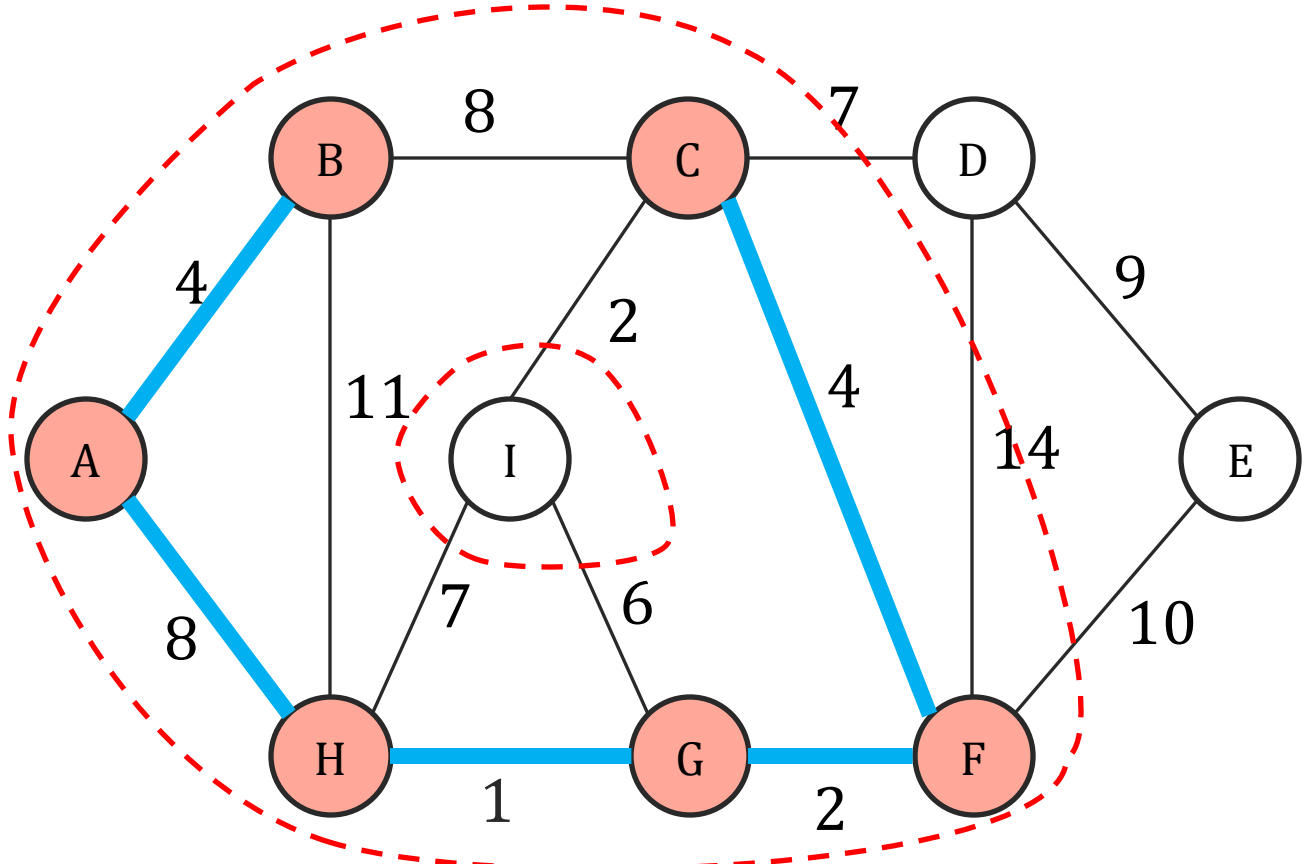
Return X

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 $v \in V \setminus S$.

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

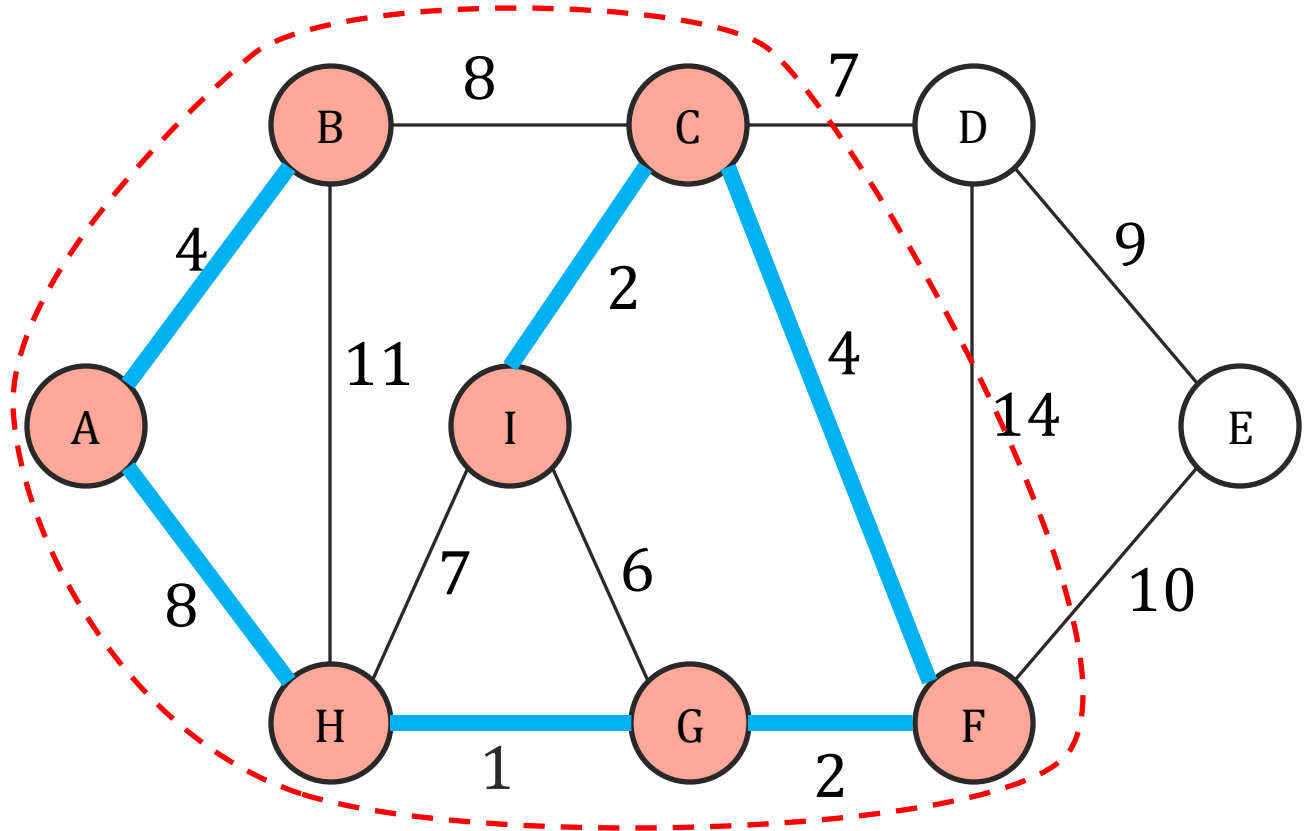
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

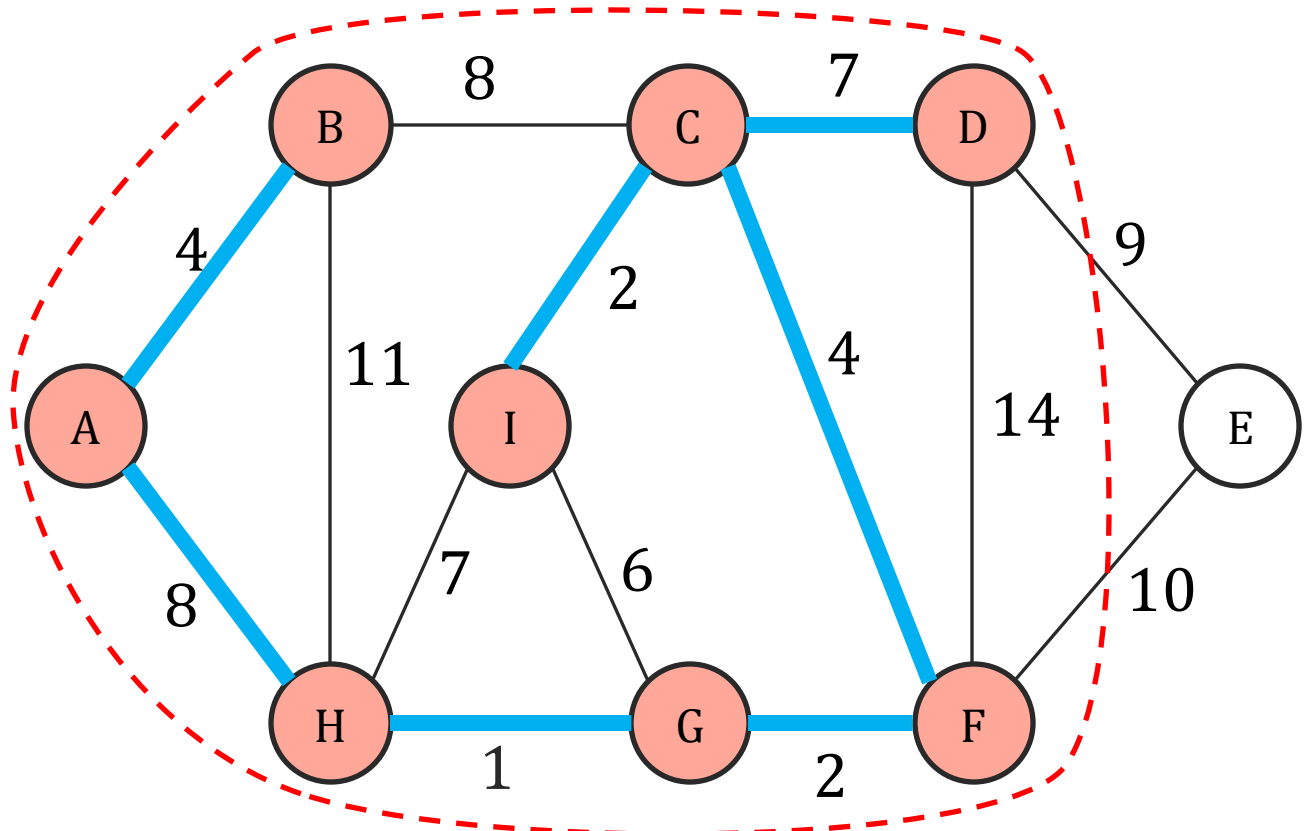
    $S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

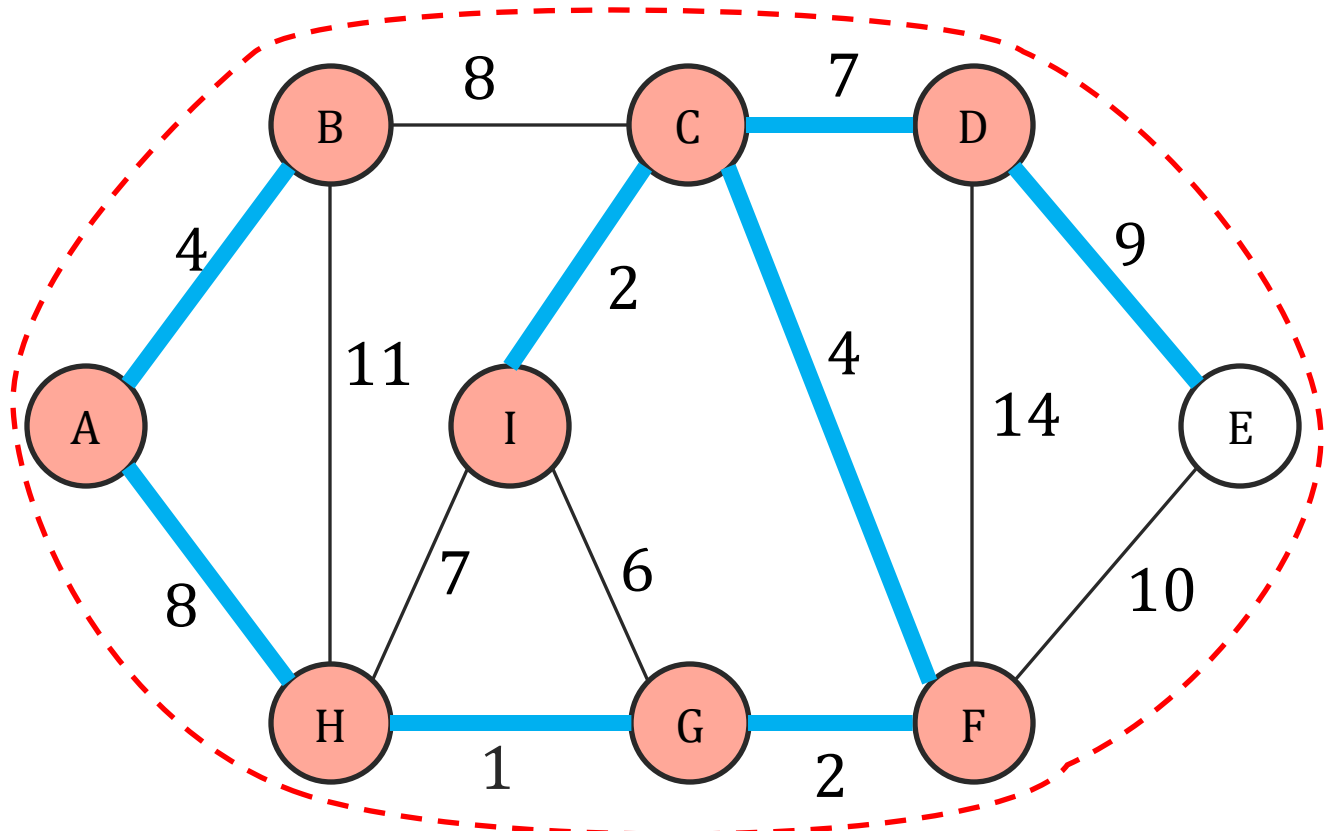
    $S \leftarrow S \cup \{v\}$

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 find 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.

