CS 170
Efficient Algorithms and Intractable Problems

Lecture 11
Minimum Spanning Trees and Set Cover

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 next Tuesday 10/3

→ No class on Tuesday!

→ Midterm 1 Review Sessions: 11 - 2 Saturday, Sunday @ Woz - Soda 411

→ There will be a logistics post on Ed

→ Use Ed, OH, HWP, and Review Session to prepare for the exam

→ Scope: Up to and including Sept 26^{th} lecture.

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. Example: Kruskal's alg.

"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: 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.

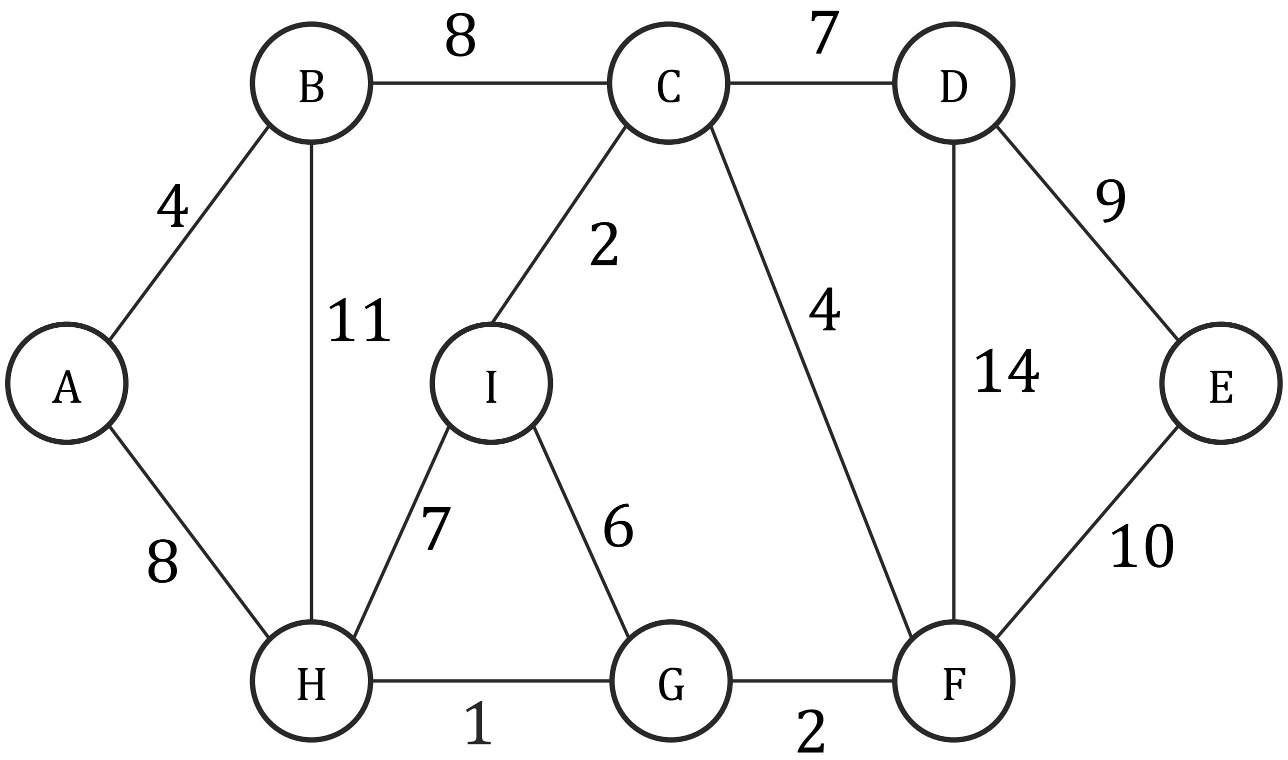
"Cut Property":
If X is a subset of an MST and has no edges from S to $V \setminus S$, then $X \cup \{e\}$ is also a subset of an MST.

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 $v \in V \setminus S$.

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

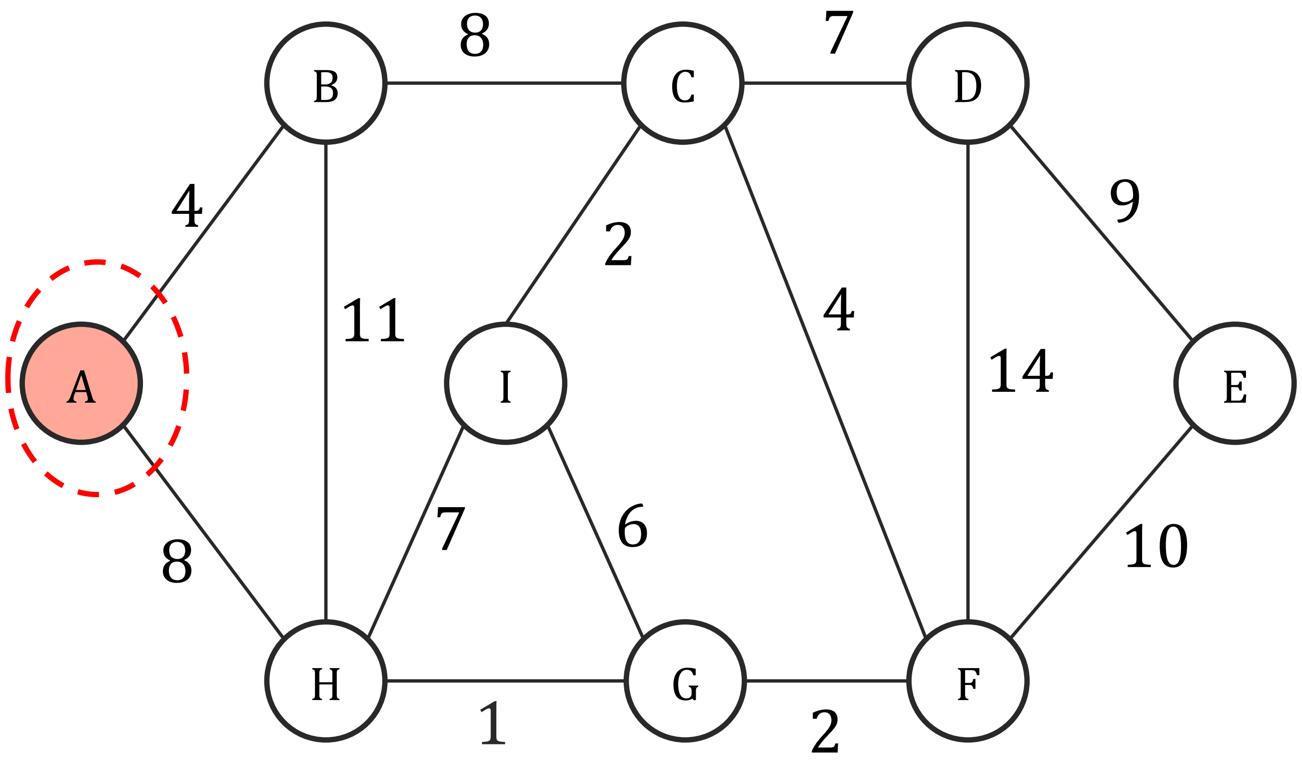
    $S \leftarrow S \cup \{v\}$

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 $|X| < |V| - 1$

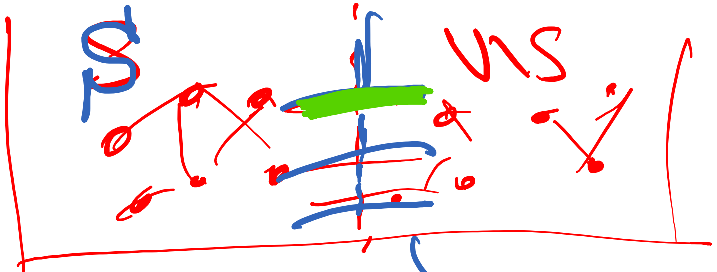
Let $e = (u, v)$ be the lightest edge such that $u \in S$ and $v \in V \setminus S$.

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

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

Return X

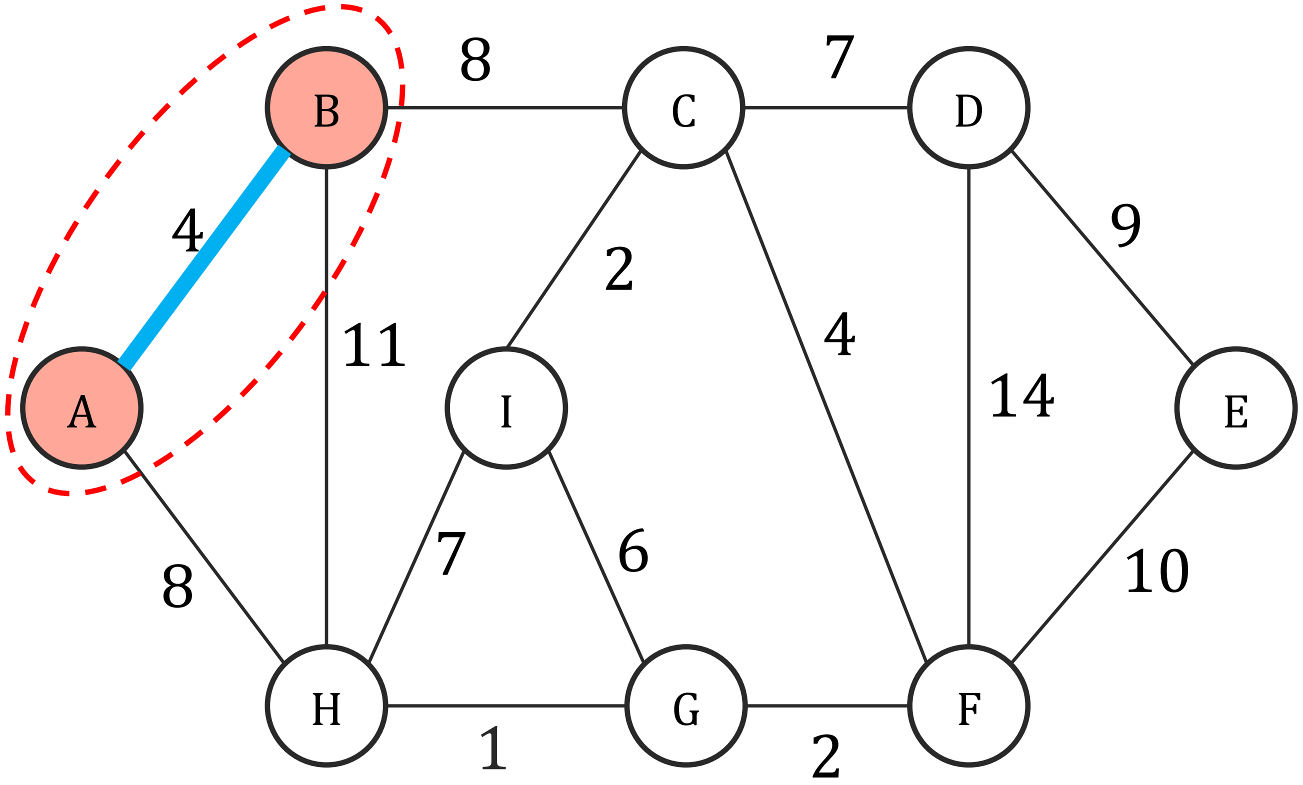
Prim's Algorithm



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

none belong to X

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 $v \in V \setminus S$.

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

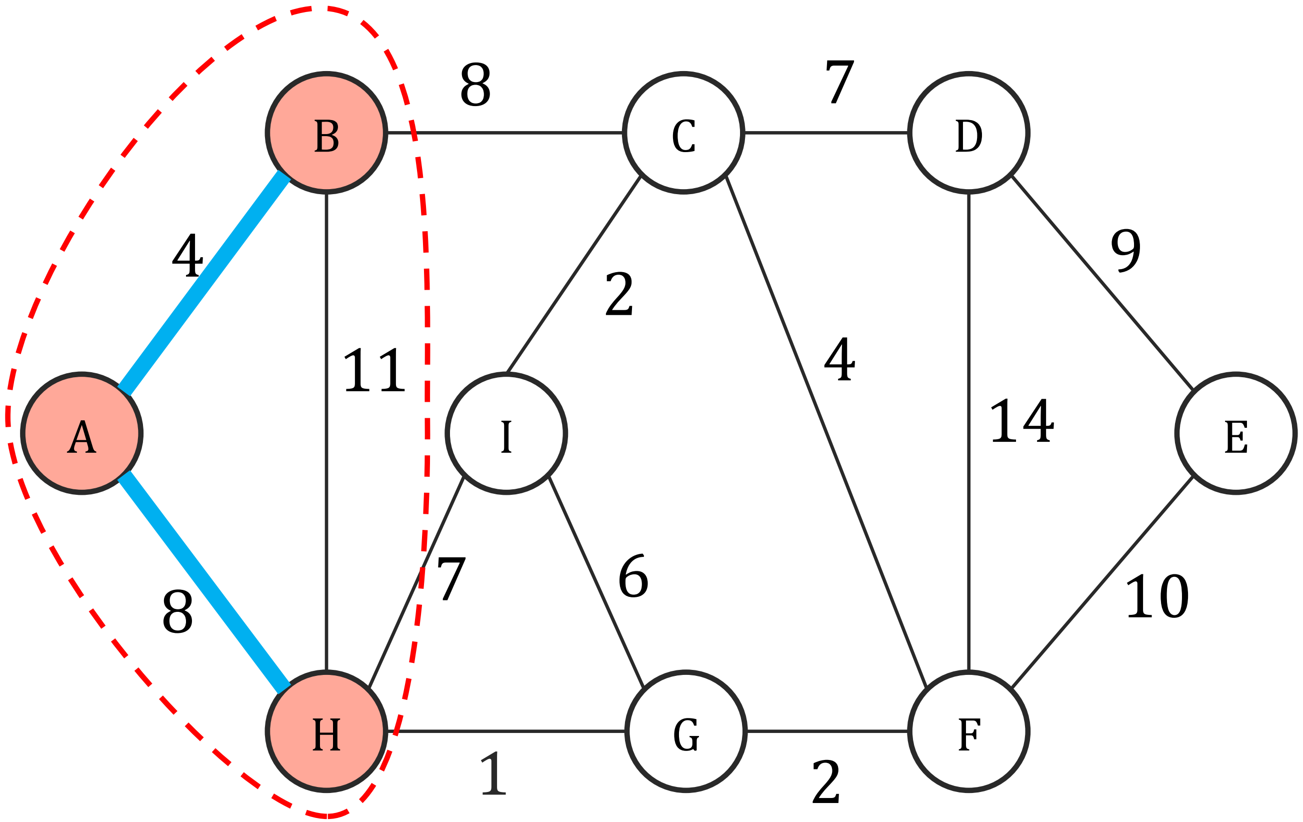
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

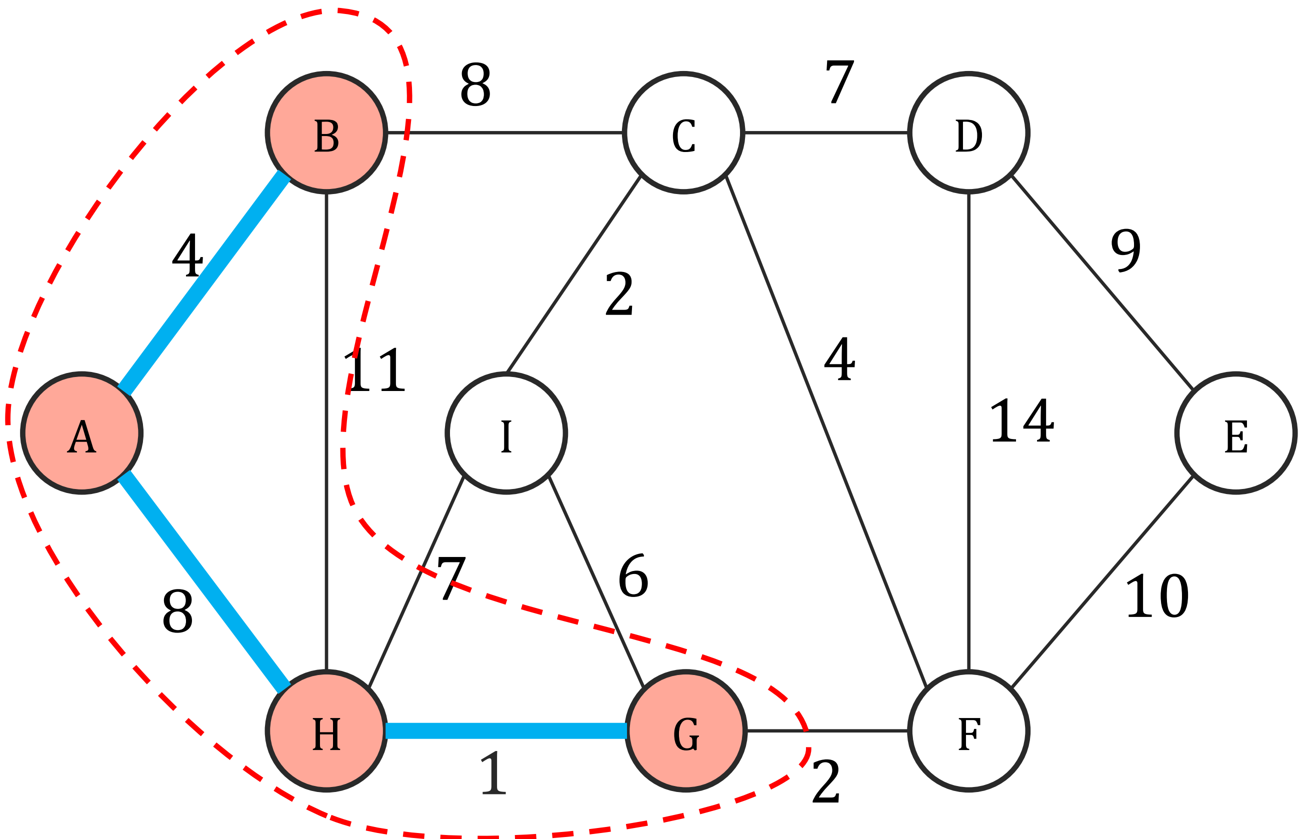
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

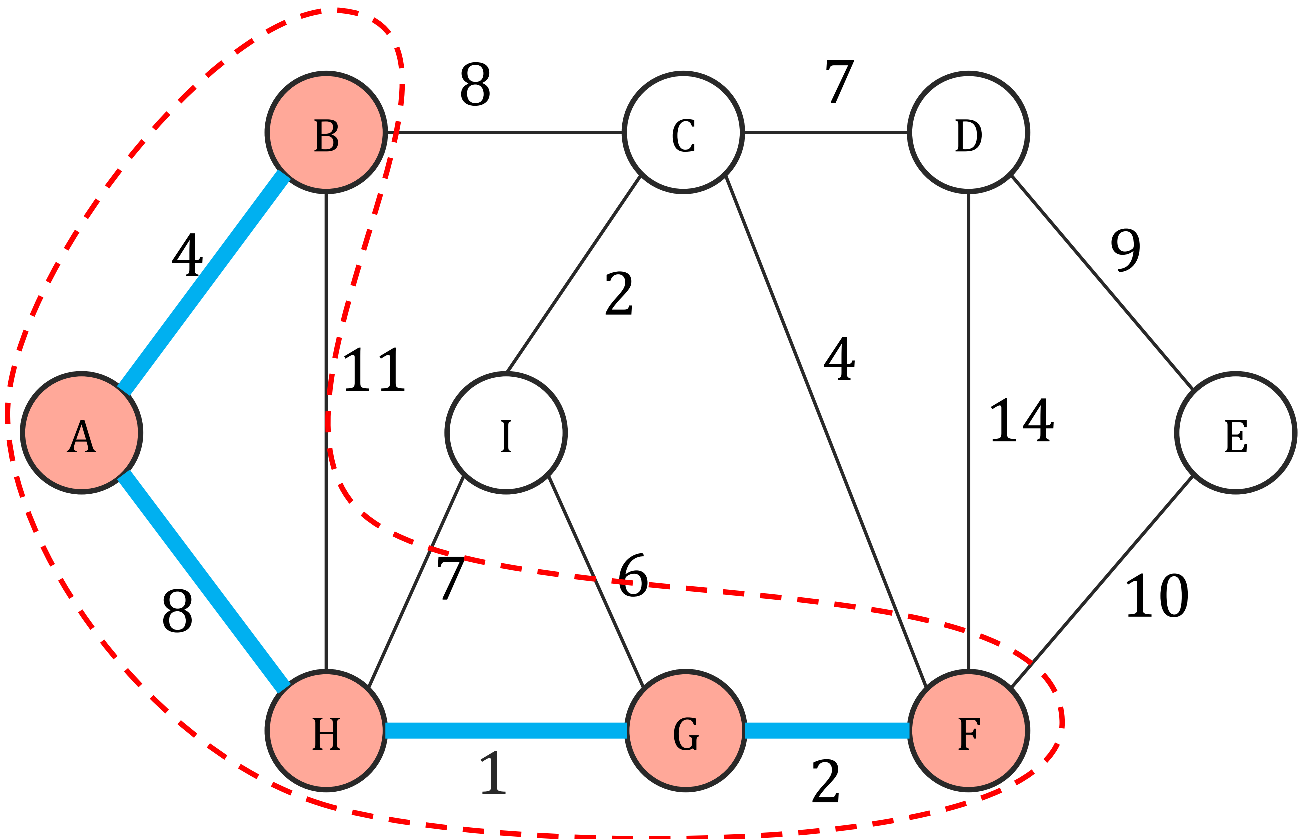
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

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

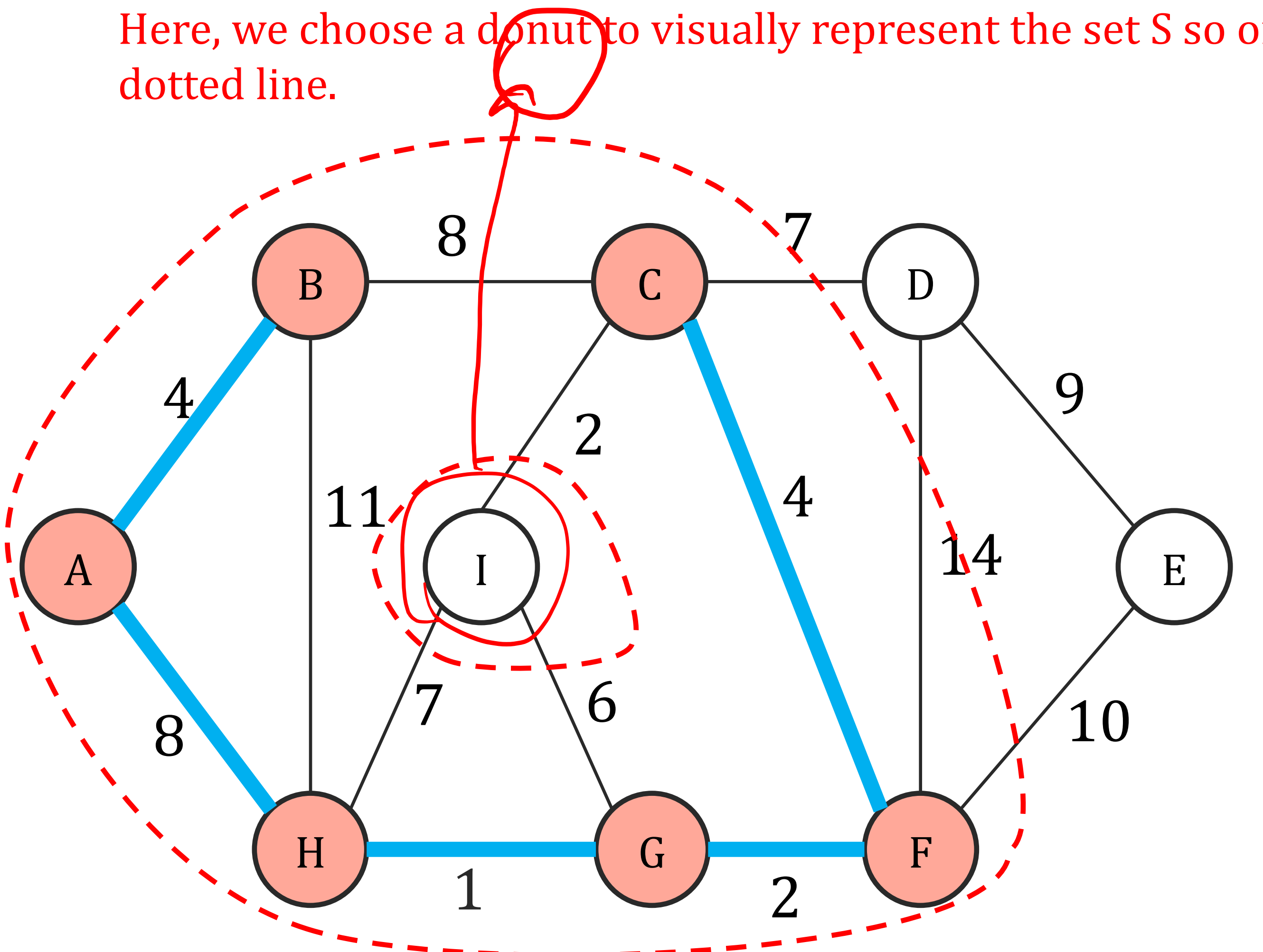
Return X

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 $v \in V \setminus S$.

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

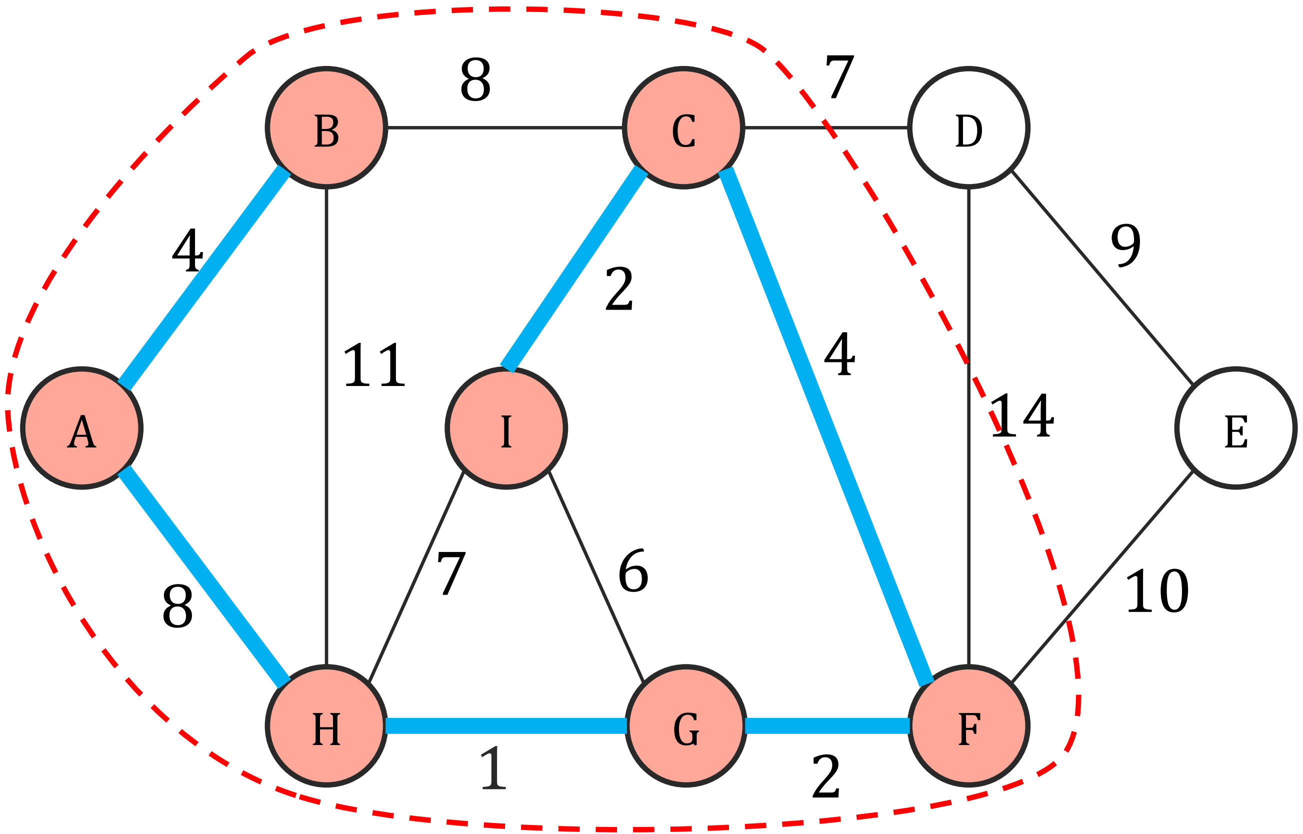
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