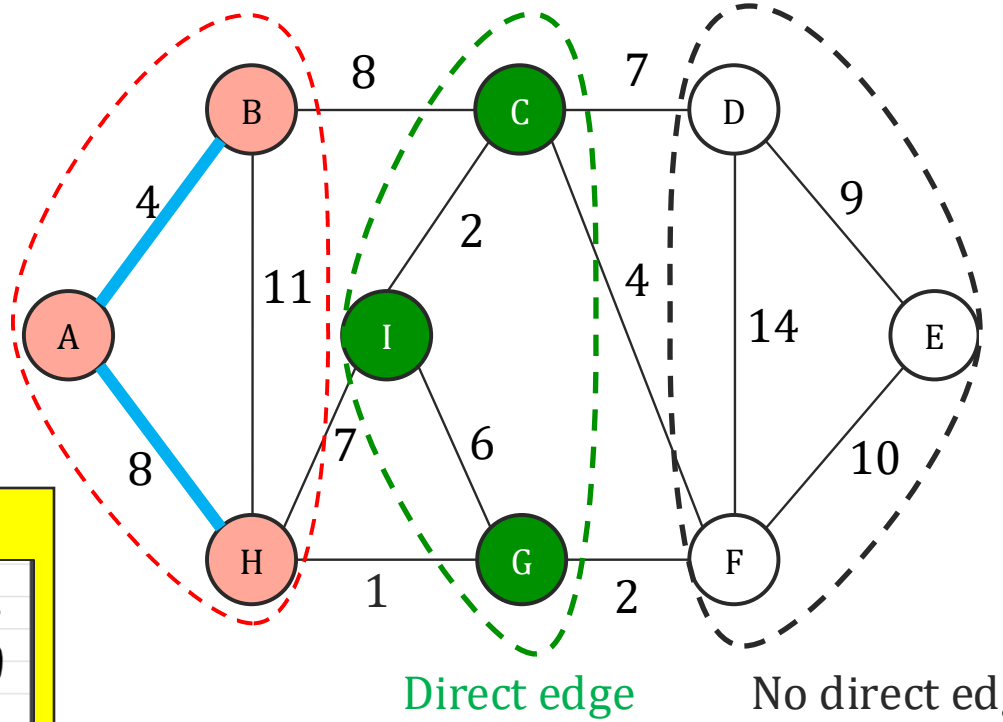
Naively implementing this, take $O(nm)$.

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

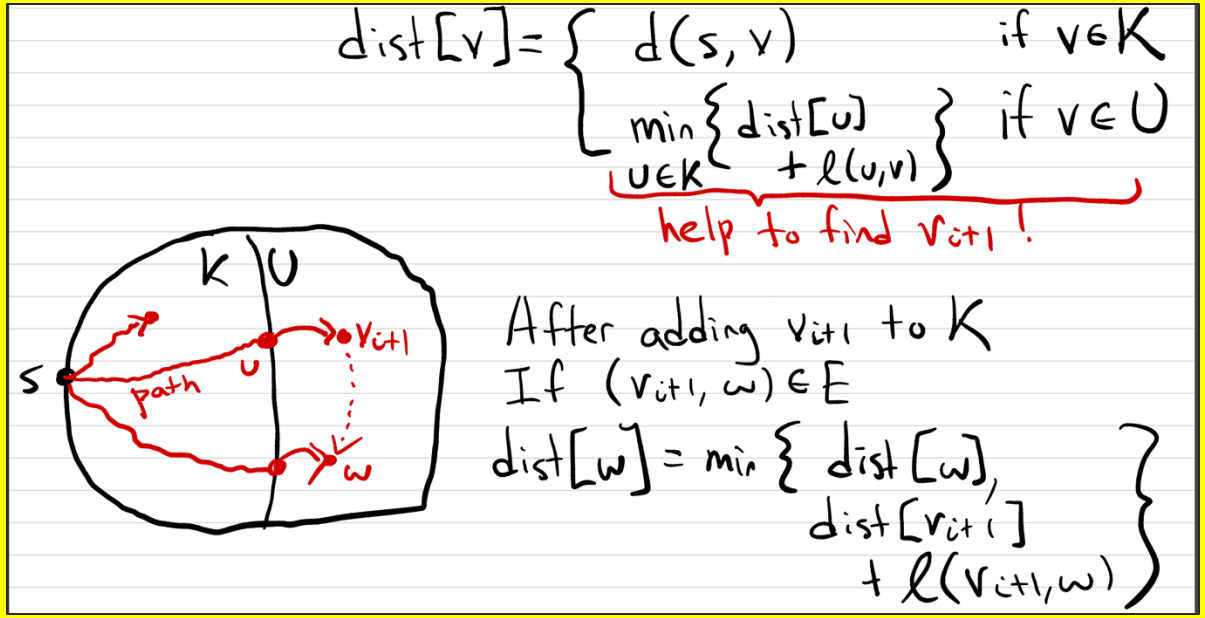
How do we actually implement Prim's Algorithm?

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 Prim's Algorithm Fast

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

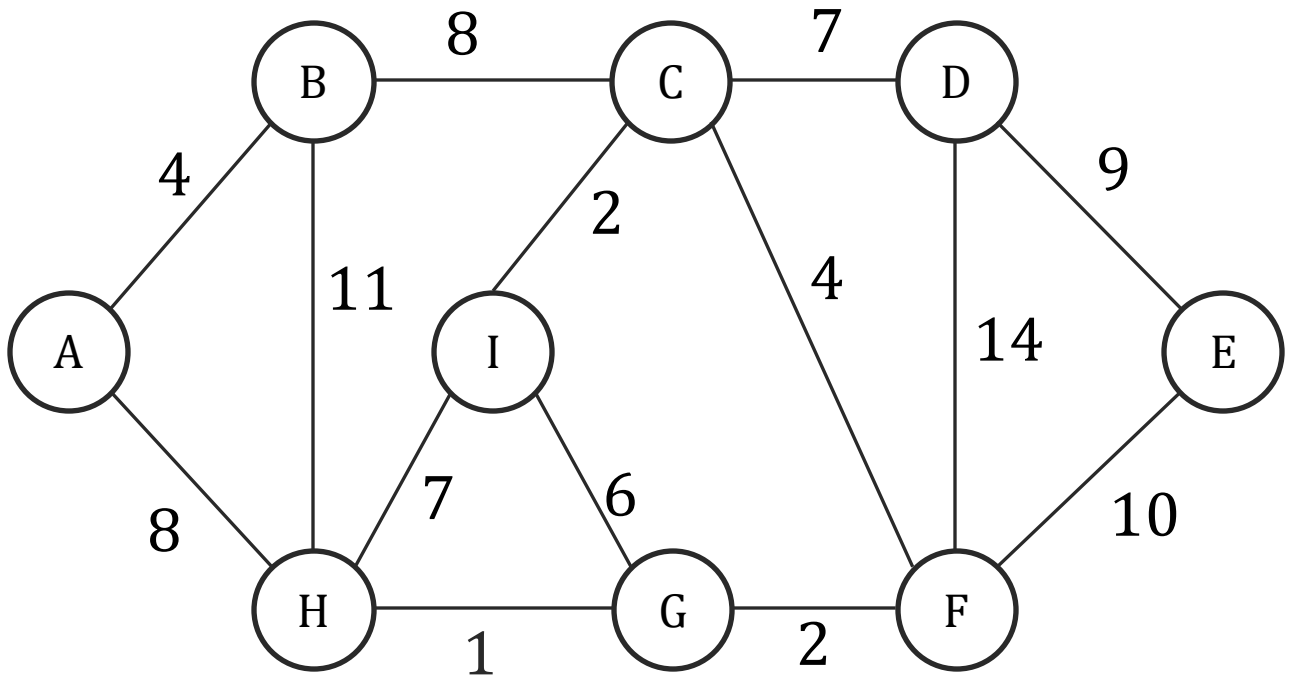
  update $prev[v] \leftarrow u$

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.

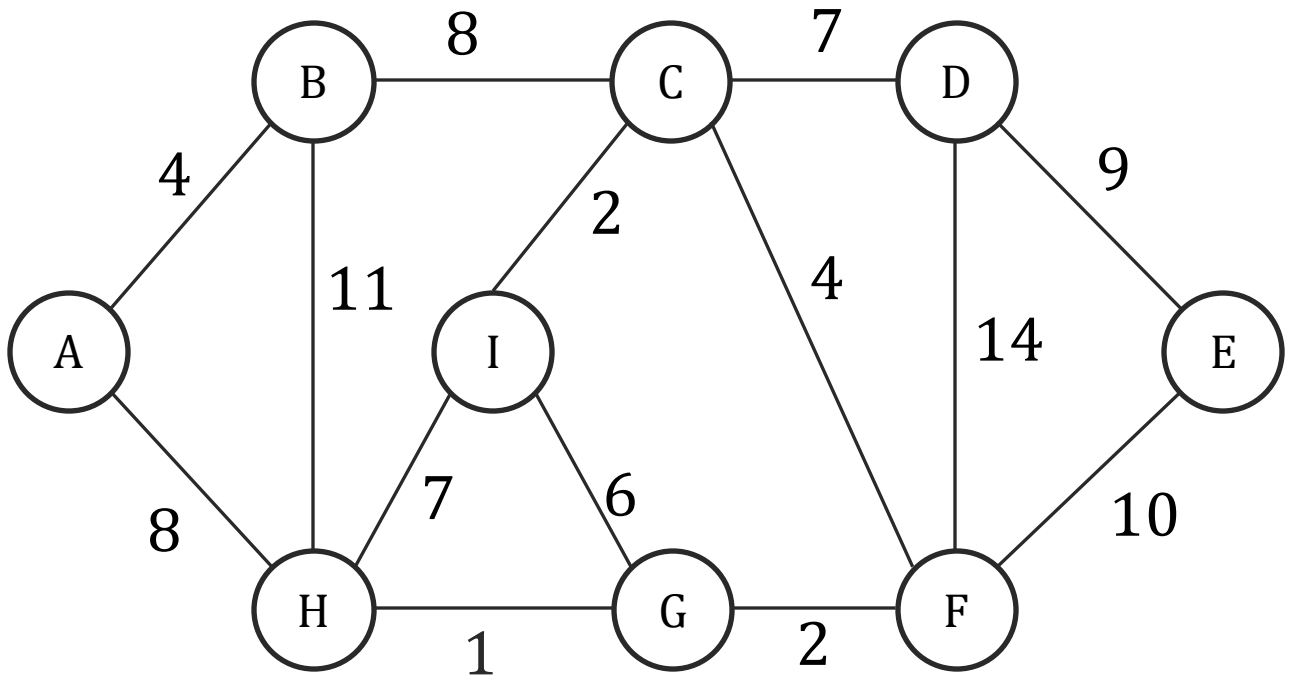
Prim's Algorithm: Efficient Implementation



	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 $v \in V$, 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 $z \in Q$.

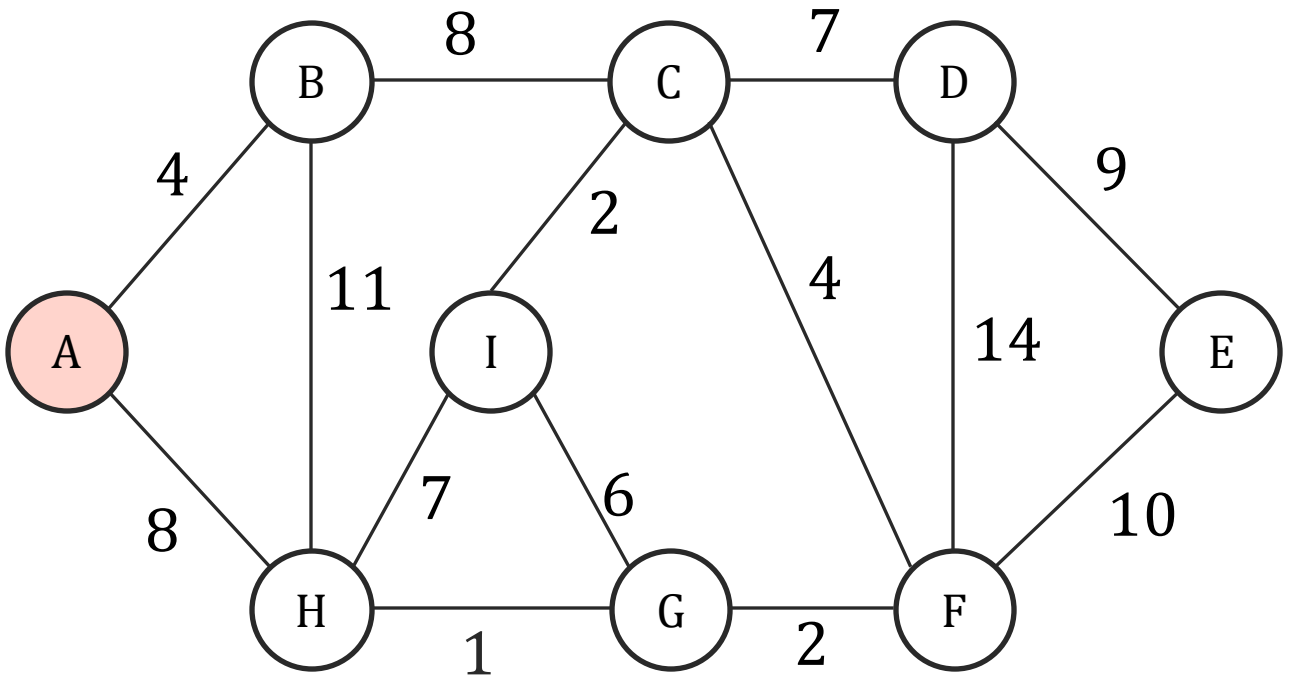
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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 $v \in V$, 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 $z \in Q$.