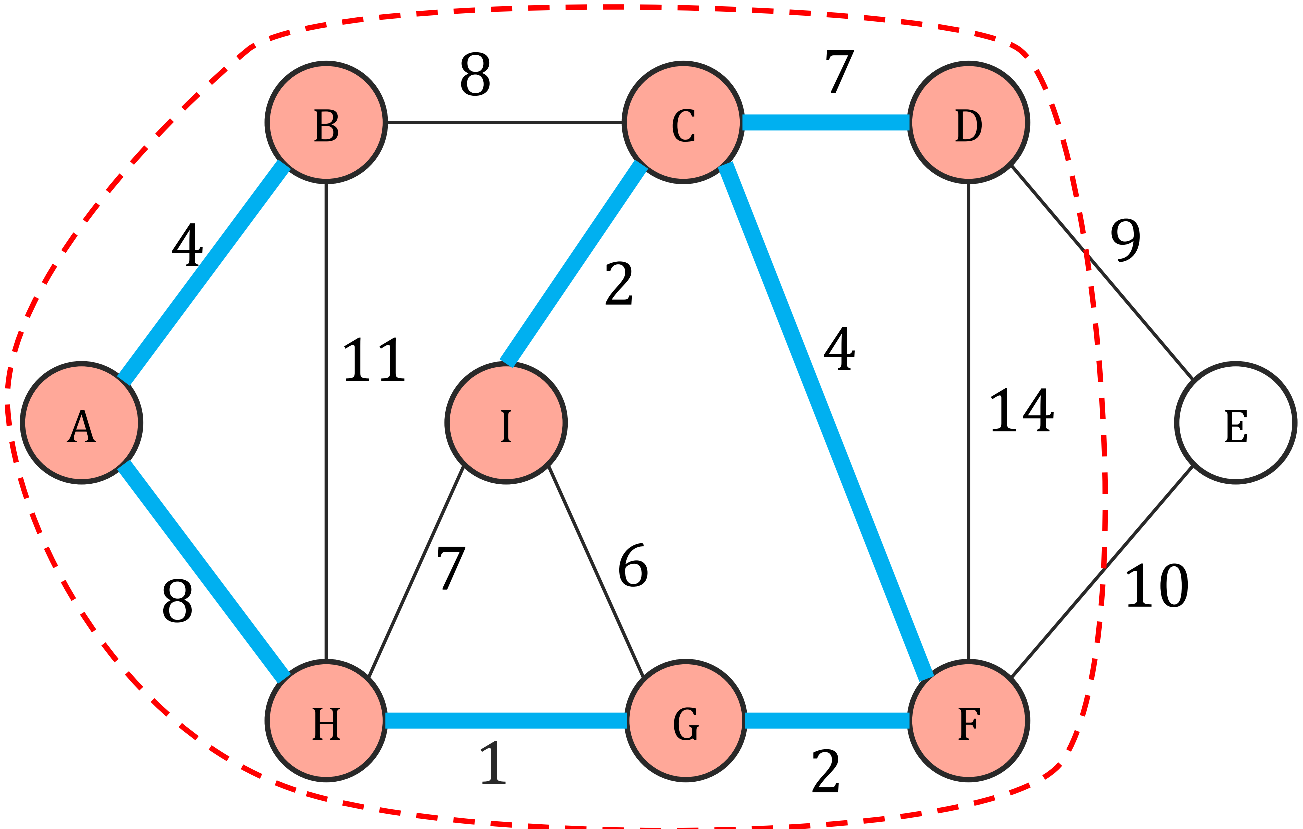
$S \leftarrow S \cup \{v\}$

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 $v \in V \setminus S$.

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

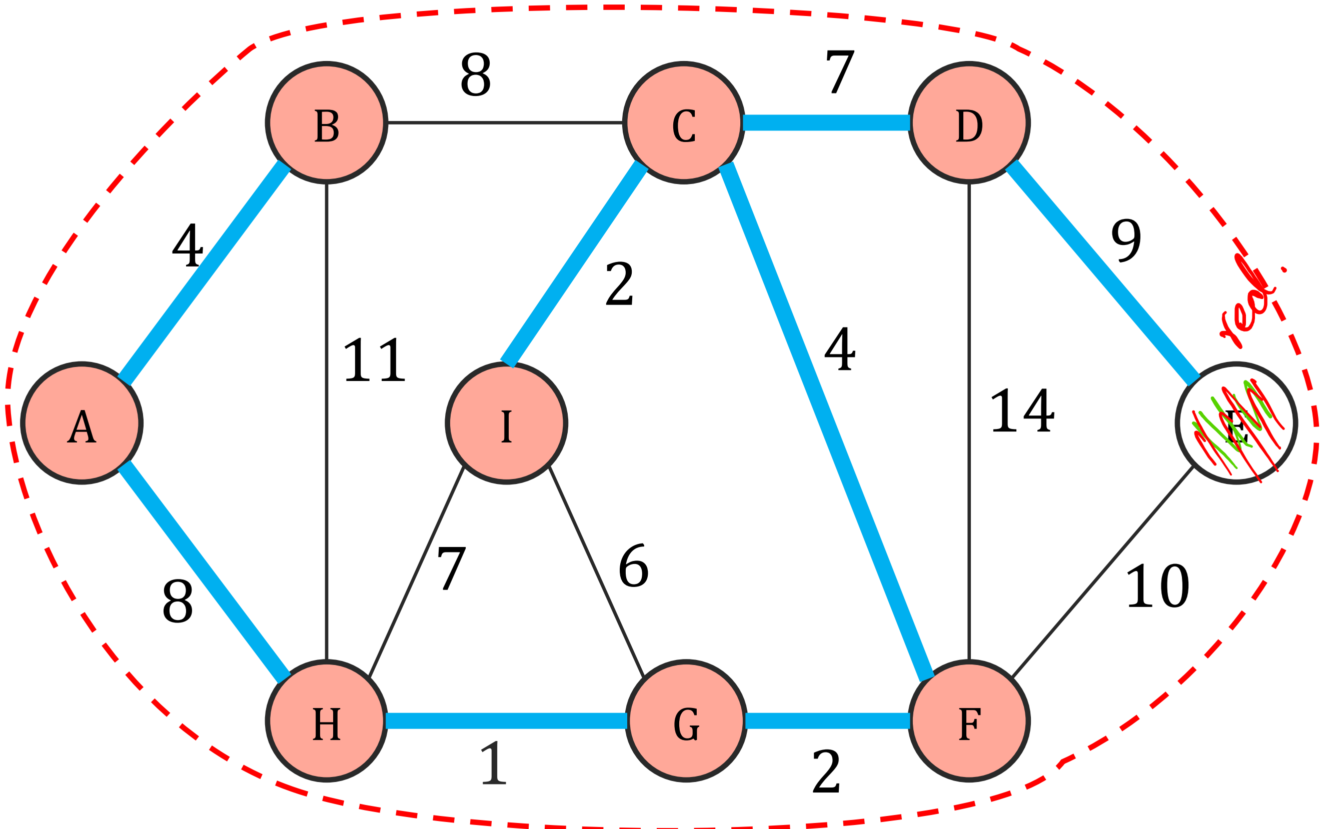
$S \leftarrow S \cup \{v\}$

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.

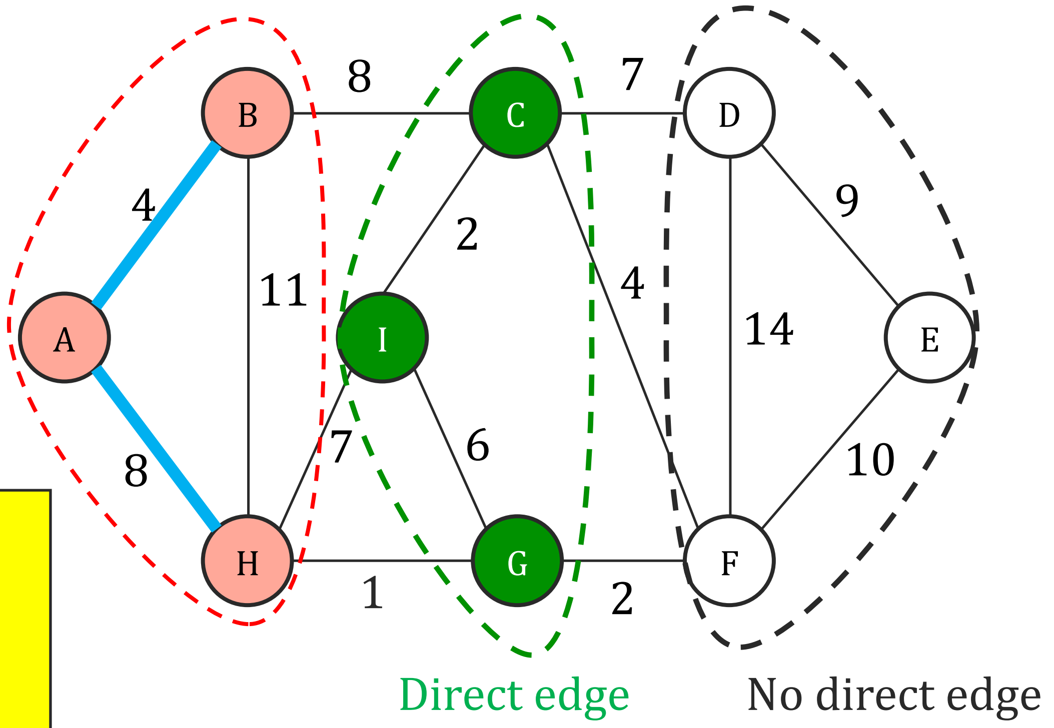
Naively implementing this, take $O(nm)$.

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

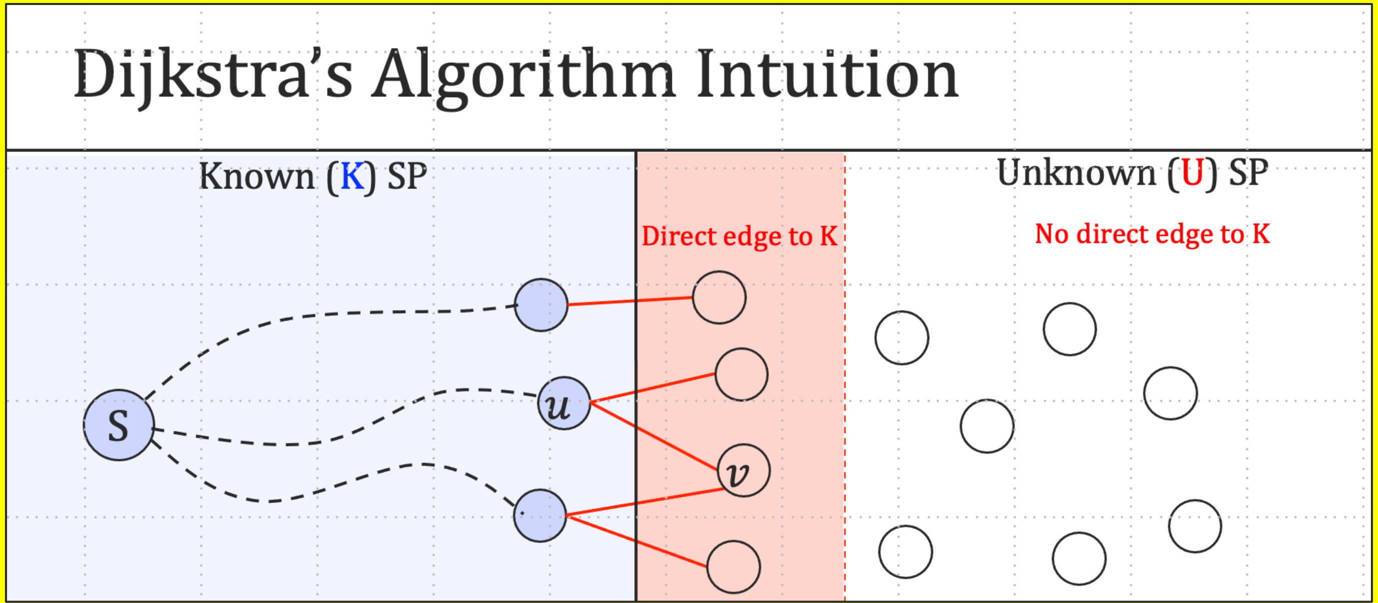
How do we actually implement Prim's Algorithm?

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.



Once before, we had the same dilemma, in Lec. 8 ….



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

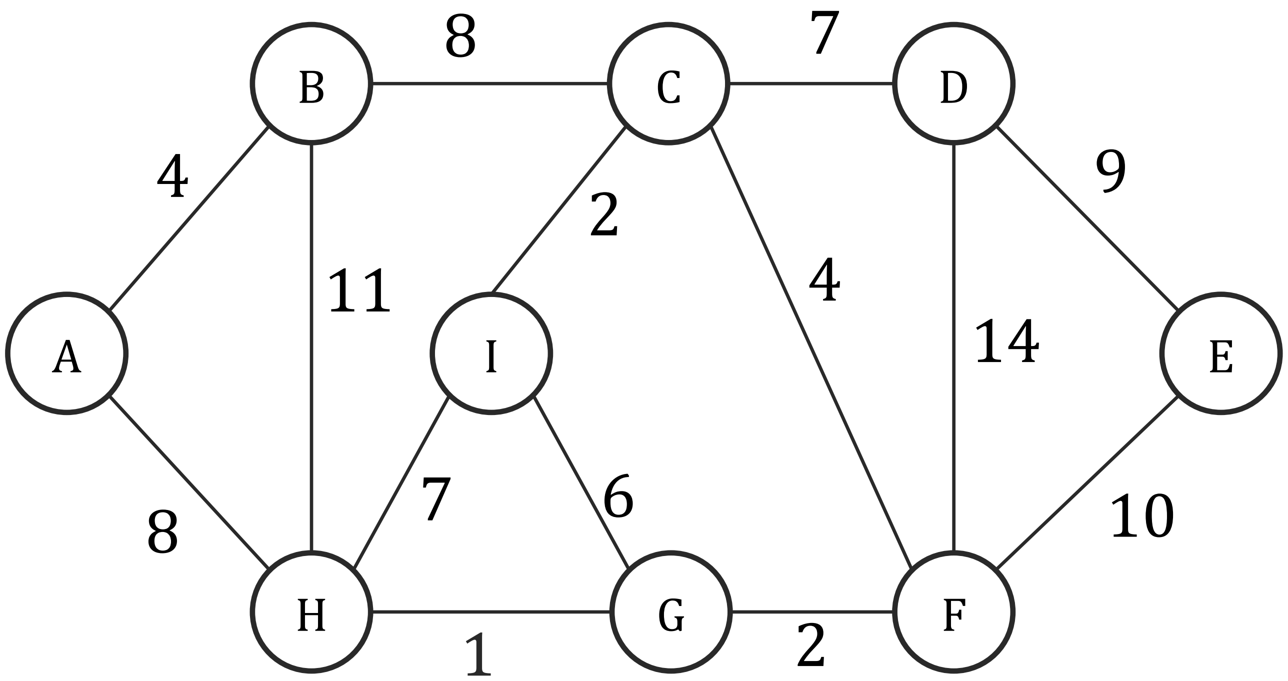
  update $prev[v] \leftarrow u$

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.

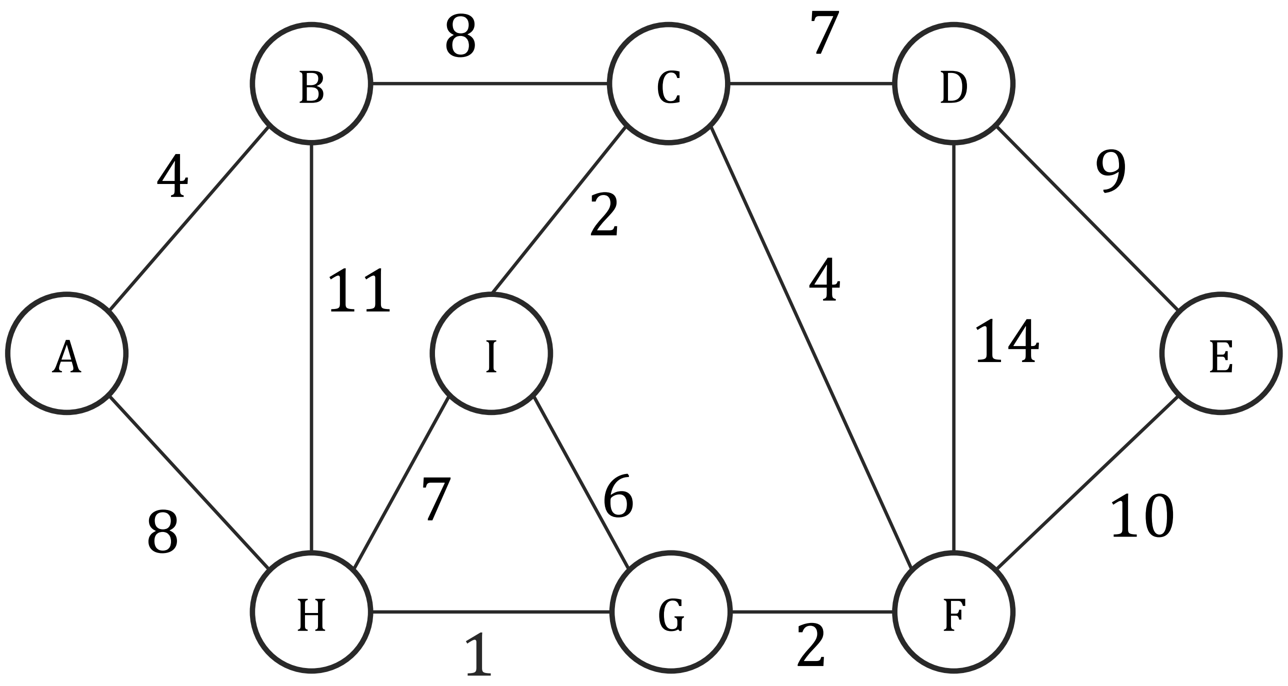
Prim's Algorithm: Efficient Implementation



	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 $v \in V$, 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



	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 $z \in Q$.

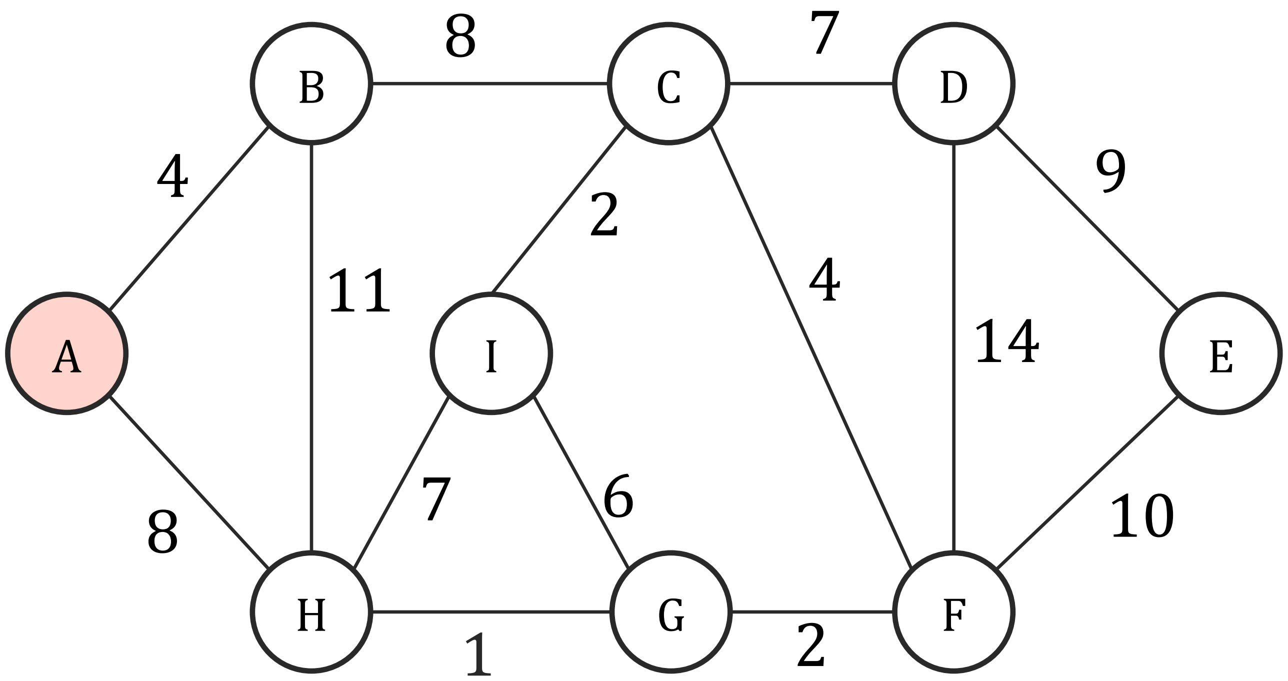
        Q. decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  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	~~$\emptyset$~~	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

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 $z \in Q$.

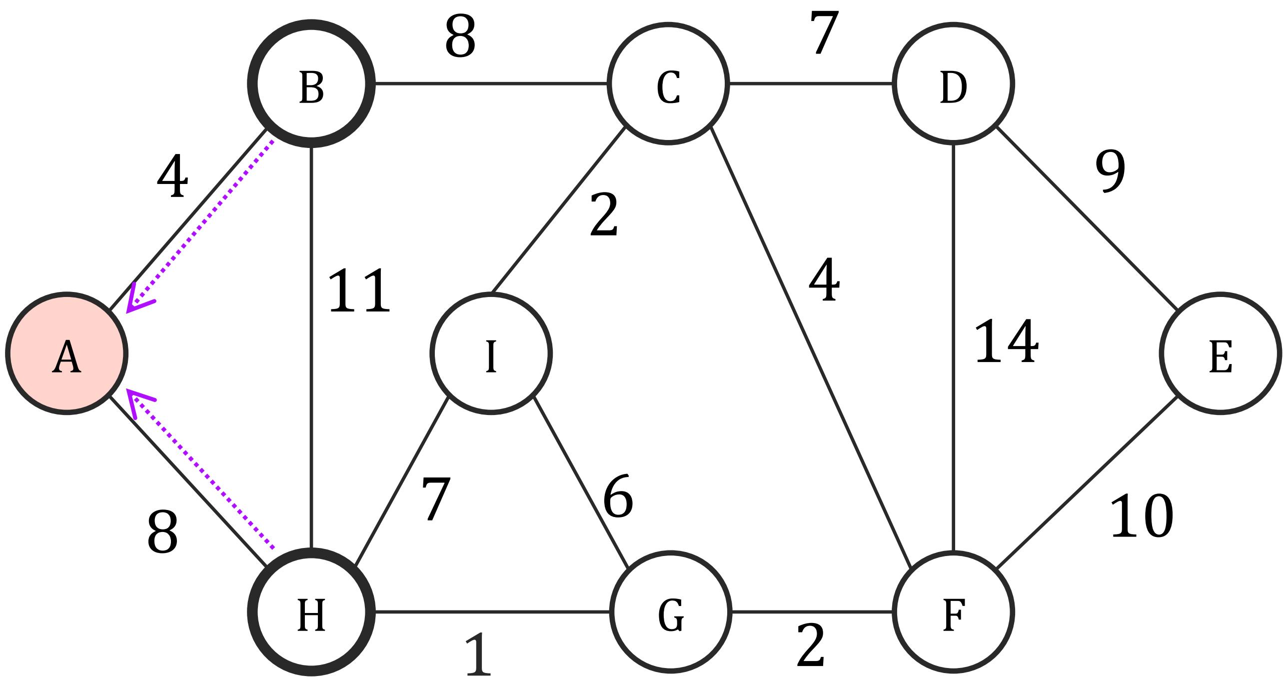
        Q. decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  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 $v \in V$, $Q.\mathrm{insert}(v, dist[v])$

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 $z \in Q$.

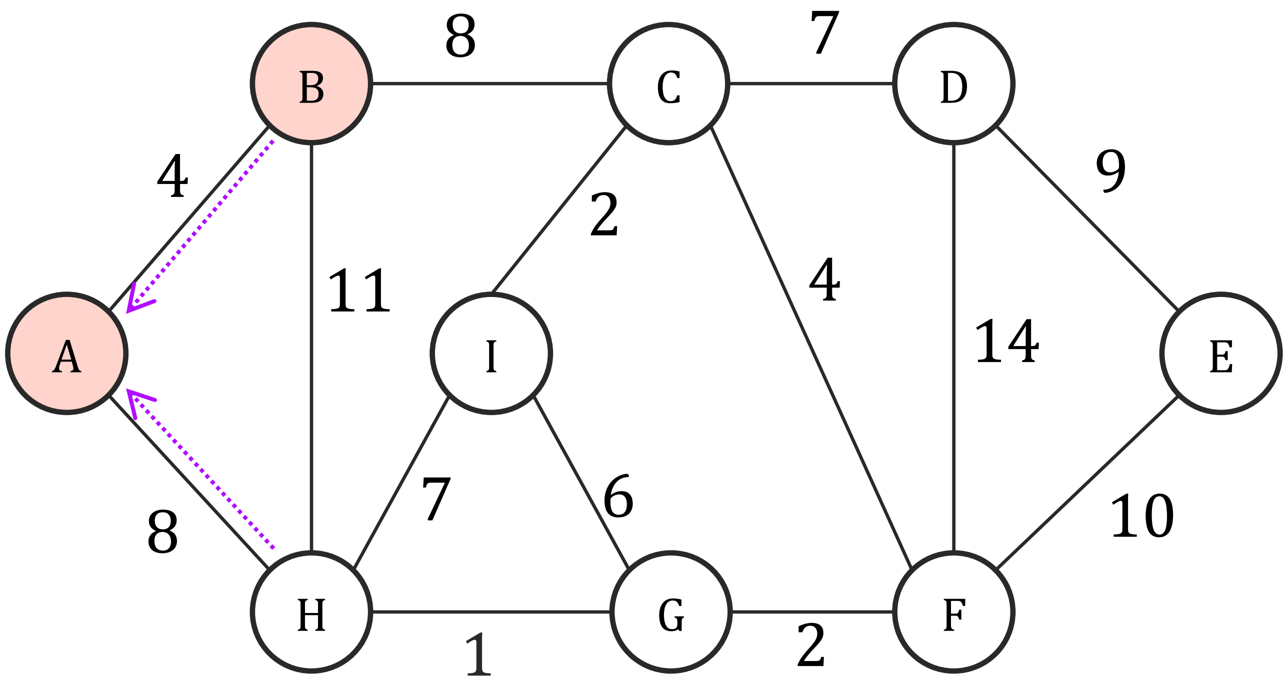
$Q.\mathrm{decreaseKey}(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