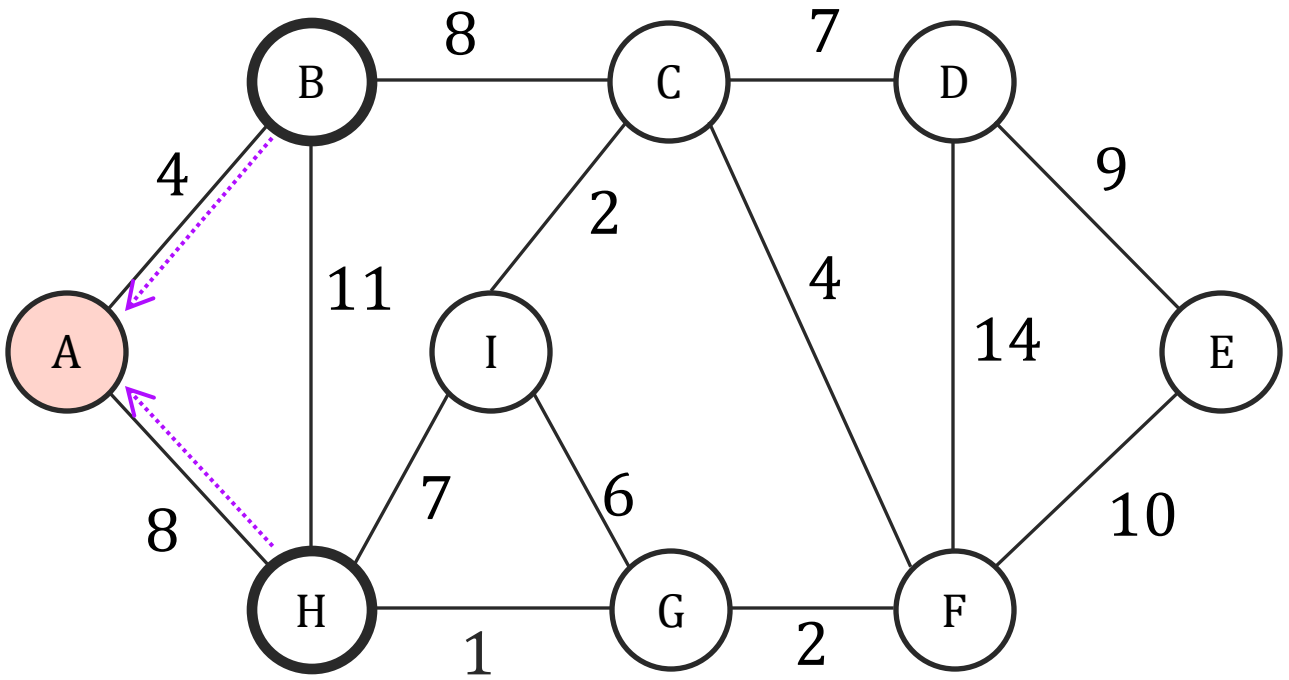
Q.decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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 $v \in V$, 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 $z \in Q$.

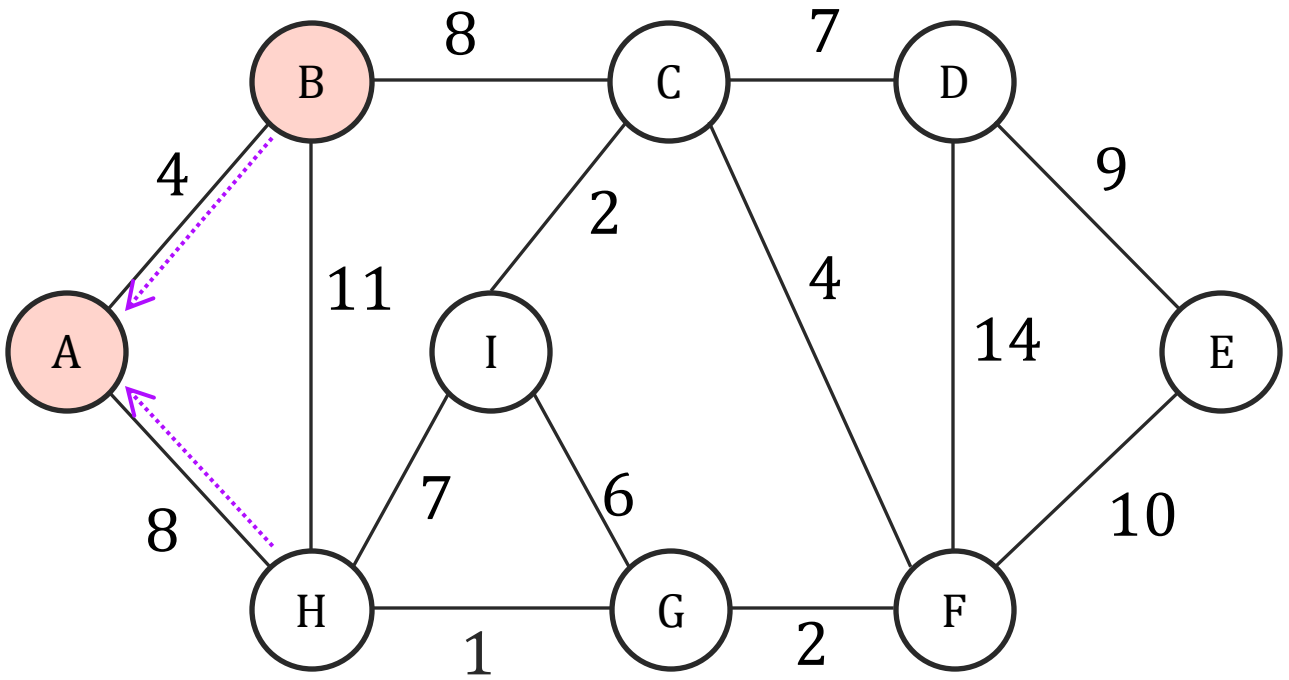
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

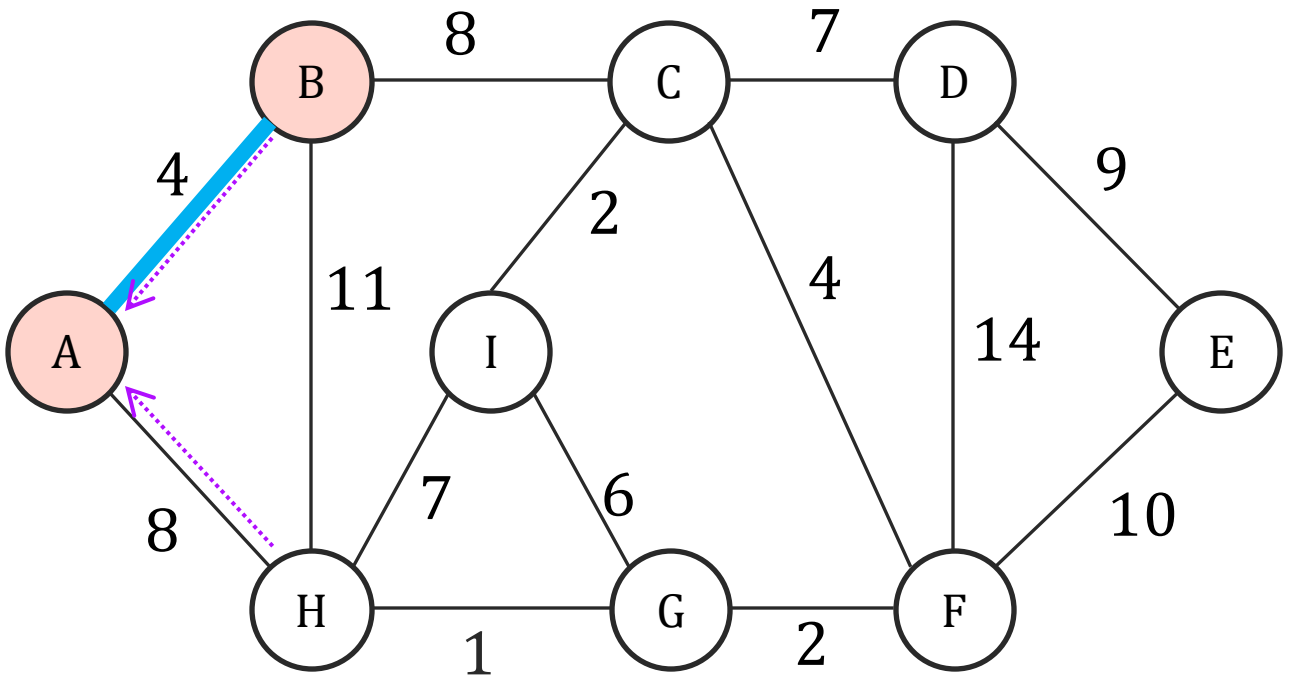


	A	B	C	D	E	F	G	H	I
$dist$	0	4	∞	∞	∞	∞	∞	8	∞
$prev$	∅	A	∅	∅	∅	∅	∅	A	∅

```
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 $prev$

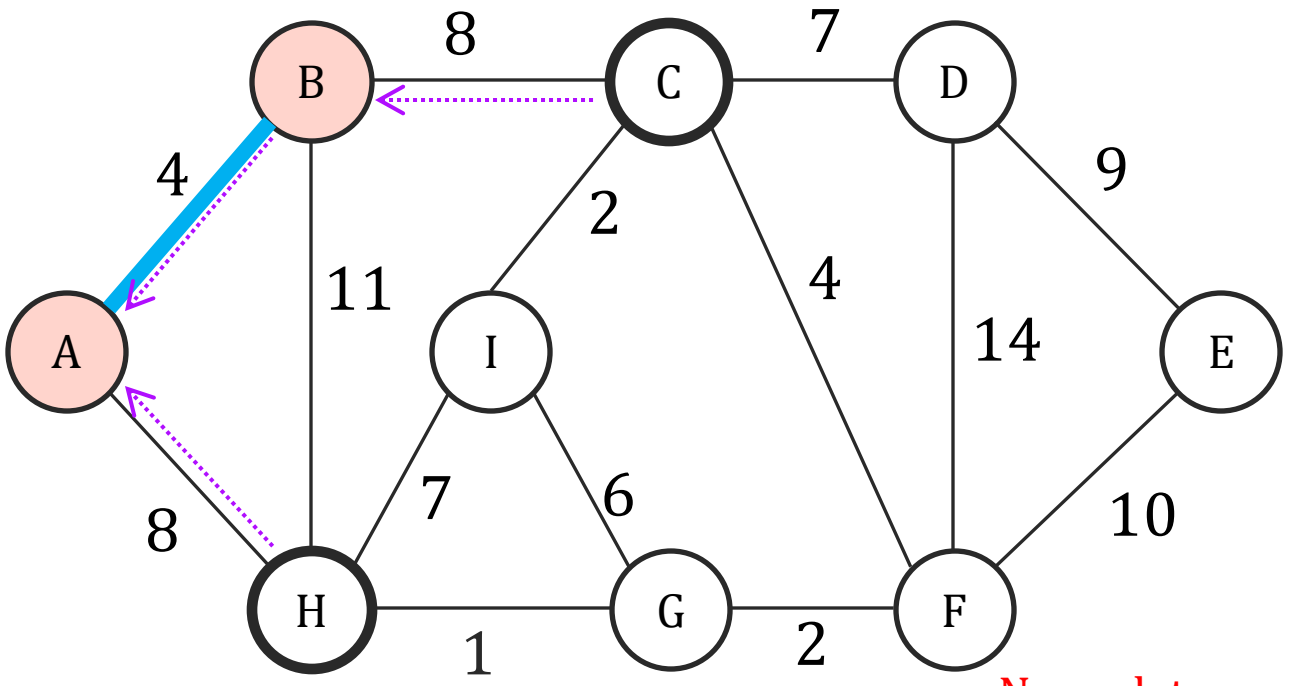


	A	B	C	D	E	F	G	H	I
$dist$	0	4	∞	∞	∞	∞	∞	8	∞
$prev$	∅	A	∅	∅	∅	∅	∅	A	∅

```
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 $v \in V$, 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 $z \in Q$.

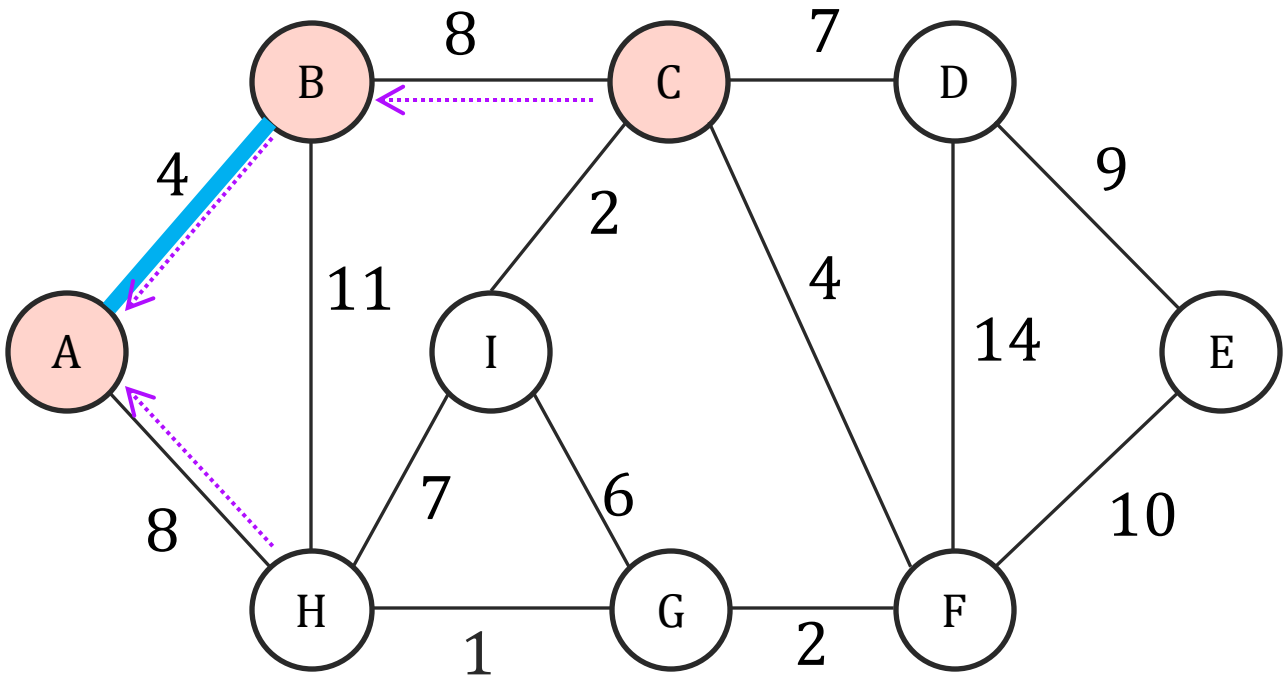
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