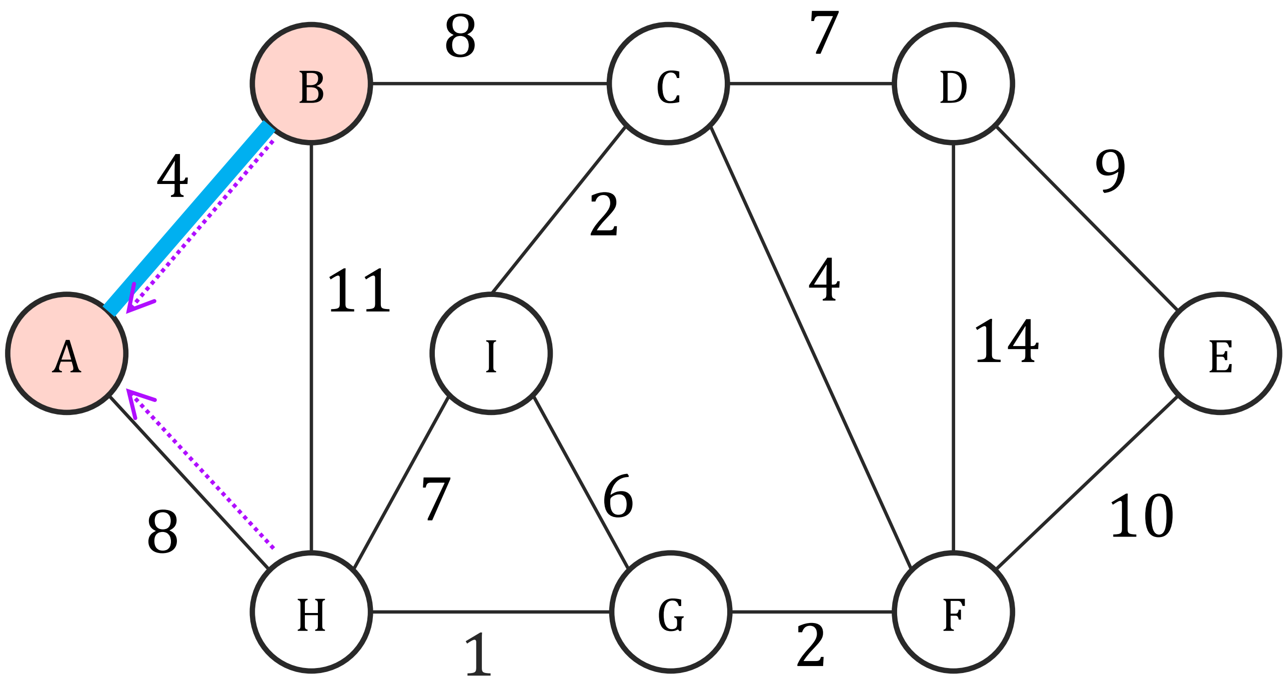


	A	B	C	D	E	F	G	H	I
dist	~~0~~	~~4~~	∞	∞	∞	∞	∞	8	∞
prev	~~∅~~	~~A~~	∅	∅	∅	∅	∅	A	∅

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 $v \in V$, $Q.\mathrm{insert}(v, dist[v])$
  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 $z \in Q$.
        $Q.\mathrm{decreaseKey}(z, w_{(v,z)})$
        $prev[z] \leftarrow v$
  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 $z \in Q$.

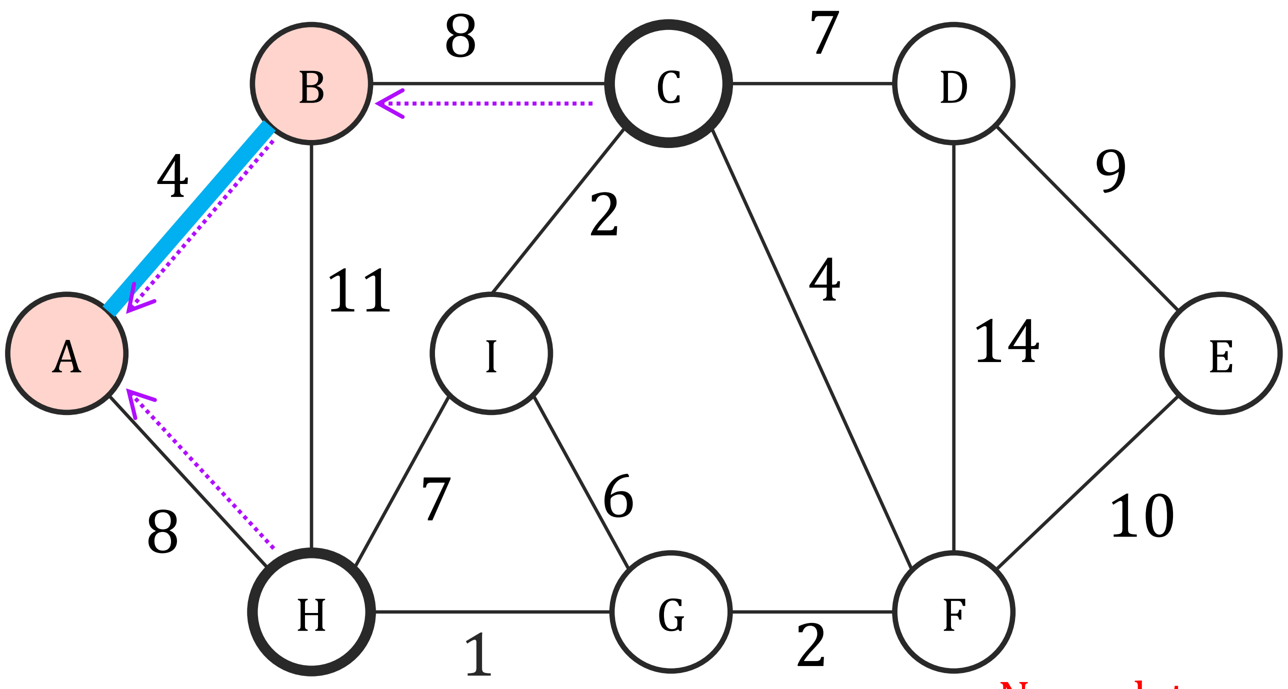
        Q.decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  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**	∞	∞	∞	∞	8	∞
prev	~~$\emptyset$~~	~~A~~	$\boldsymbol{B}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	A	$\emptyset$

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 $v \in V$, $Q.\text{insert}(v, dist[v])$

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 $z \in Q$.

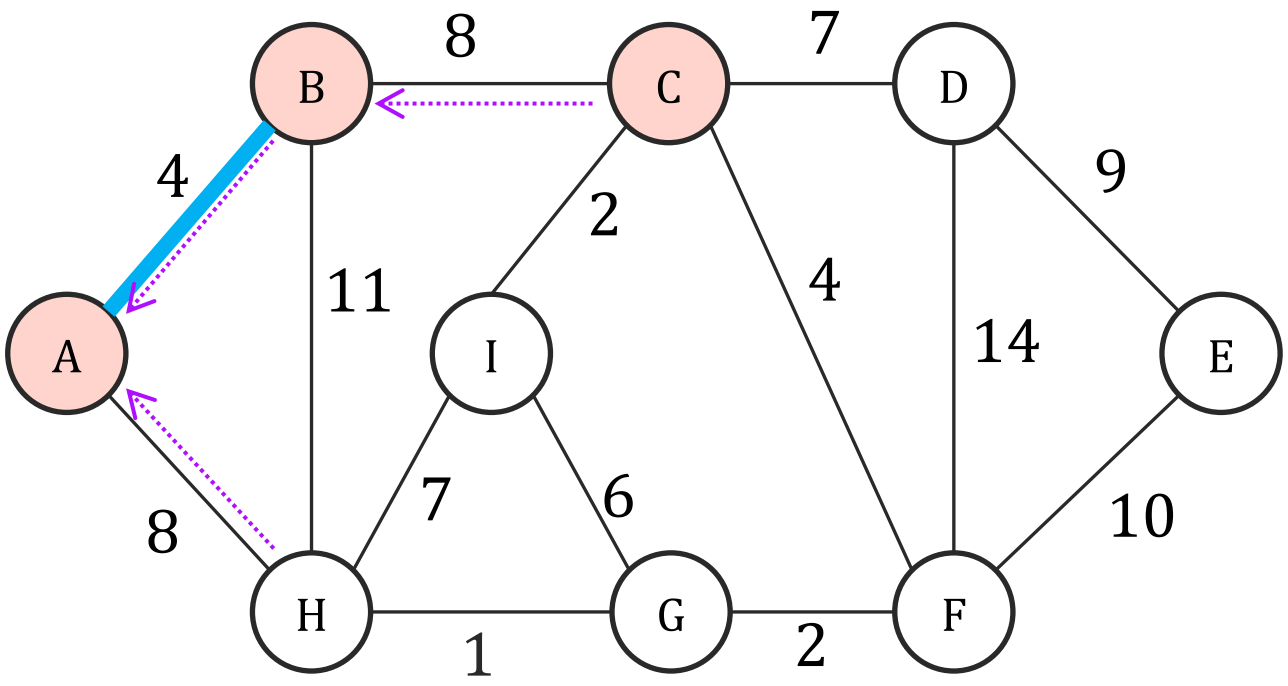
$Q.\text{decreaseKey}(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

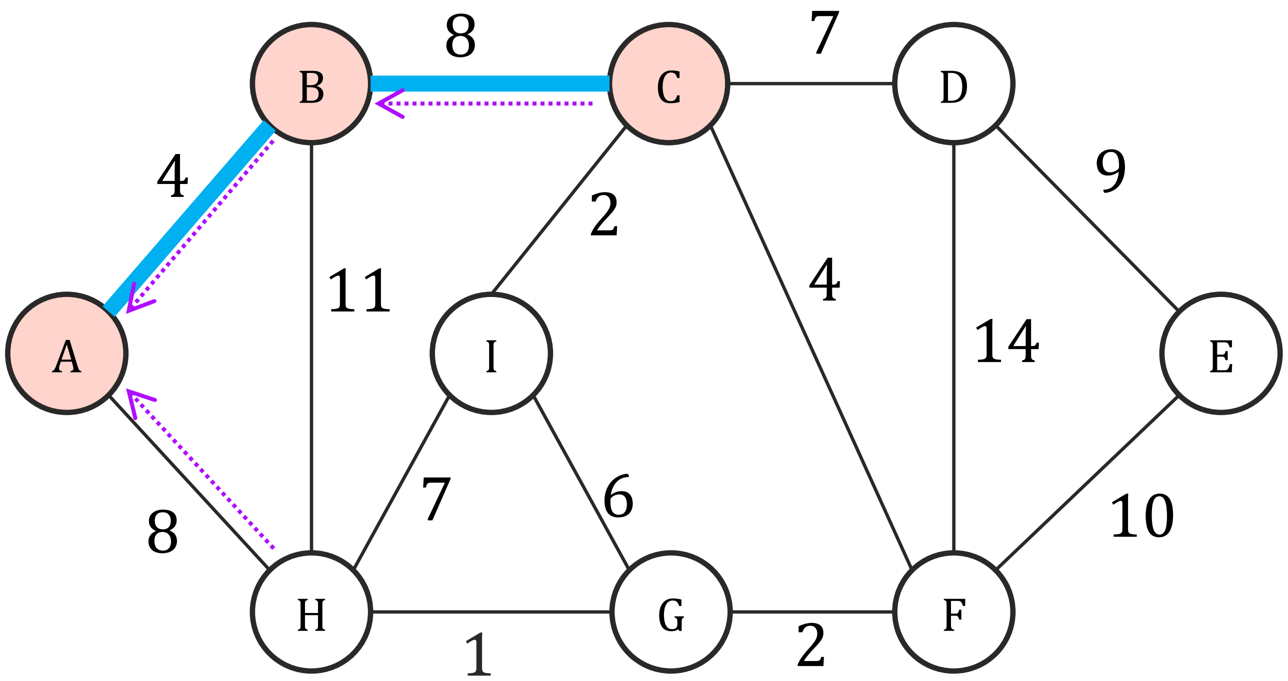


	A	B	C	D	E	F	G	H	I
dist	0	4	8	∞	∞	∞	∞	8	∞
prev	∅	A	B	∅	∅	∅	∅	A	∅

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 $v \in V$, $Q.\text{insert}(v, dist[v])$
  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 $z \in Q$.
        $Q.\text{decreaseKey}(z, w_{(v,z)})$
        $prev[z] \leftarrow v$
  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 $v \in V$, 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 $z \in Q$.

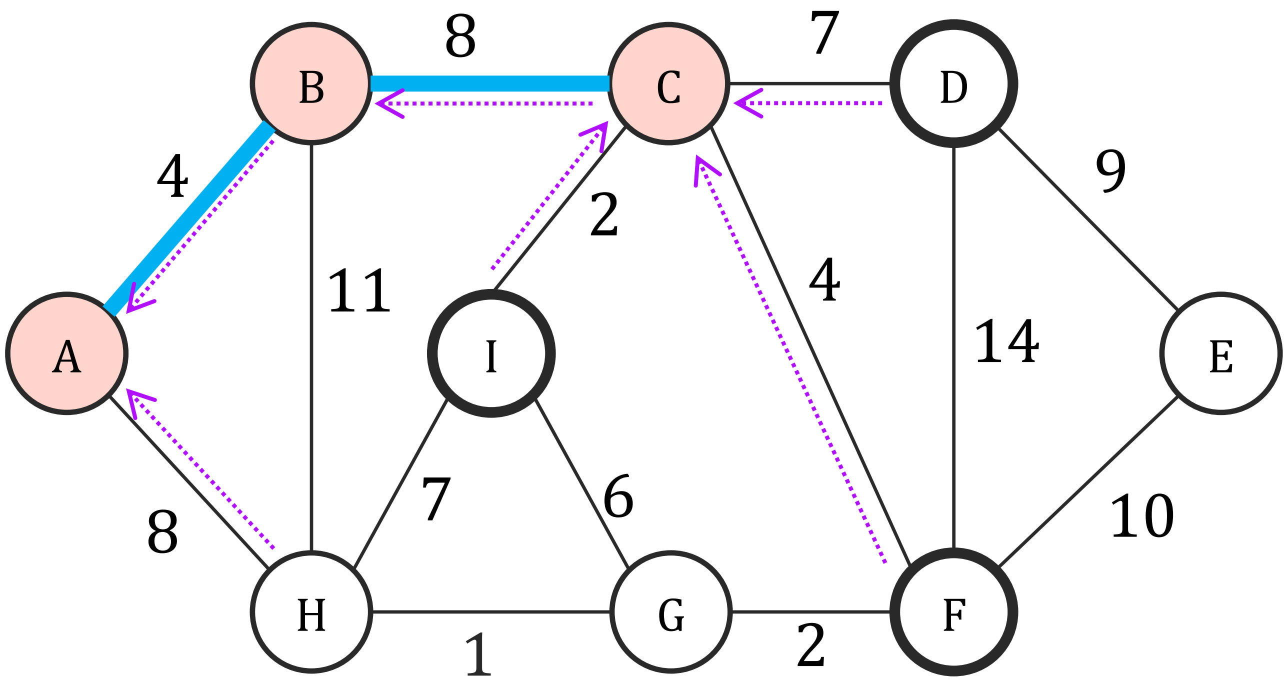
        Q. decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  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	∅	*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 $z \in Q$.