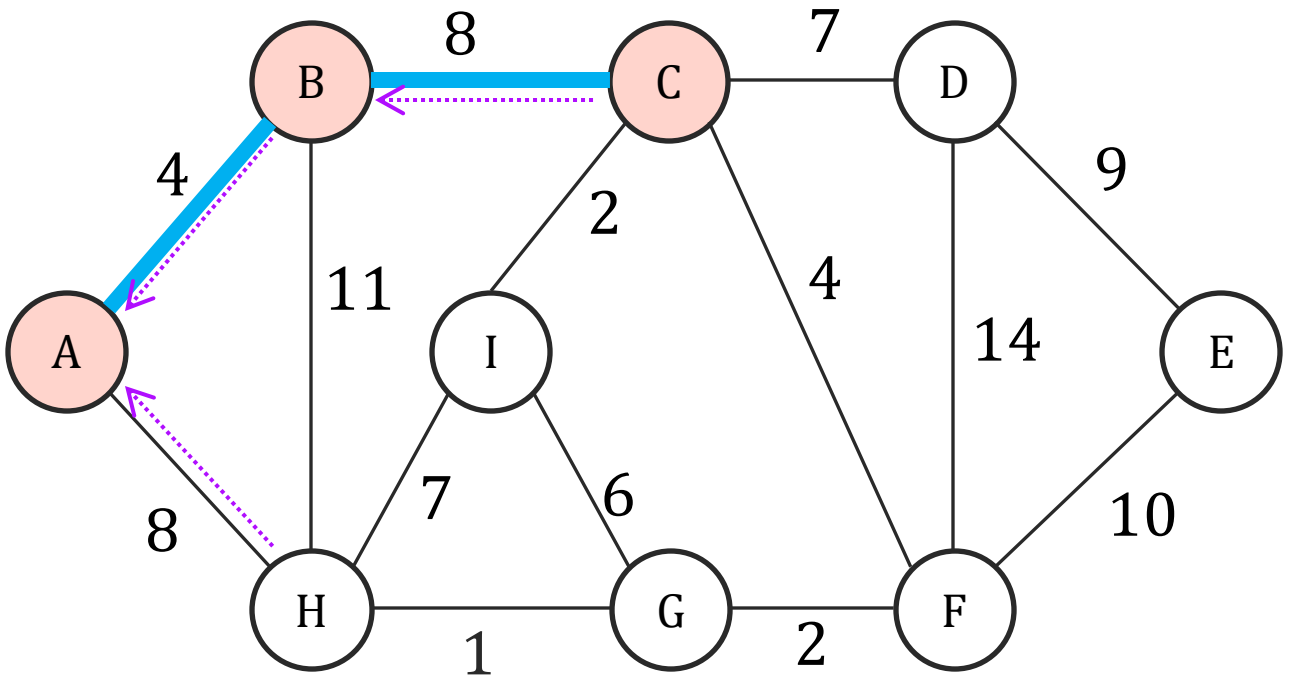


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	∞	∞	∞	∞	8	∞
$prev$	∅	A	B	∅	∅	∅	∅	A	∅

```
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 $prev$

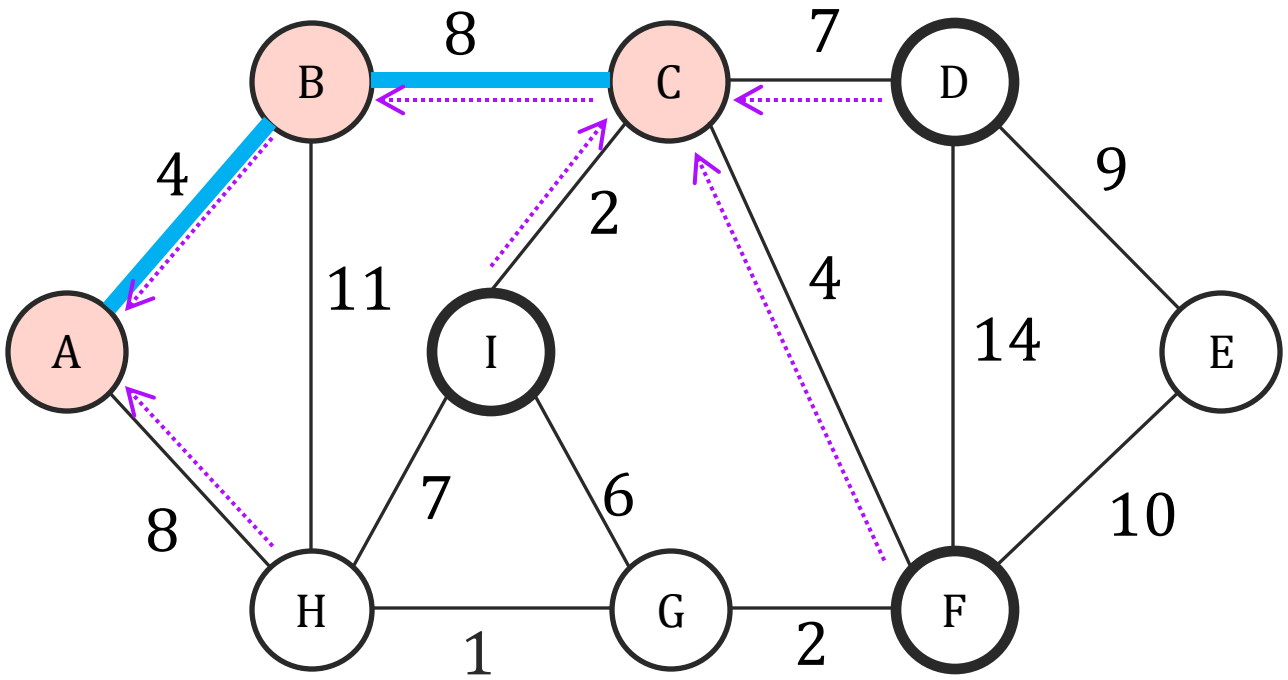


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	∞	∞	∞	∞	8	∞
$prev$	∅	A	B	∅	∅	∅	∅	A	∅

```
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 $v \in V$, 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 $z \in Q$.

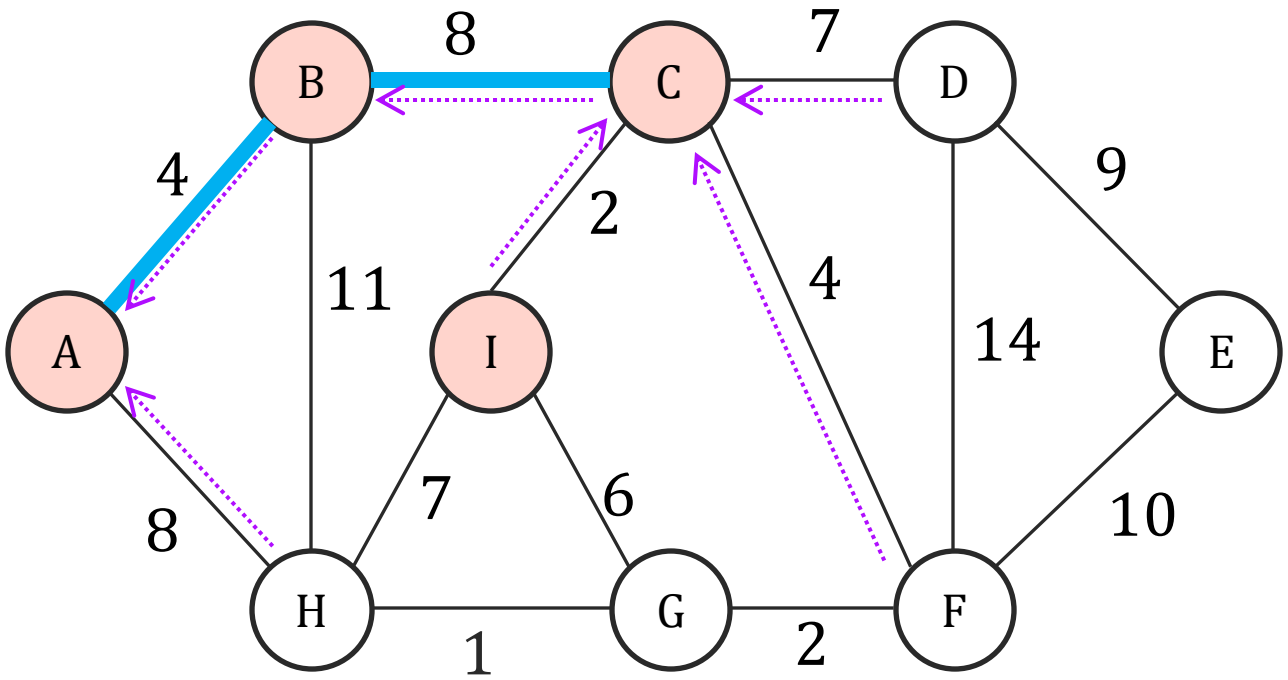
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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
$prev$	$\emptyset$	A	B	C	$\emptyset$	C	$\emptyset$	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 $v \in V$, 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 $z \in Q$.

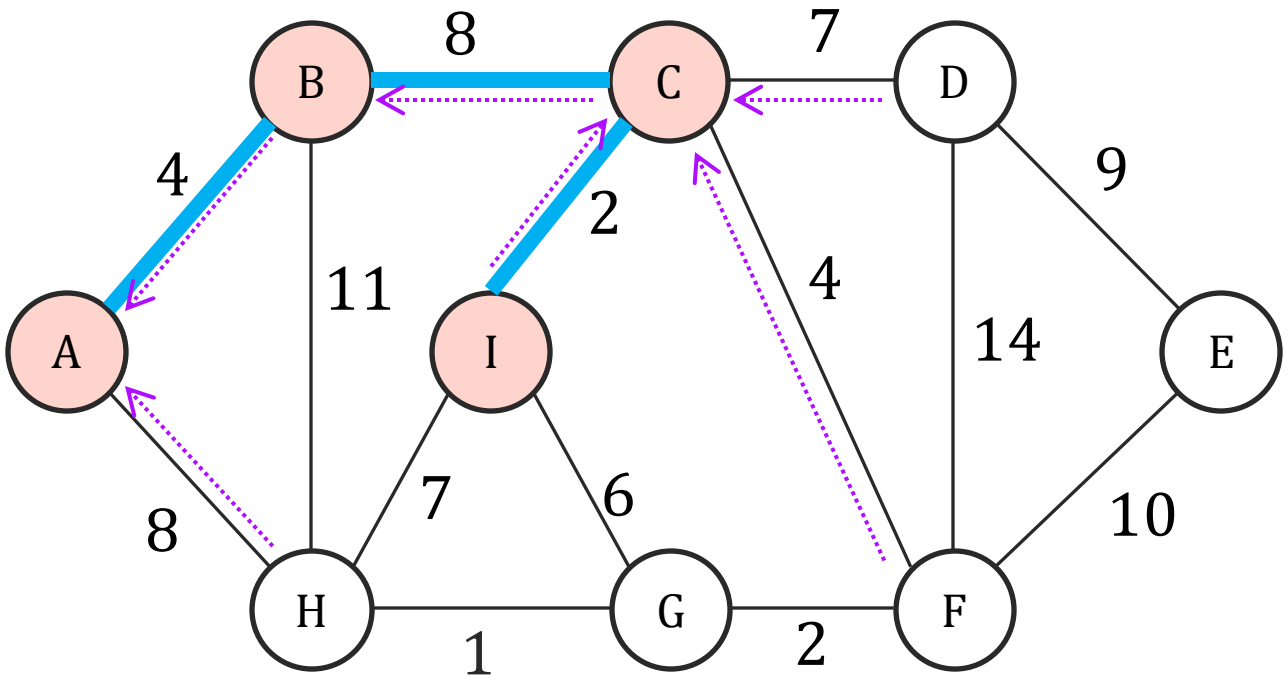
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

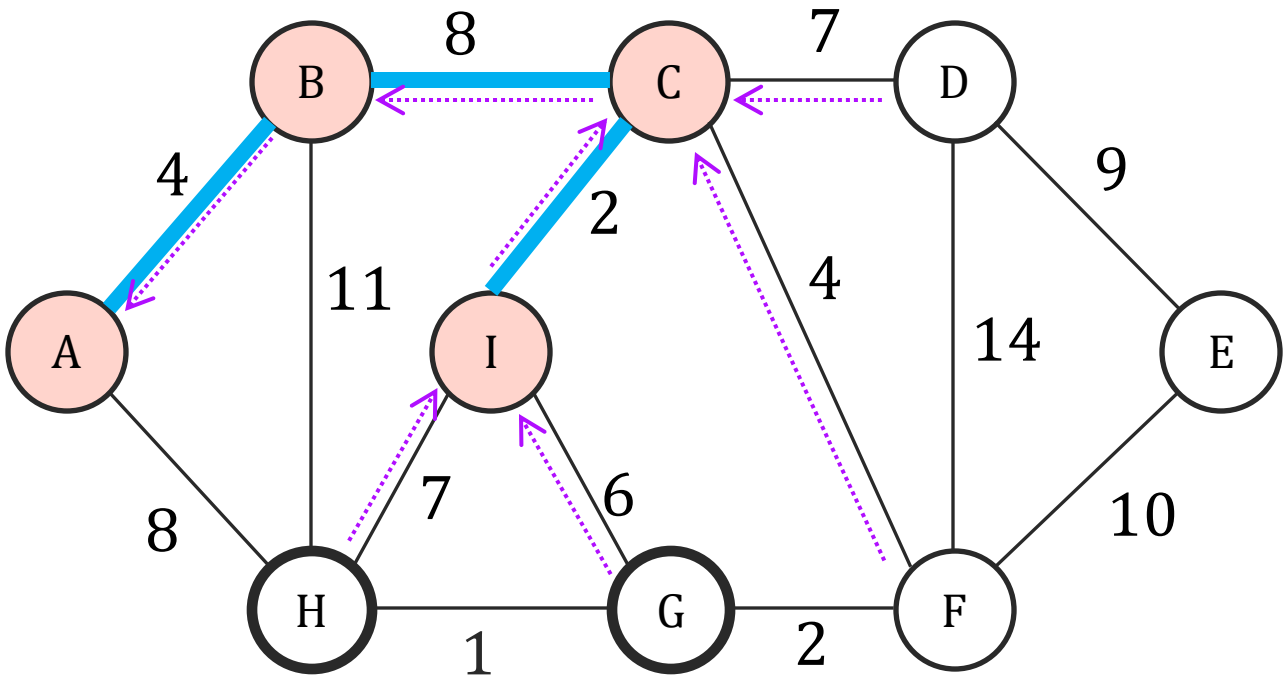


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	∞	4	∞	8	2
$prev$	$\emptyset$	A	B	C	$\emptyset$	C	$\emptyset$	A	C

```
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 $v \in V$, 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 $z \in Q$.

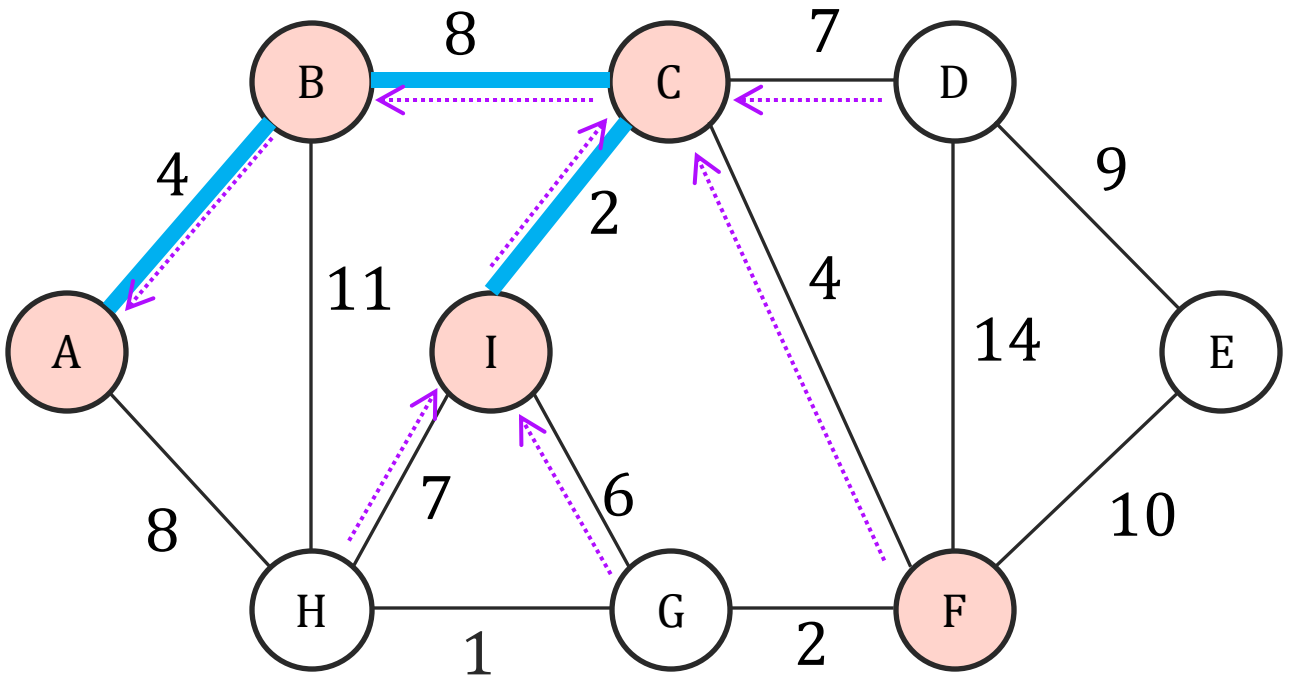
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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 $v \in V$, 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 $z \in Q$.

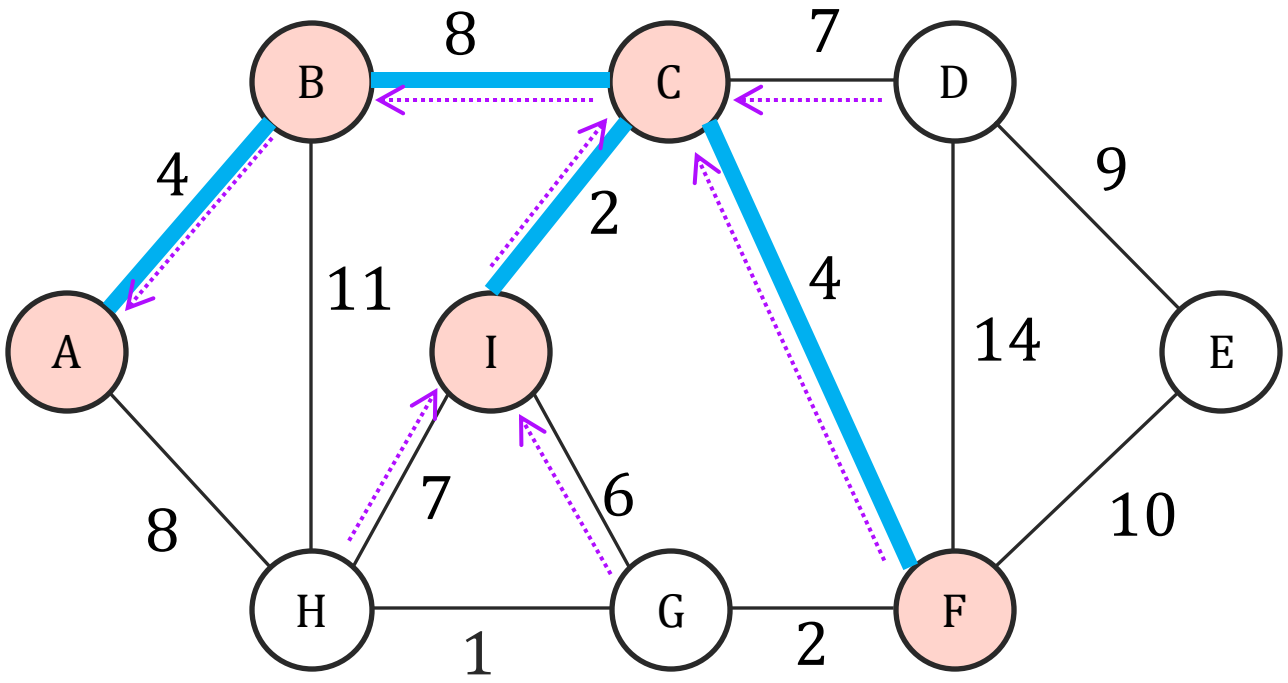
Q.decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

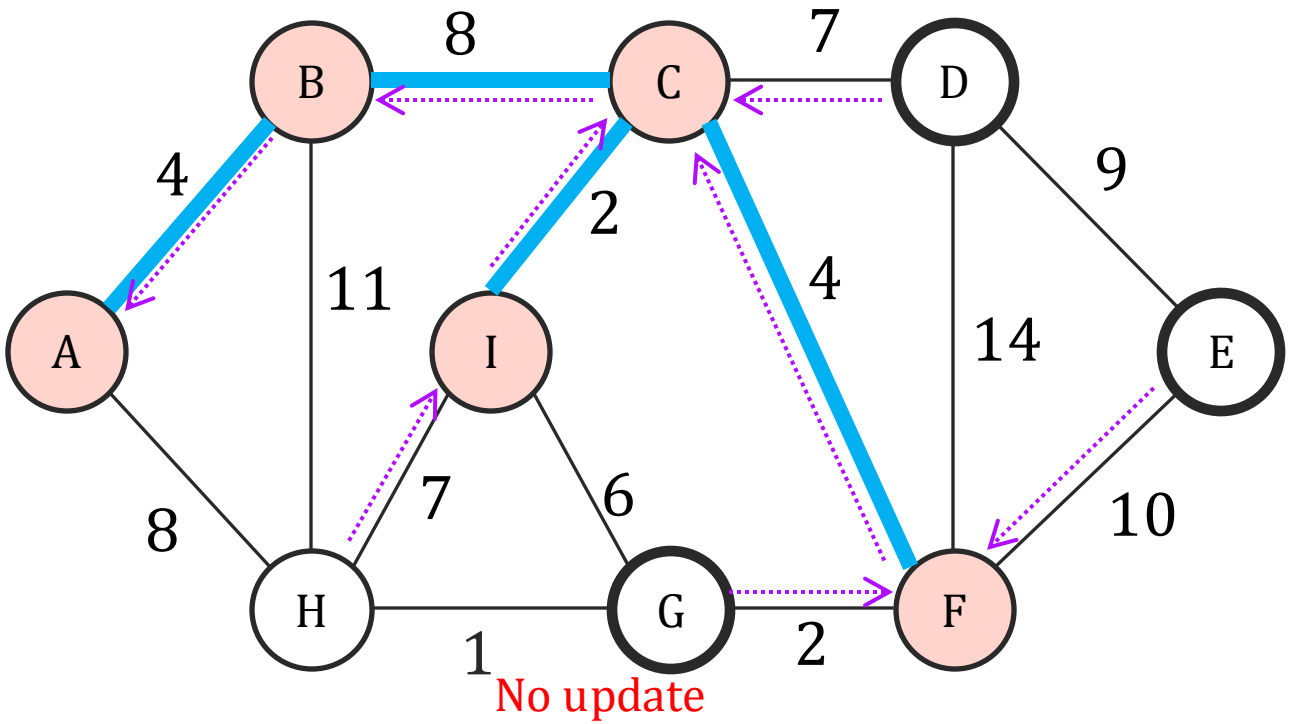


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	∞	4	6	7	2
$prev$	$\emptyset$	A	B	C	$\emptyset$	C	I	I	C

```
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
$prev$	$\emptyset$	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 $v \in V$, 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 $z \in Q$.

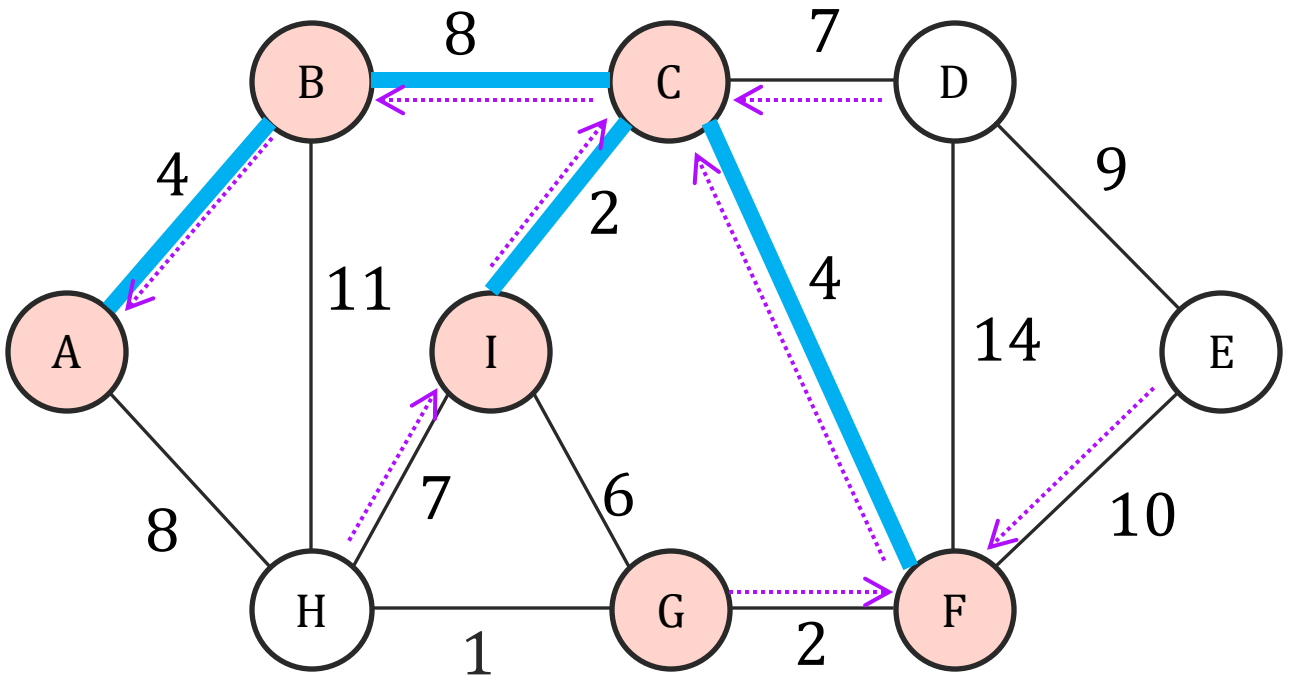
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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
$prev$	$\emptyset$	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 $v \in V$, 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 $z \in Q$.