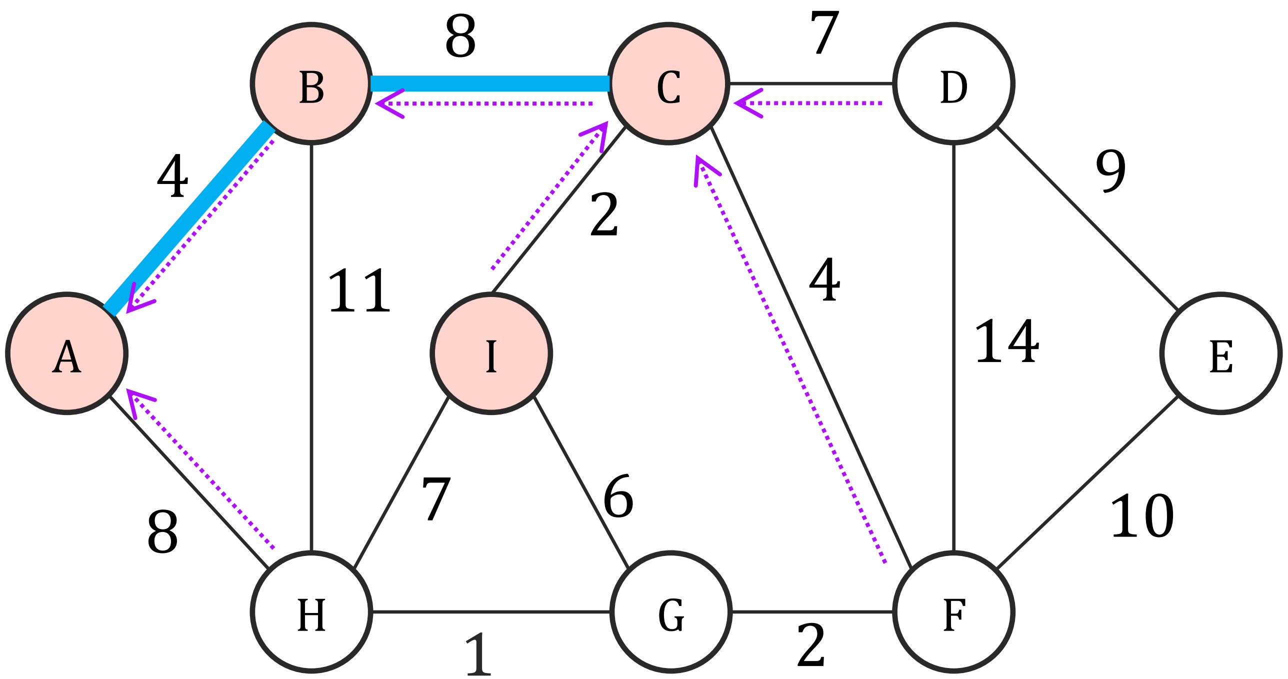
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

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 $v \in V$, $Q.\text{insert}(v, dist[v])$

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 $z \in Q$.

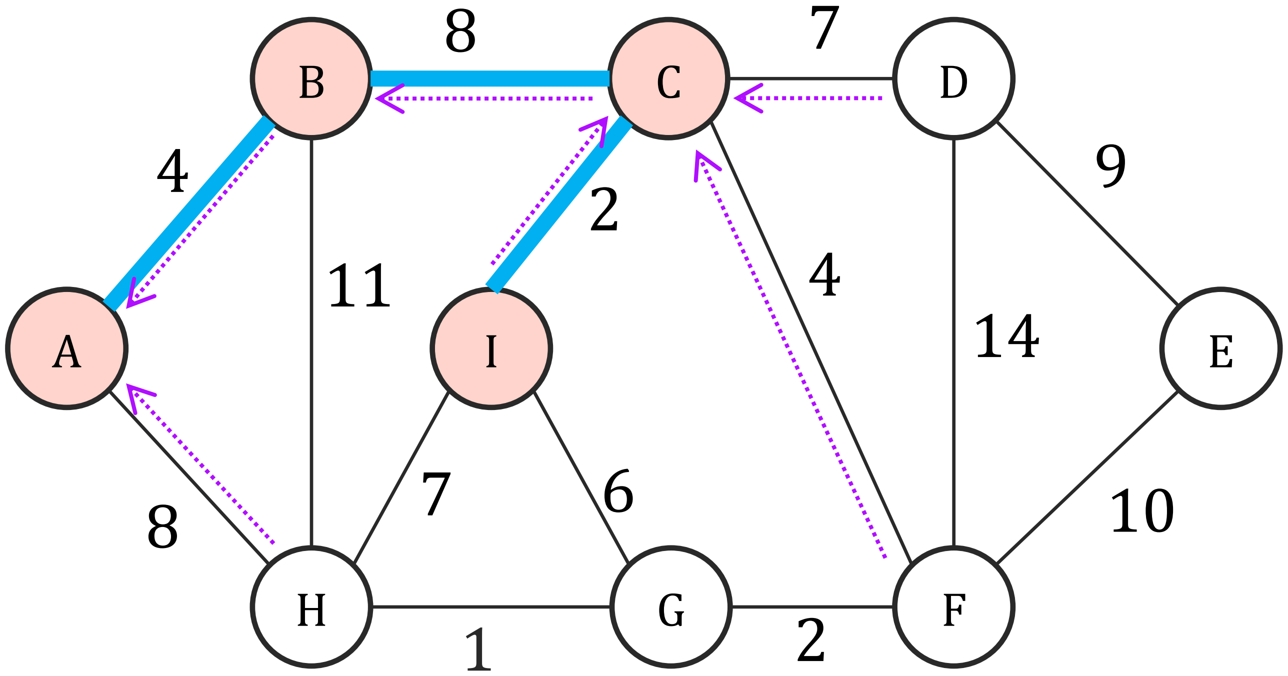
$Q.\text{decreaseKey}(z, w_{(v,z)})$

$prev[z] \leftarrow v$

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 $z \in Q$.

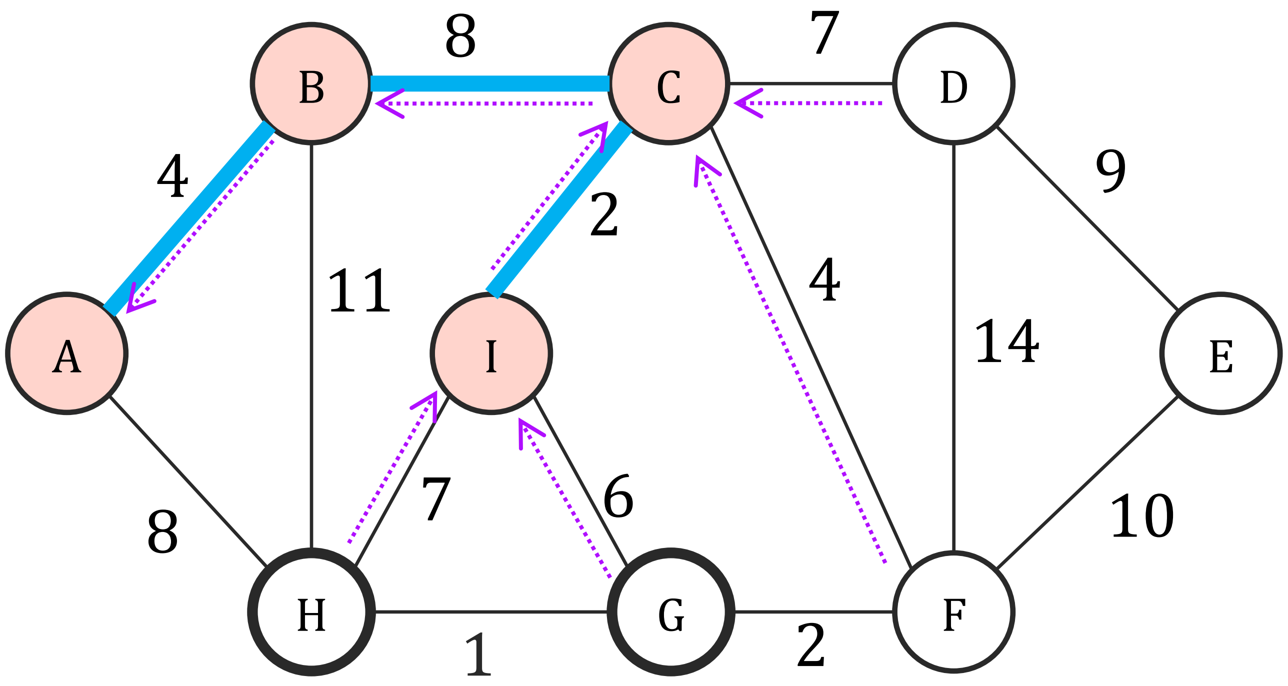
        Q.decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  return X

Prim's Algorithm: Efficient Implementation

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

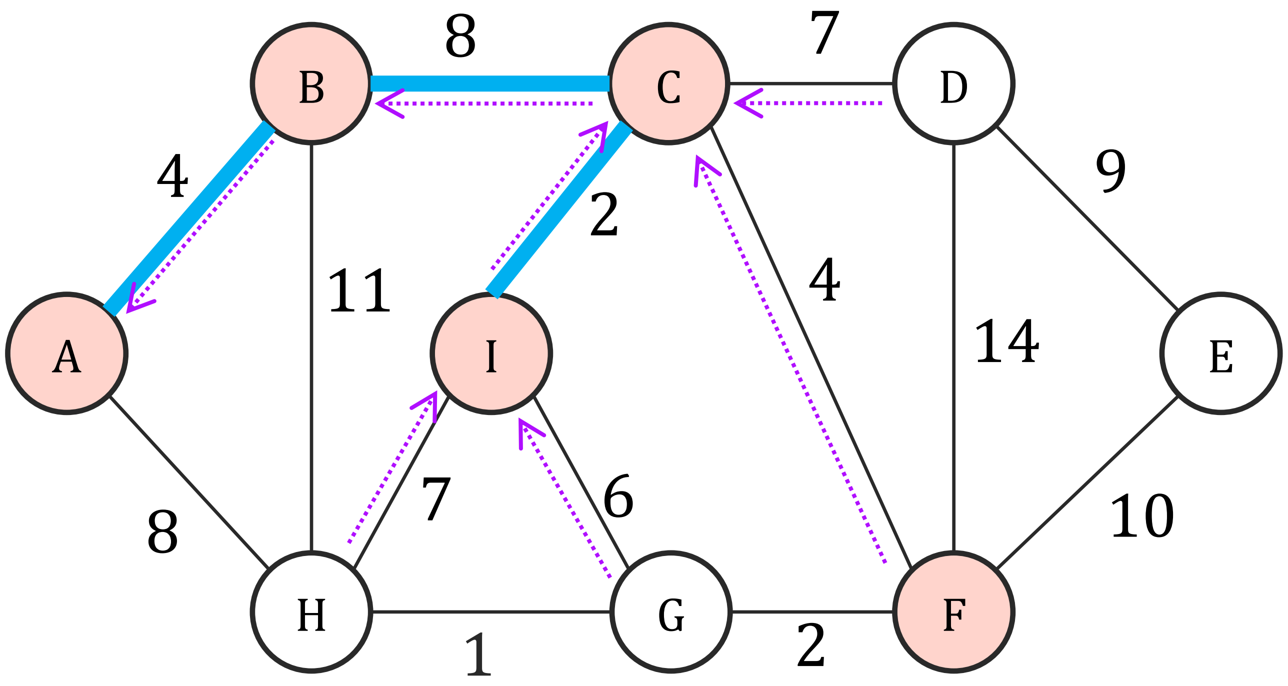


	A	B	C	D	E	F	G	H	I
dist	0	4	8	7	∞	4	**6**	**7**	2
prev	∅	*A*	*B*	*C*	∅	*C*	***I***	***I***	*C*