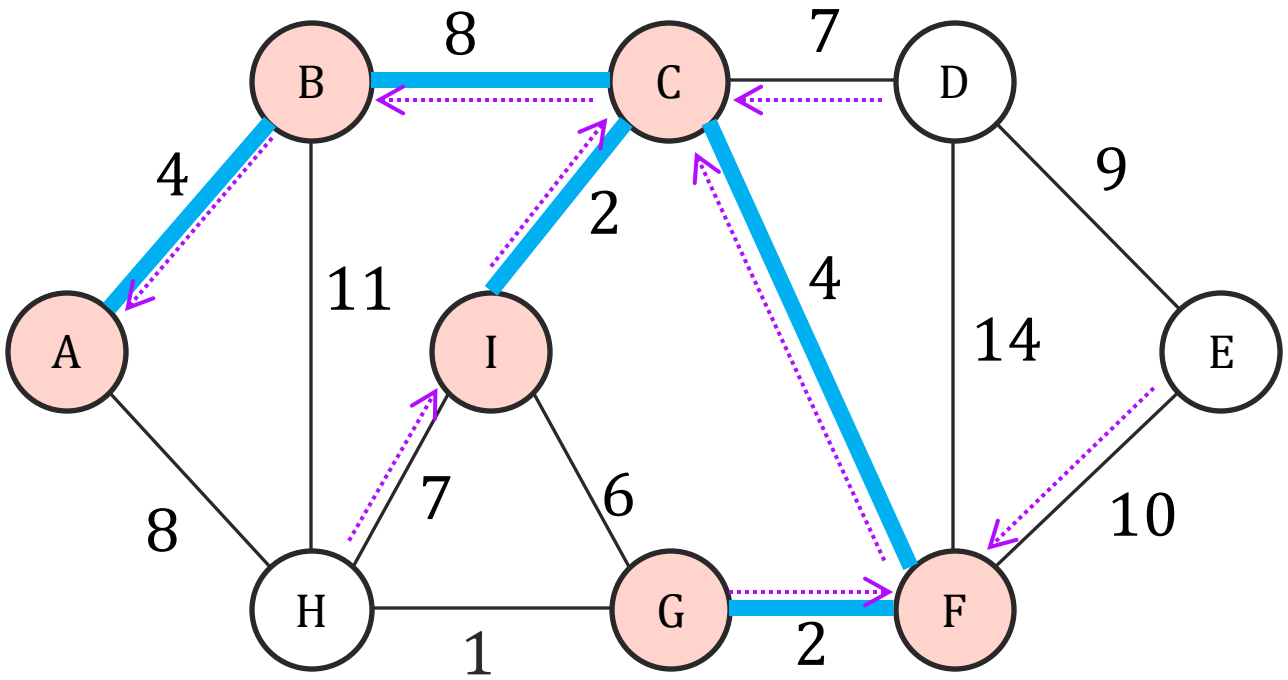
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

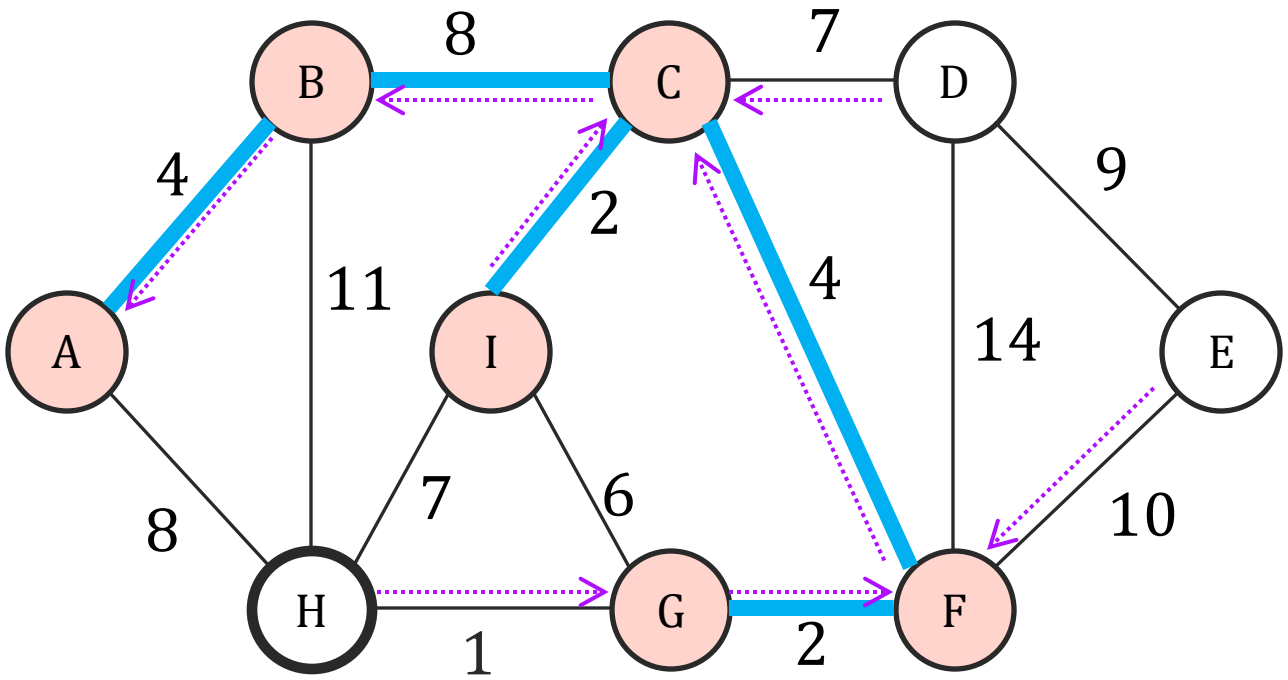


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	7	2
$prev$	$\emptyset$	A	B	C	F	C	F	I	C

```
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 $v \in V$, 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 $z \in Q$.

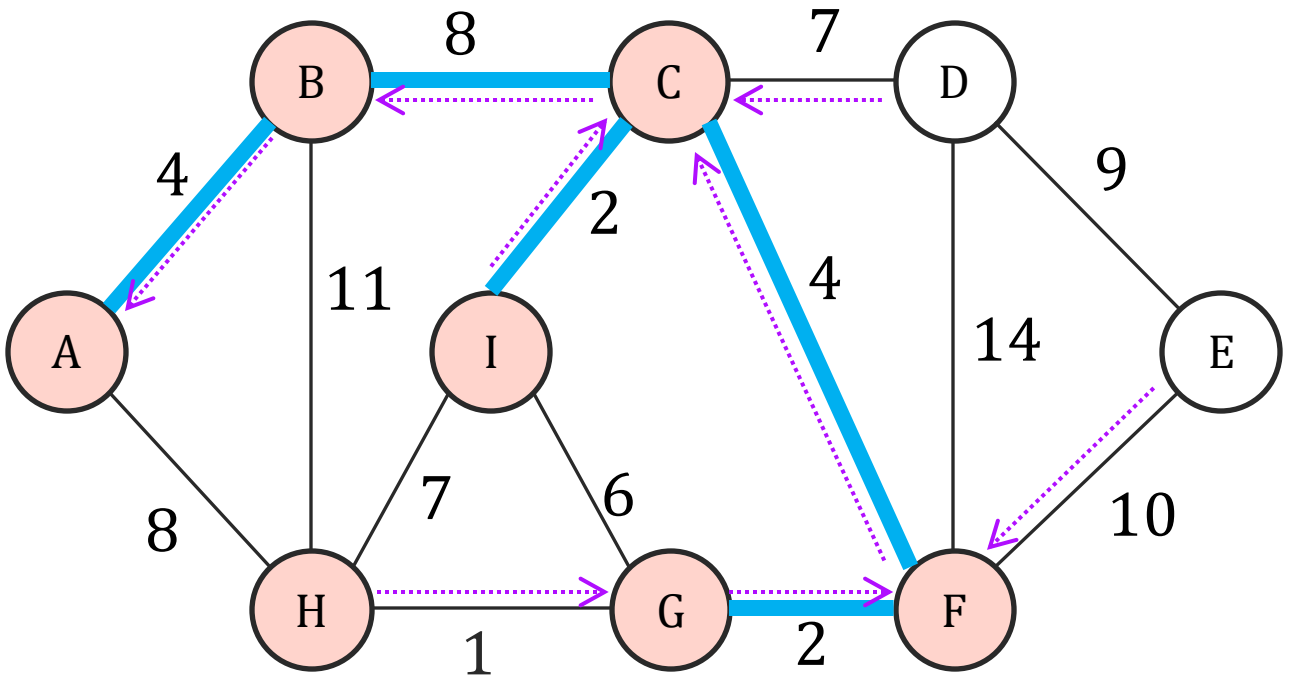
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

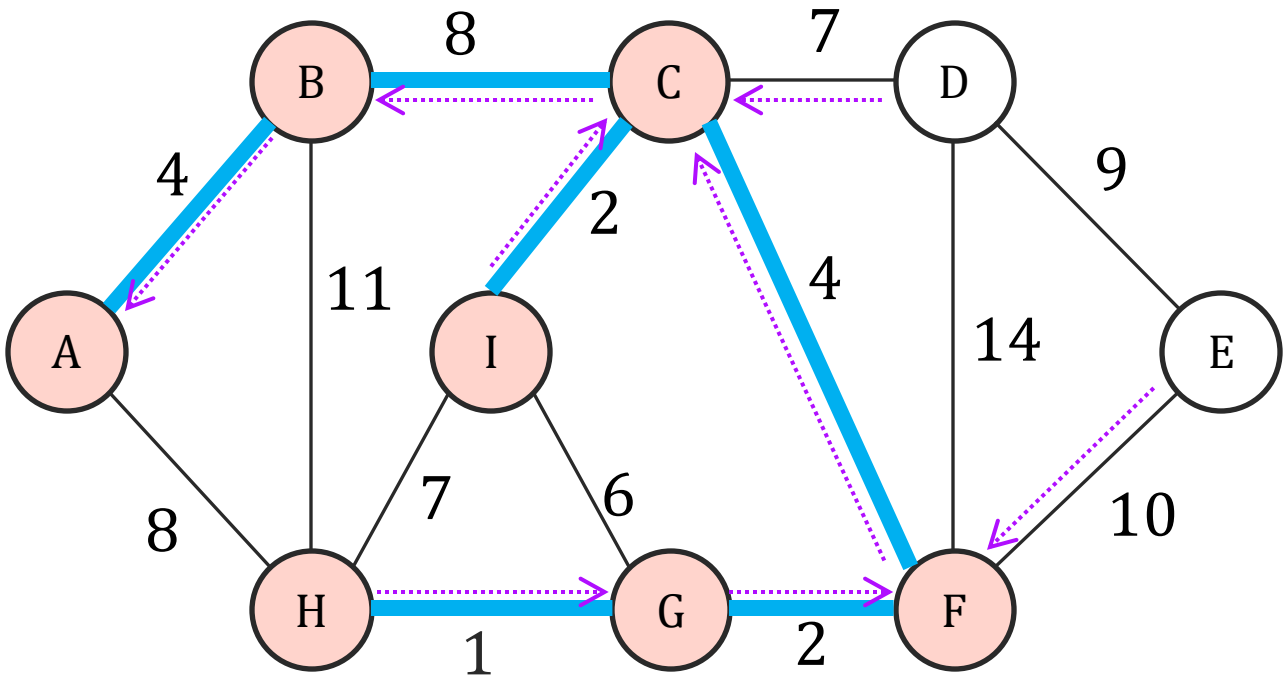


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	$\emptyset$	A	B	C	F	C	F	G	C

```
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 $v \in V$, 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 $z \in Q$.

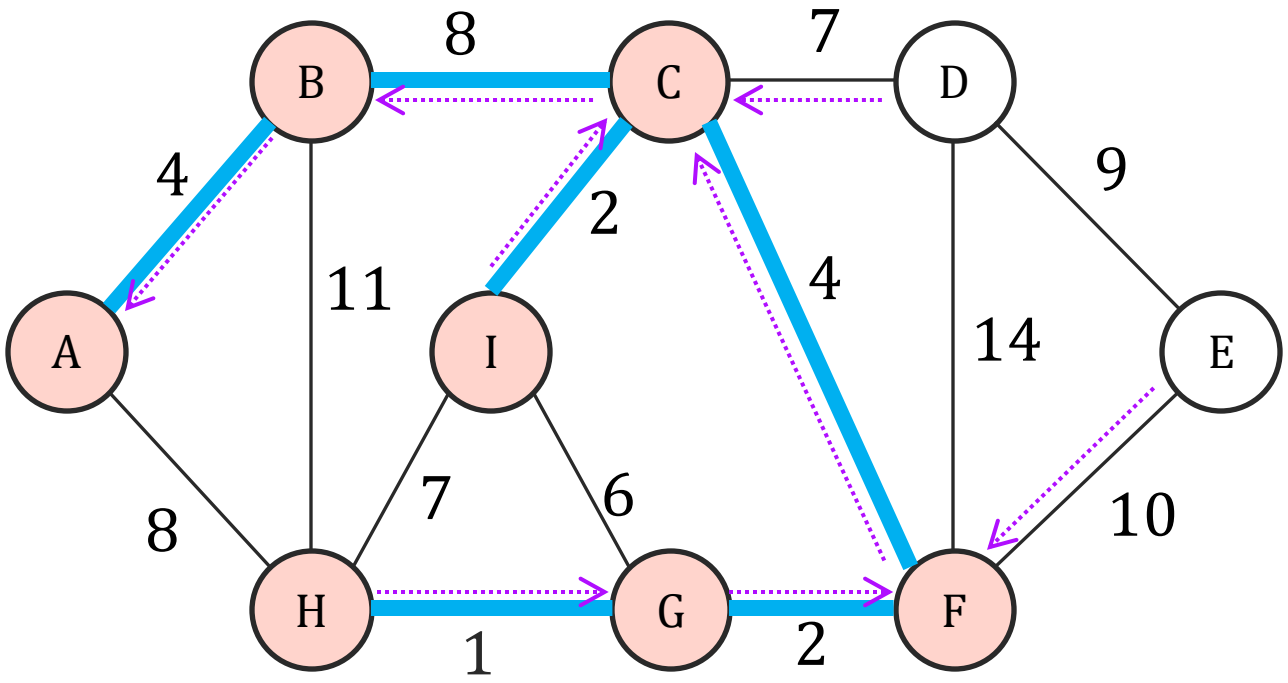
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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