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*

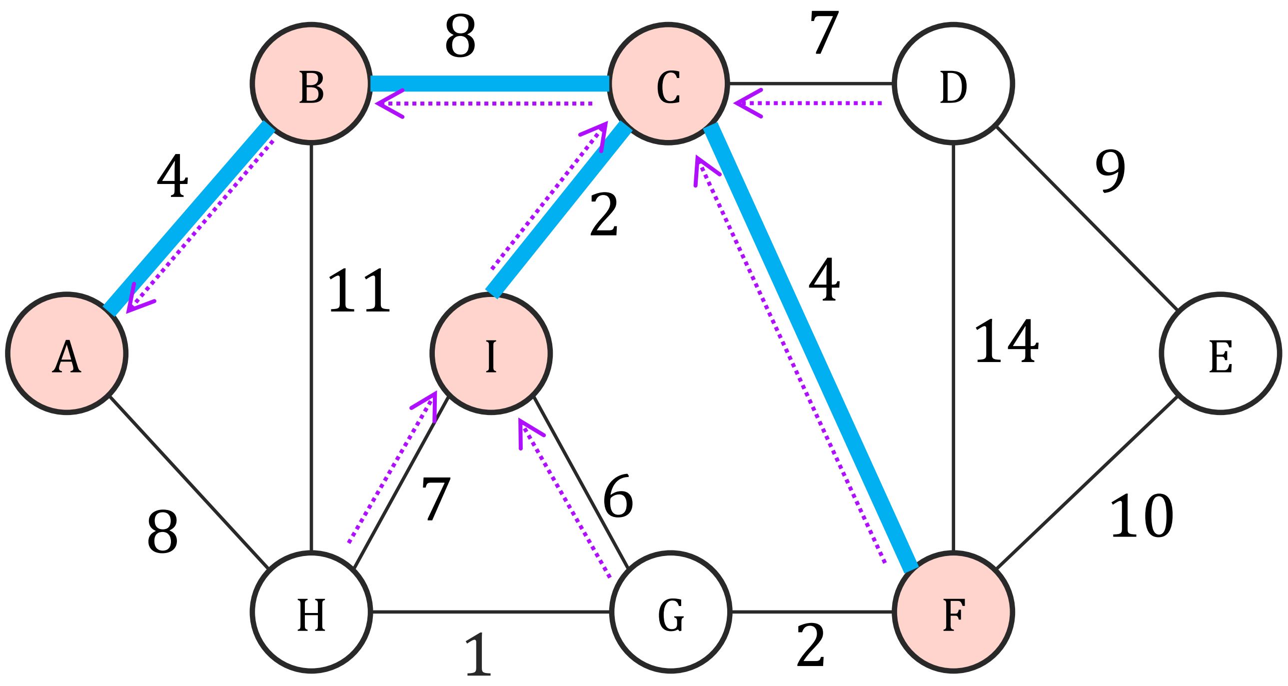


	A	B	C	D	E	F	G	H	I
dist	0	4	8	7	∞	4	6	7	2
prev	$\emptyset$	*A*	*B*	*C*	$\emptyset$	*C*	*I*	*I*	*C*

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

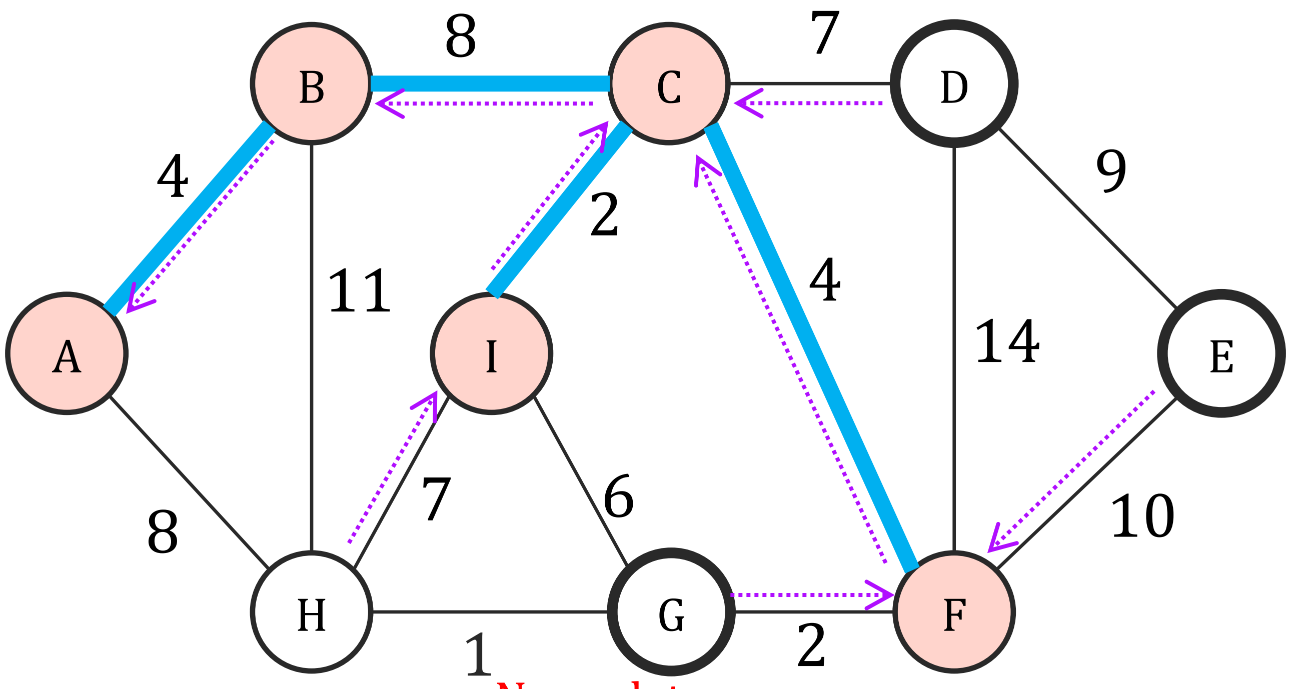
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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



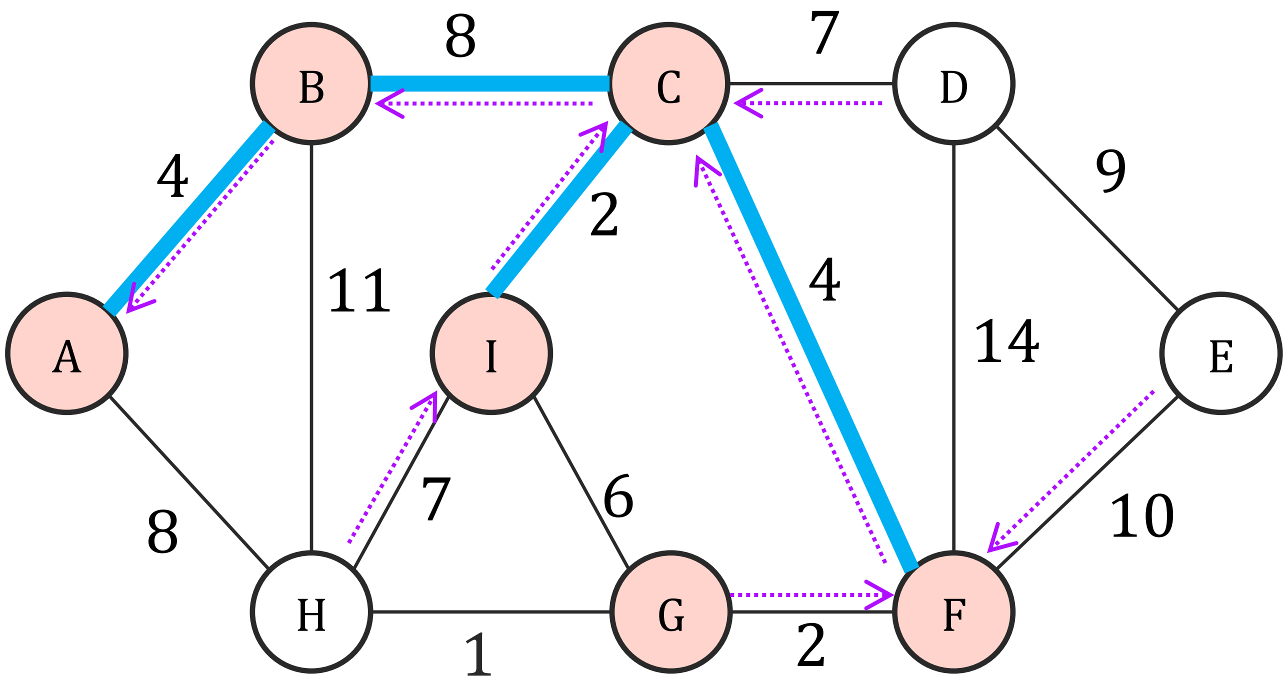
No update

	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** $v \in V$, 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	∅	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 $z \in Q$.

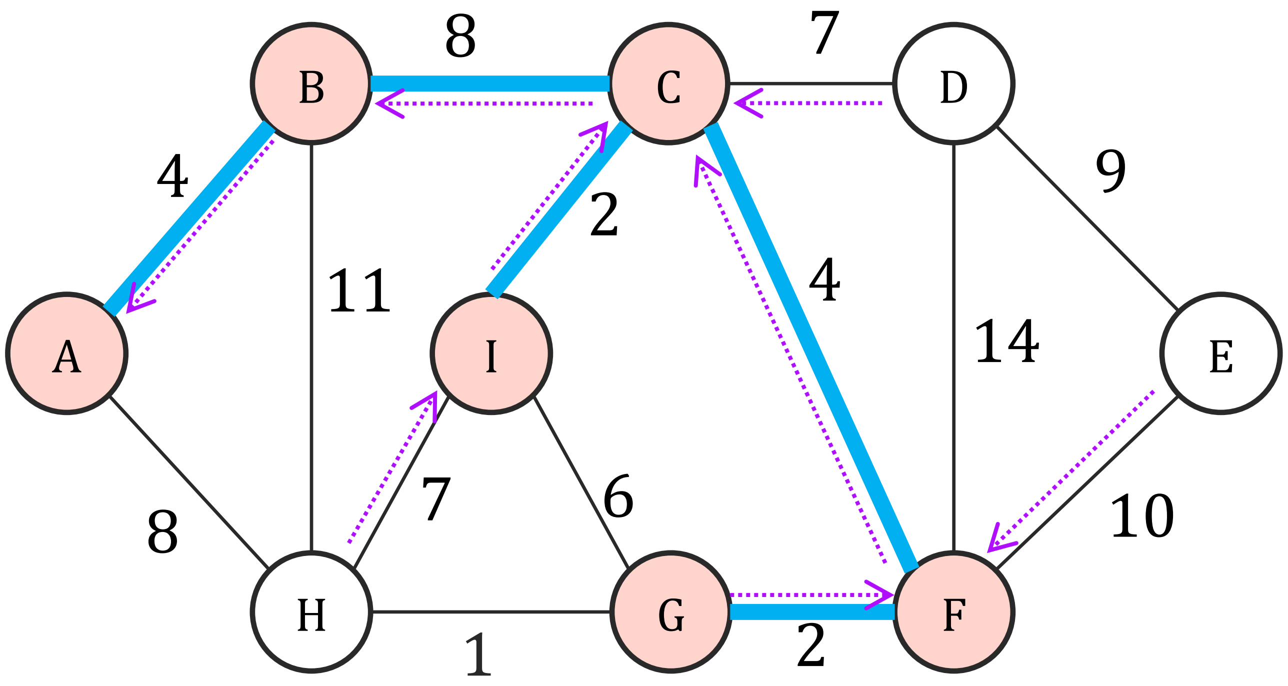
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

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	~~Ø~~	~~*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 $z \in Q$.

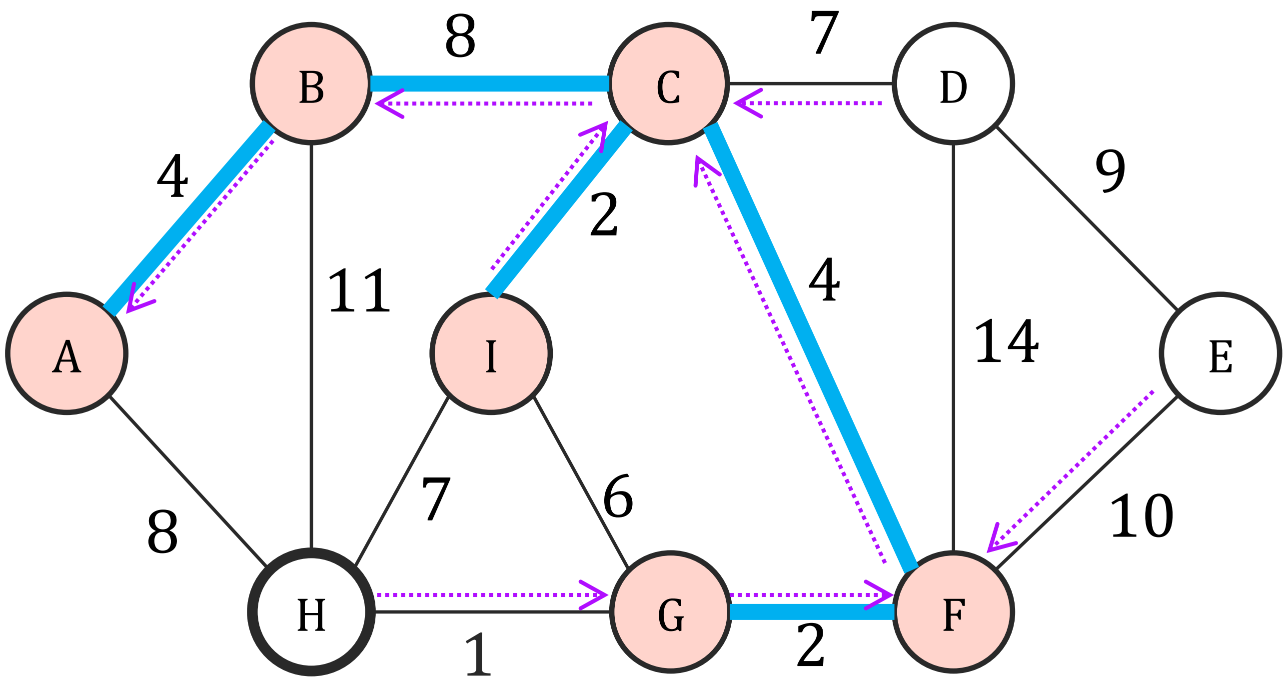
        Q. decreaseKey$(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  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	**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 $v \in V$, $Q.\text{insert}(v, dist[v])$

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 $z \in Q$.