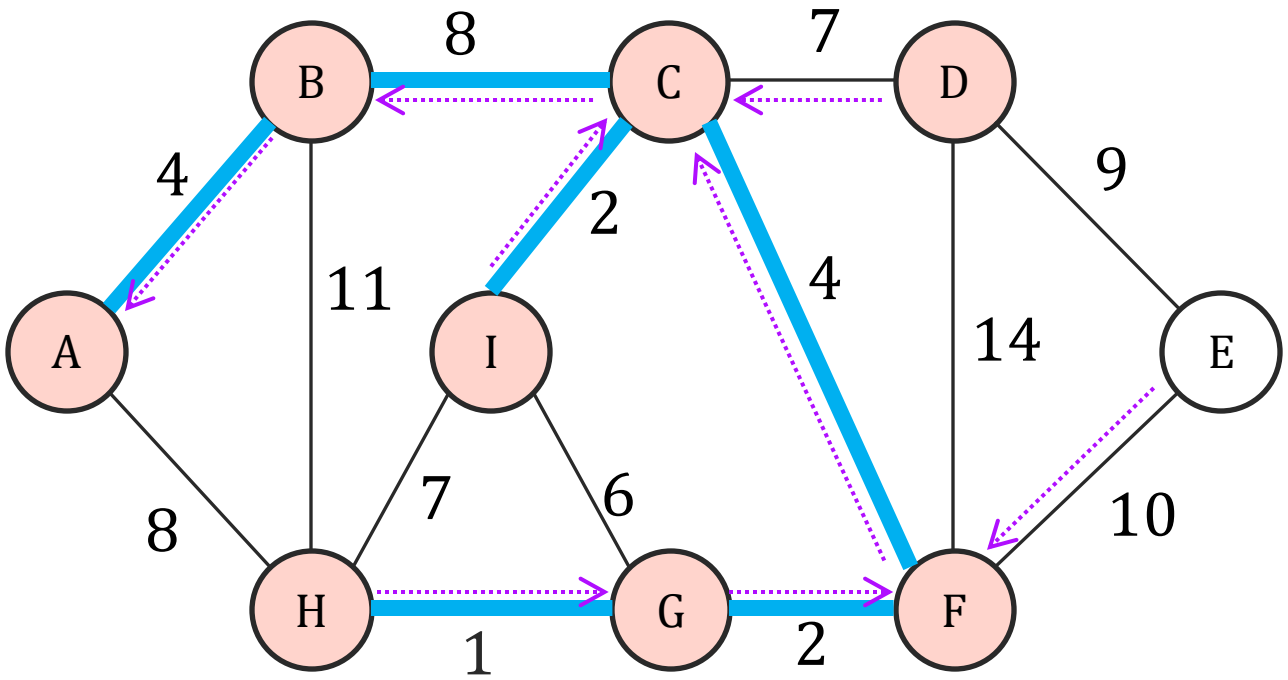
Nothing to update

	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 $v \in V$, 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 $z \in Q$.

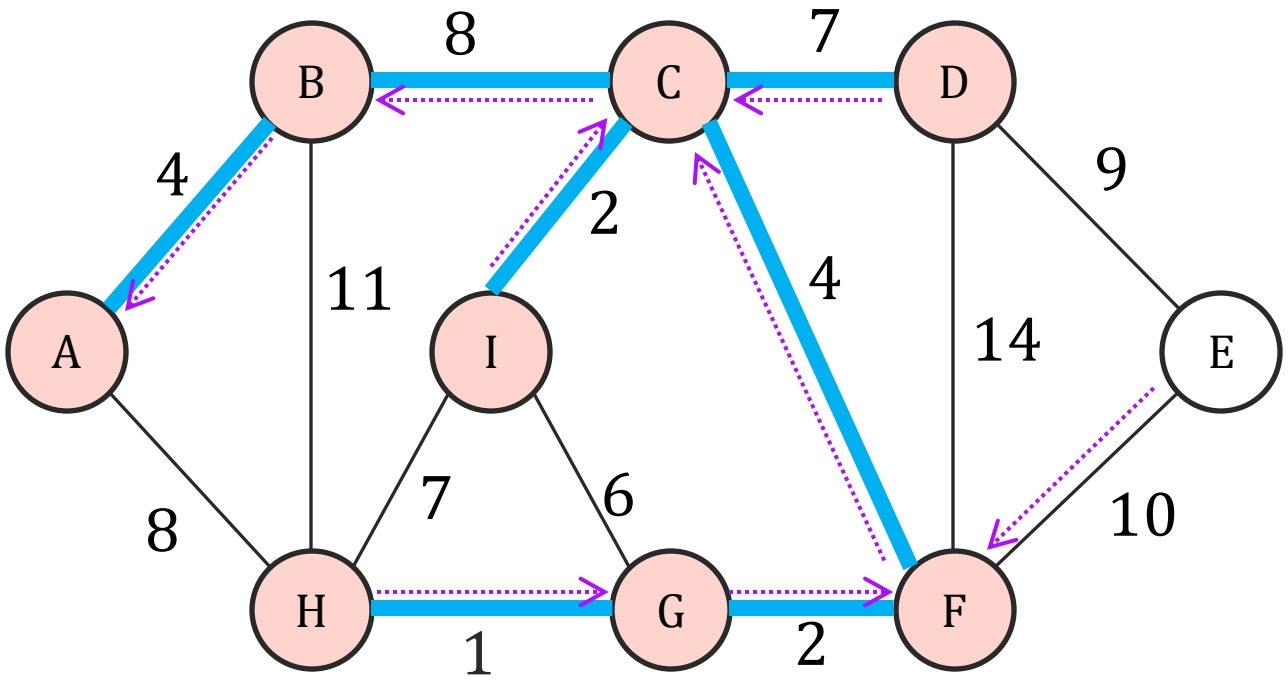
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

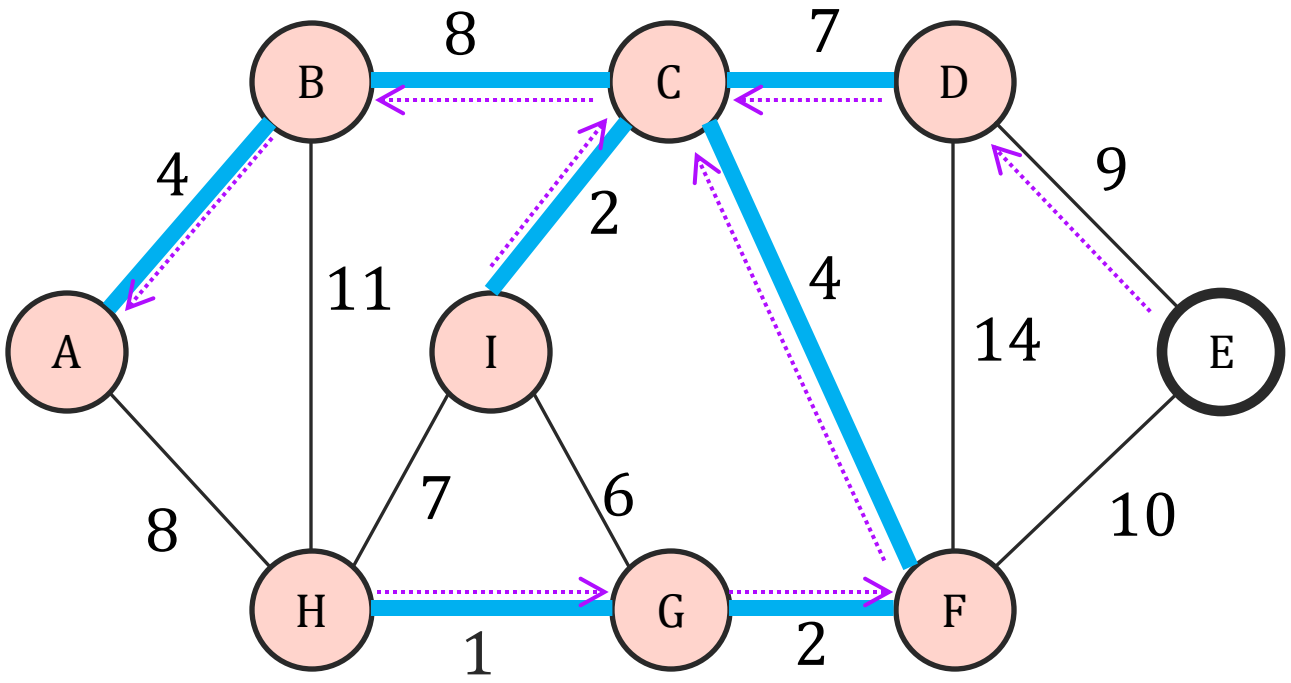


	A	B	C	D	E	F	G	H	I
$dist$	0	4	8	7	10	4	2	1	2
$prev$	$\emptyset$	A	B	C	F	C	F	G	C

```
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
$prev$	∅	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 $z \in Q$.

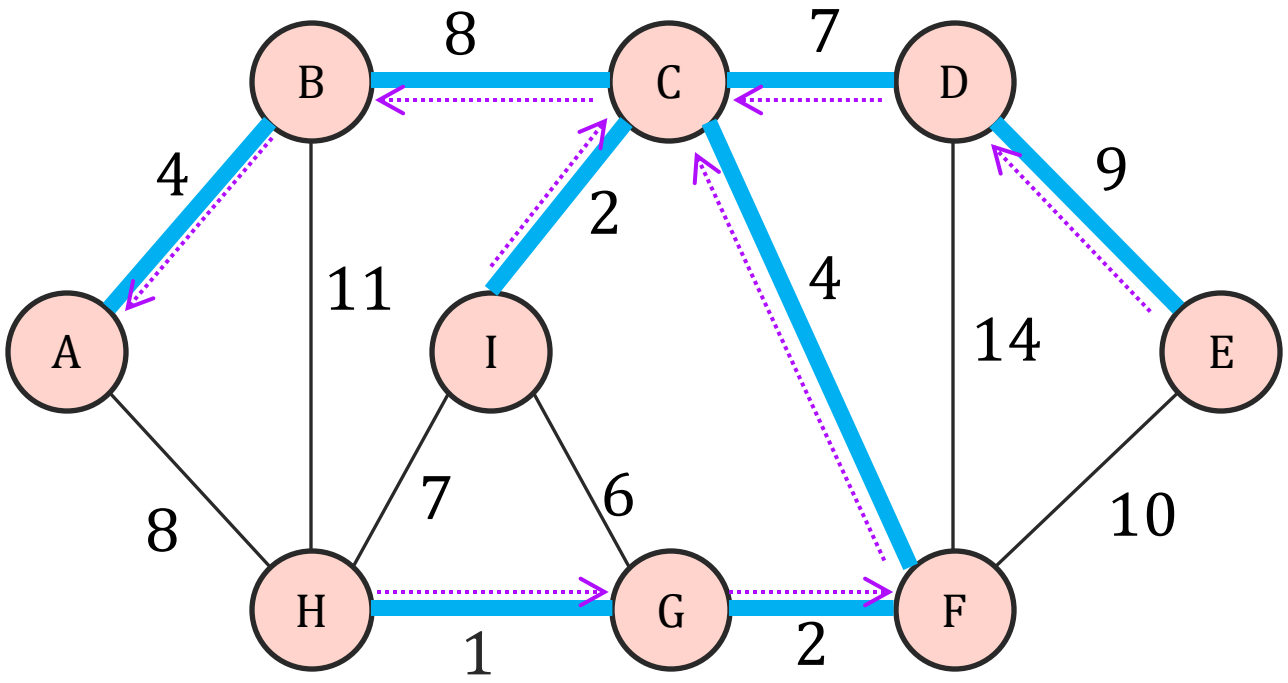
Q.decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

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!

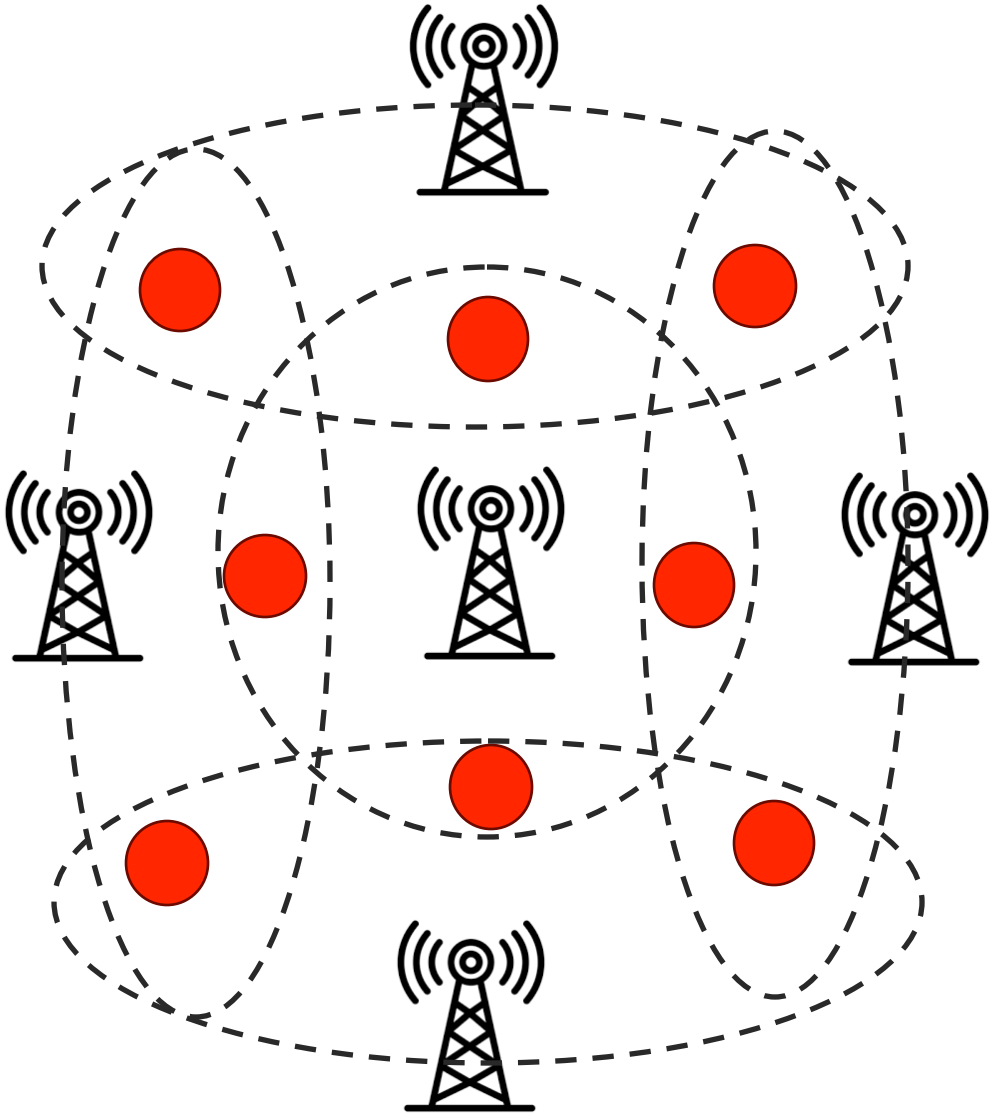
Covering

Imagine, we want to build cell towers so that we provide signal coverage to all houses in a city.

Each **possible location for a cell tower** will cover some homes.

What's the smallest number of cell towers I have to install to cover the city?

Where should these be installed?

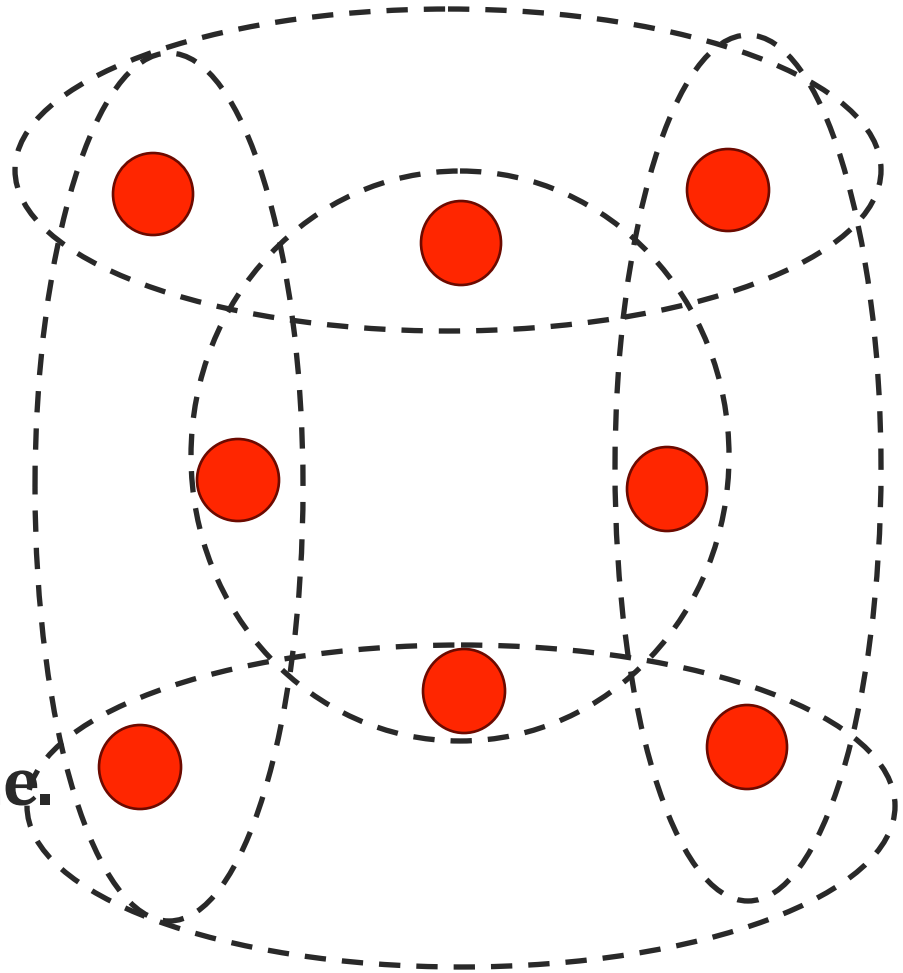


The Set Cover Problem

Input:

→Universe of n elements $U = \{1, \dots, n\}$, and

→Subsets $S_1, S_2, \dots, S_m \subseteq U$, s.t., $\bigcup_{i=1}^{m} S_i = U$



Output:

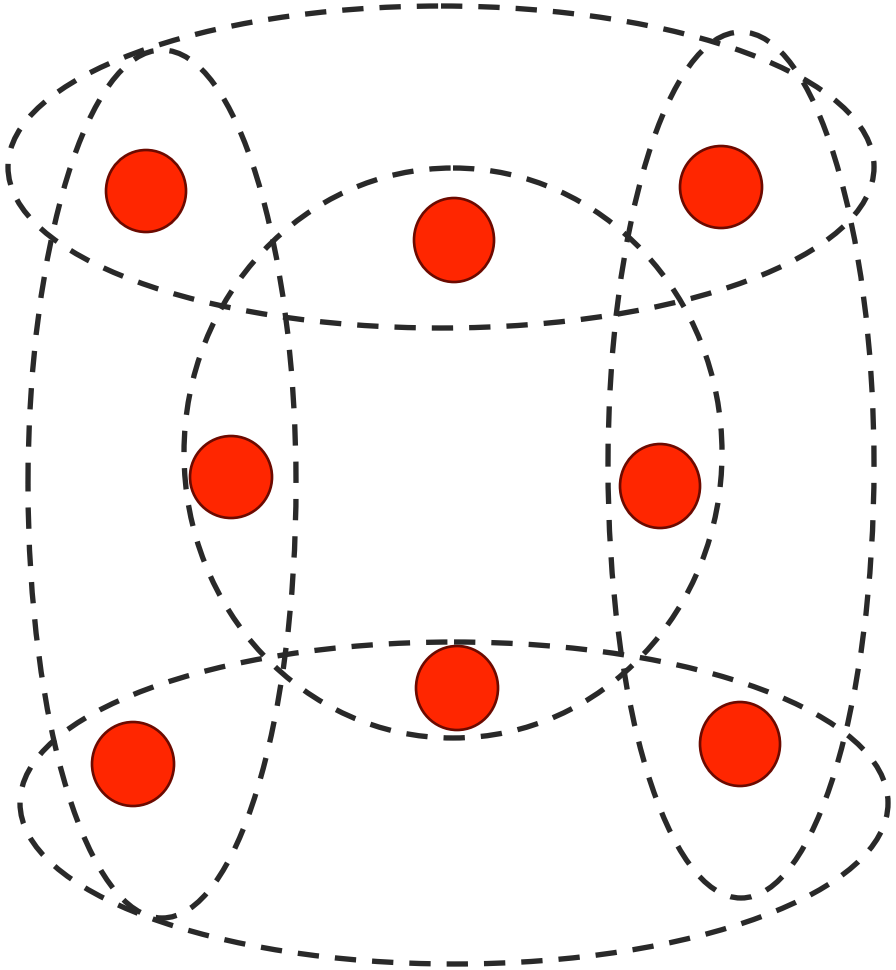
A collection of subsets covering U of **minimal size.**

i.e., $J \subseteq \{1,2, \dots, m\}$ s.t., $\bigcup_{i \in J} S_i = U$

Greedy Algorithm

Discuss

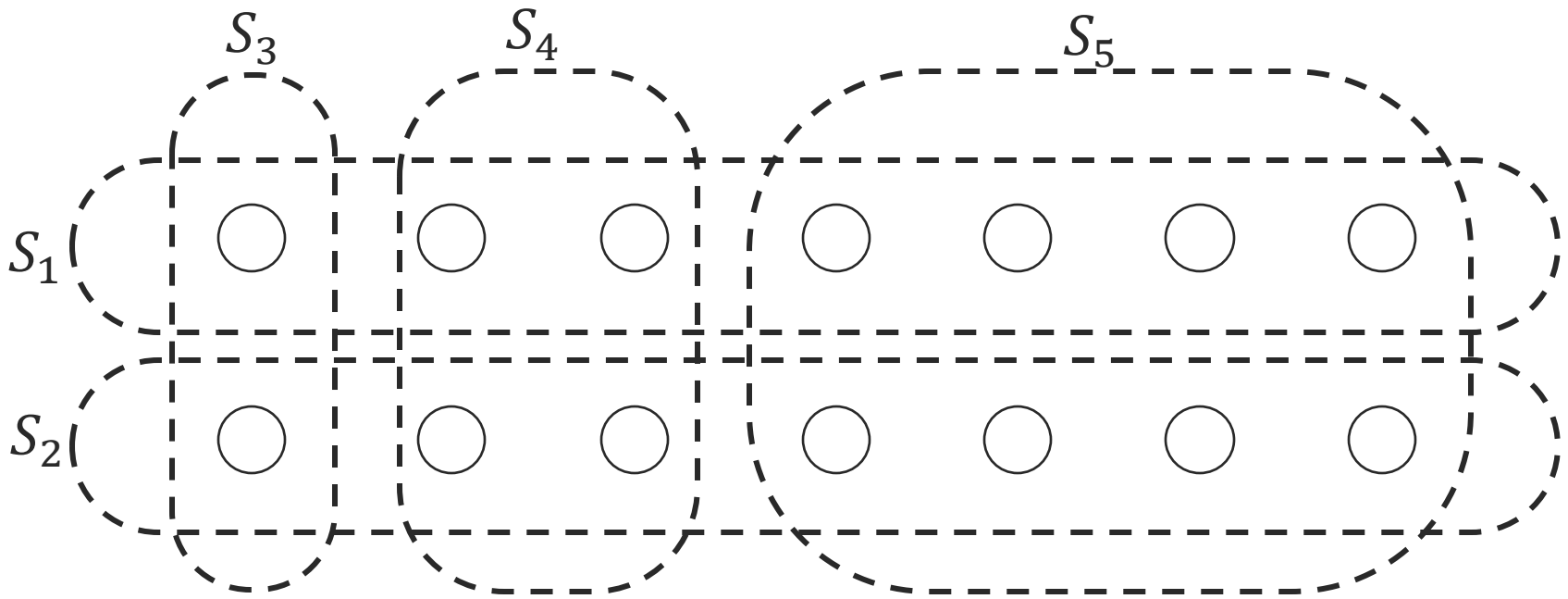
What is a good greedy algorithm?



Greedy Algorithm for Set Cover

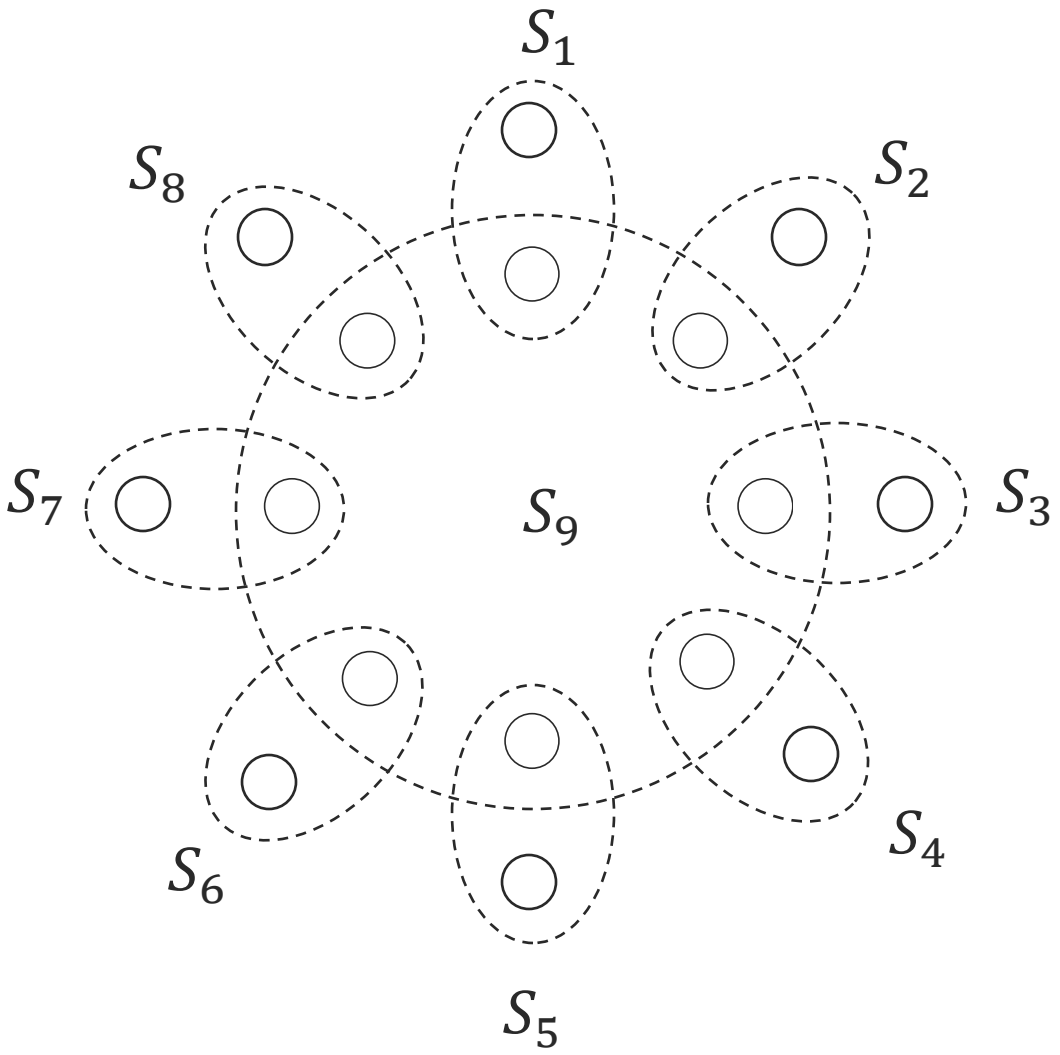
A suggested greedy algorithm:

Repeat until all elements of U are covered: Pick the set with the largest number of uncovered elements.



Greedy is not optimal for Set Cover

One other example where this greedy algorithm is not optimal



Wrap up

Almost done with being greedy!

→ Just a little left: Greedy is actually reasonably good for set cover.

→ We mastered proof by induction!

→ Scheduling, Minimum Spanning Trees, Horn-Satisfiability, MSTs, Set Cover

Next time

- Set Cover with Greedy
- Dynamic Programming