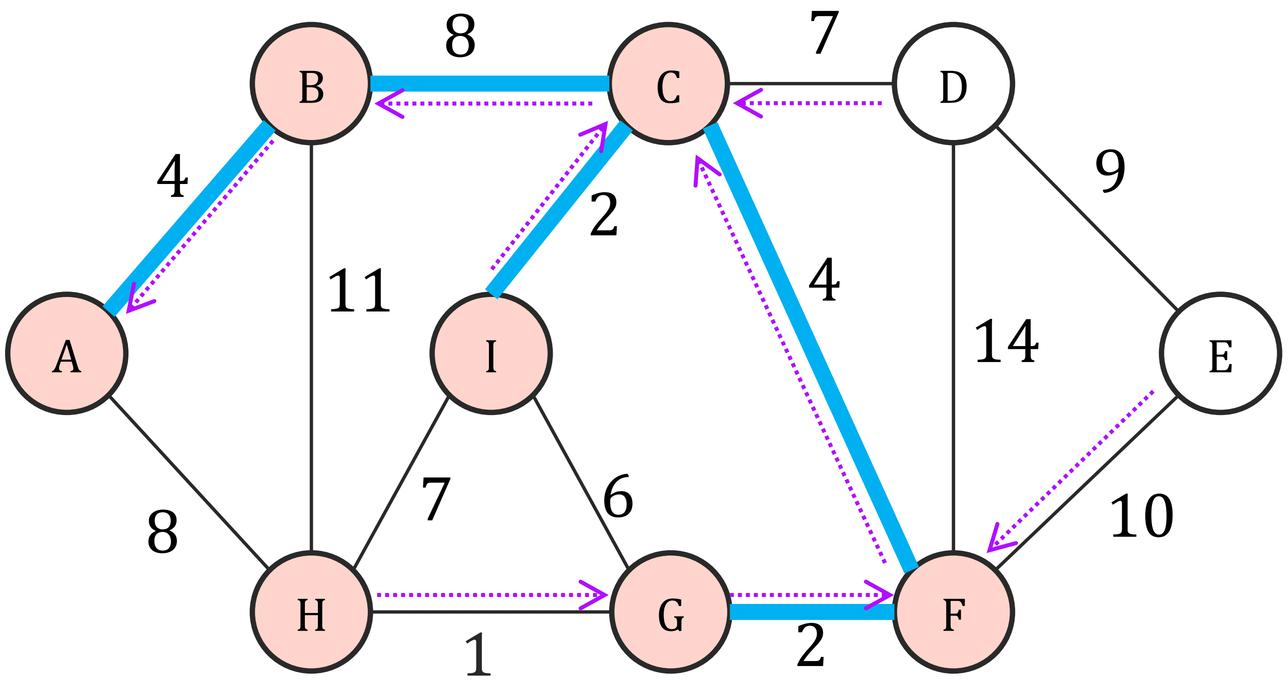
$Q.\text{decreaseKey}(z, w_{(v,z)})$

$prev[z] \leftarrow v$

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	∅	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 $v \in V$, $Q.\text{insert}(v, dist[v])$

  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 $z \in Q$.

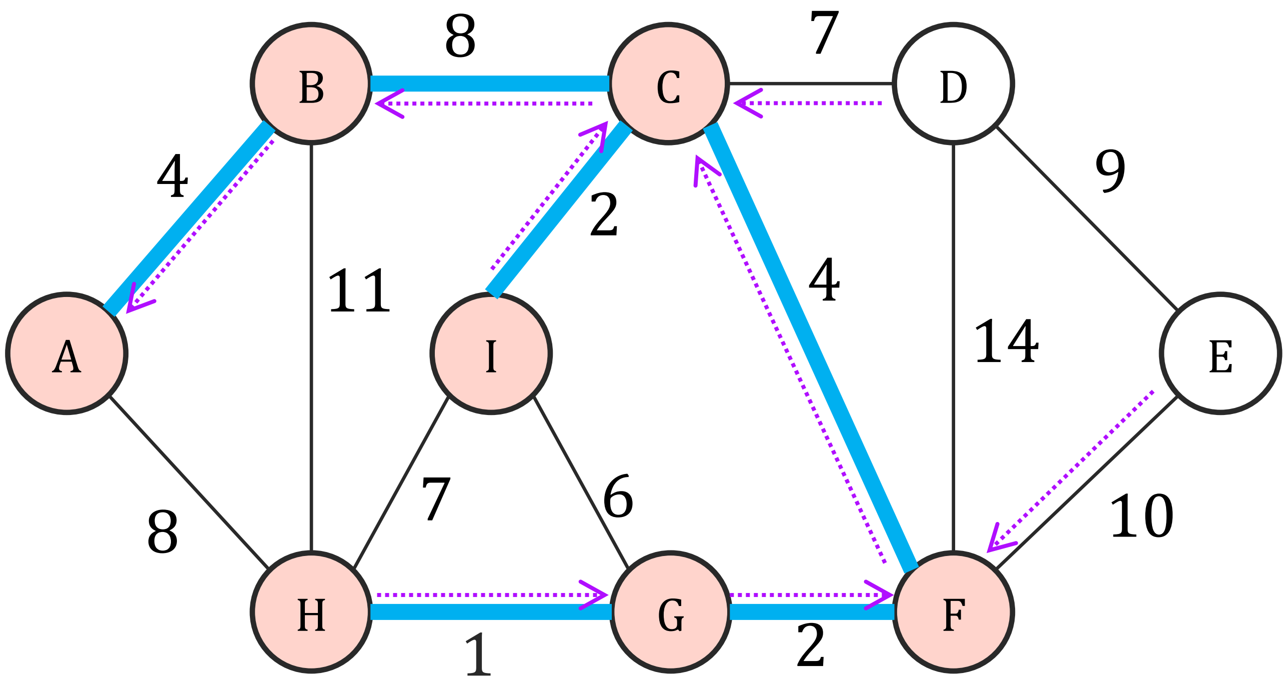
        $Q.\text{decreaseKey}(z, w_{(v,z)})$

        $prev[z] \leftarrow v$

  return X

Prim's Algorithm: Efficient Implementation

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

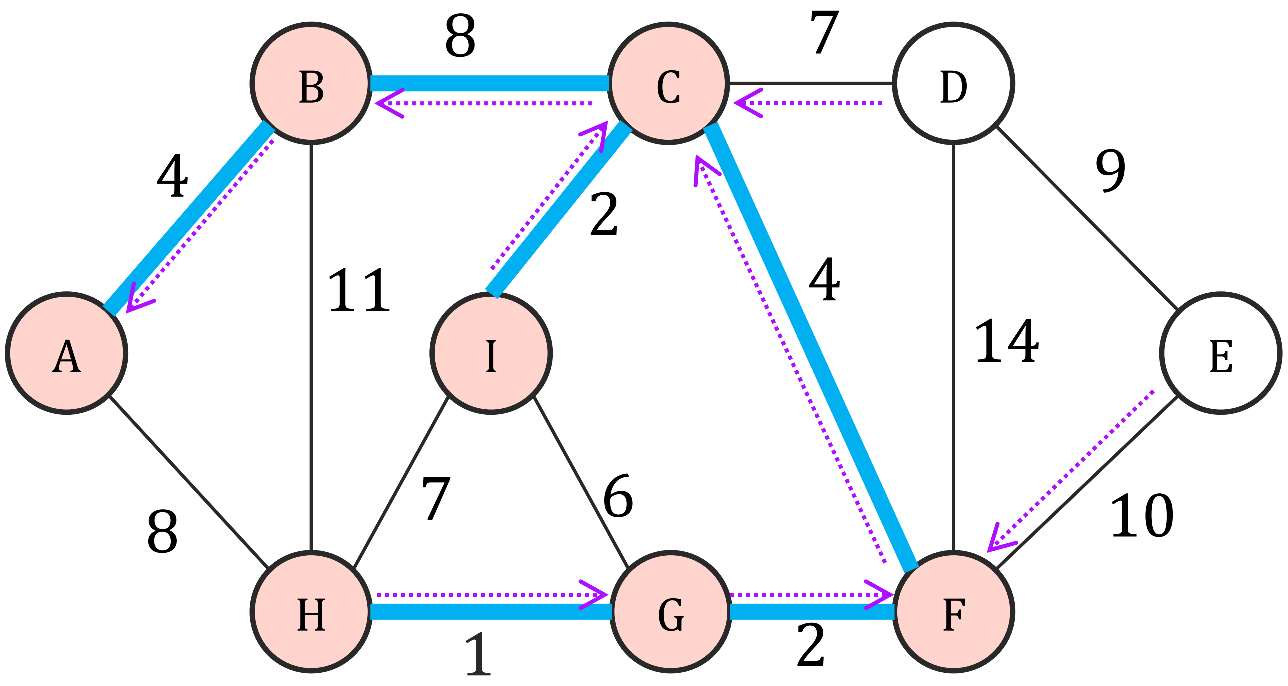


	A	B	C	D	E	F	G	H	I
dist	0	4	8	7	10	4	2	1	2
prev	∅	*A*	*B*	*C*	*F*	*C*	*F*	*G*	*C*

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	Ø	*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 $z \in Q$.

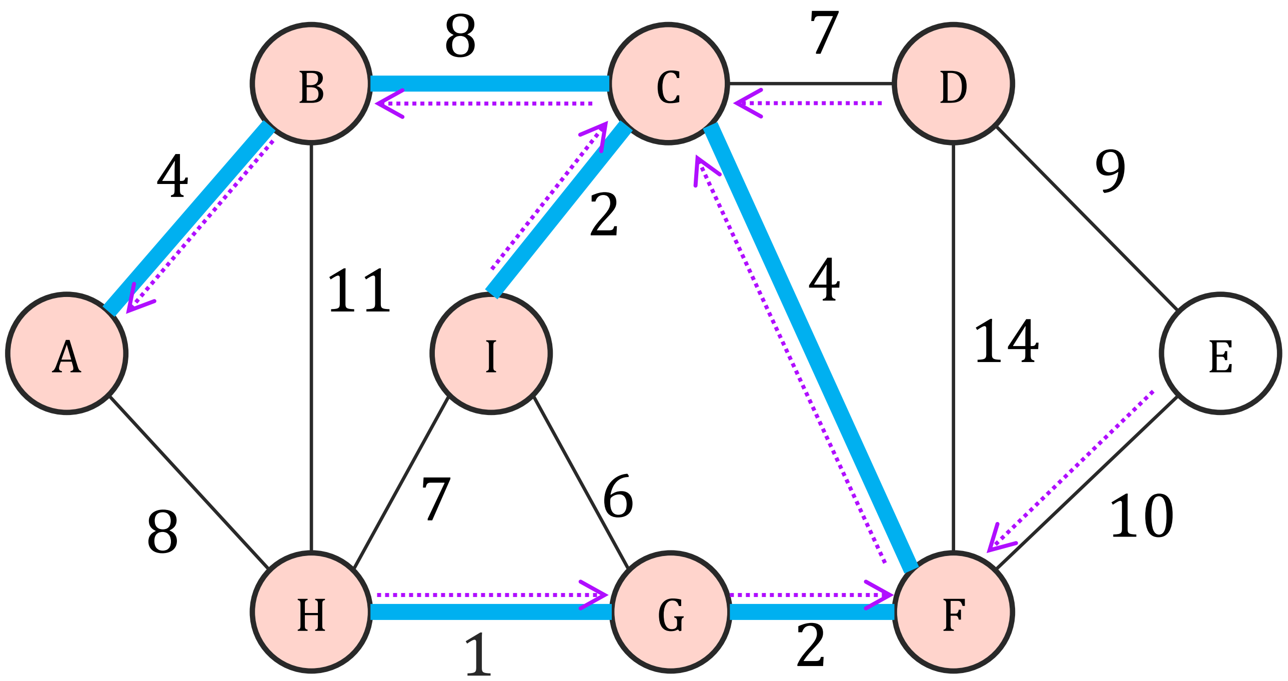
Q. decreaseKey$(z, w_{(v,z)})$

$prev[z] \leftarrow v$

return X

Prim's Algorithm: Efficient Implementation

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

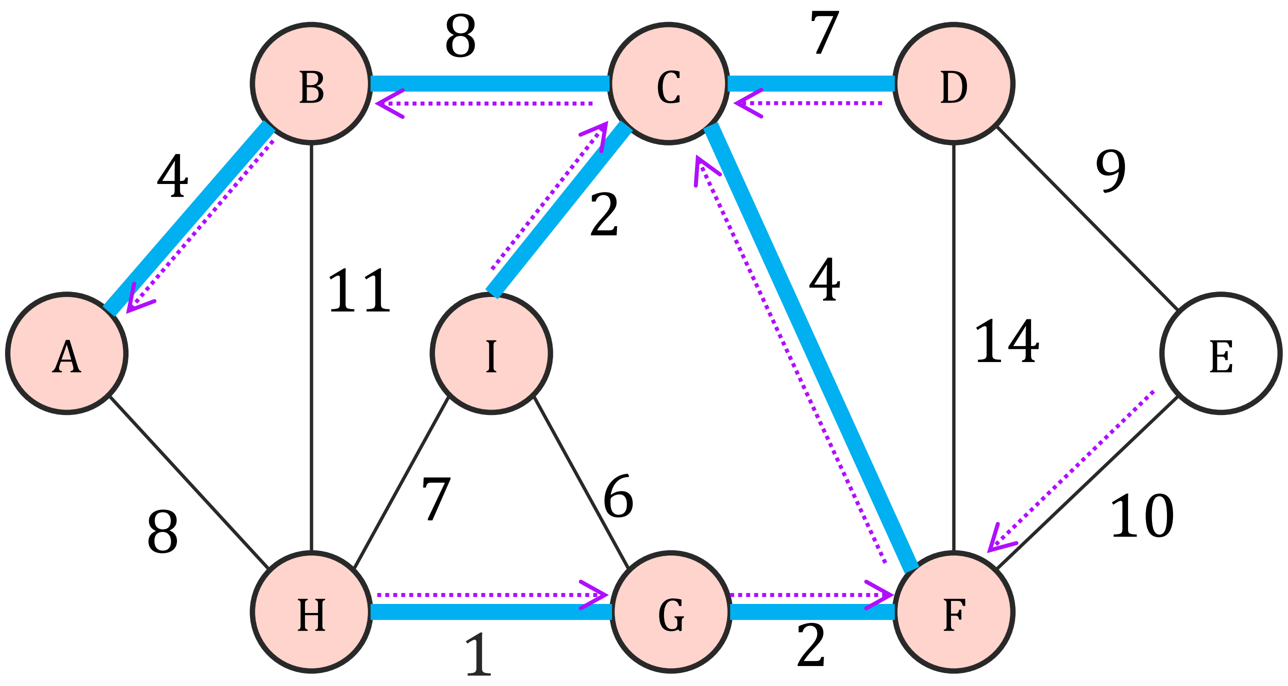


	A	B	C	D	E	F	G	H	I
dist	0	4	8	7	10	4	2	1	2
prev	∅	*A*	*B*	*C*	*F*	*C*	*F*	*G*	*C*

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*

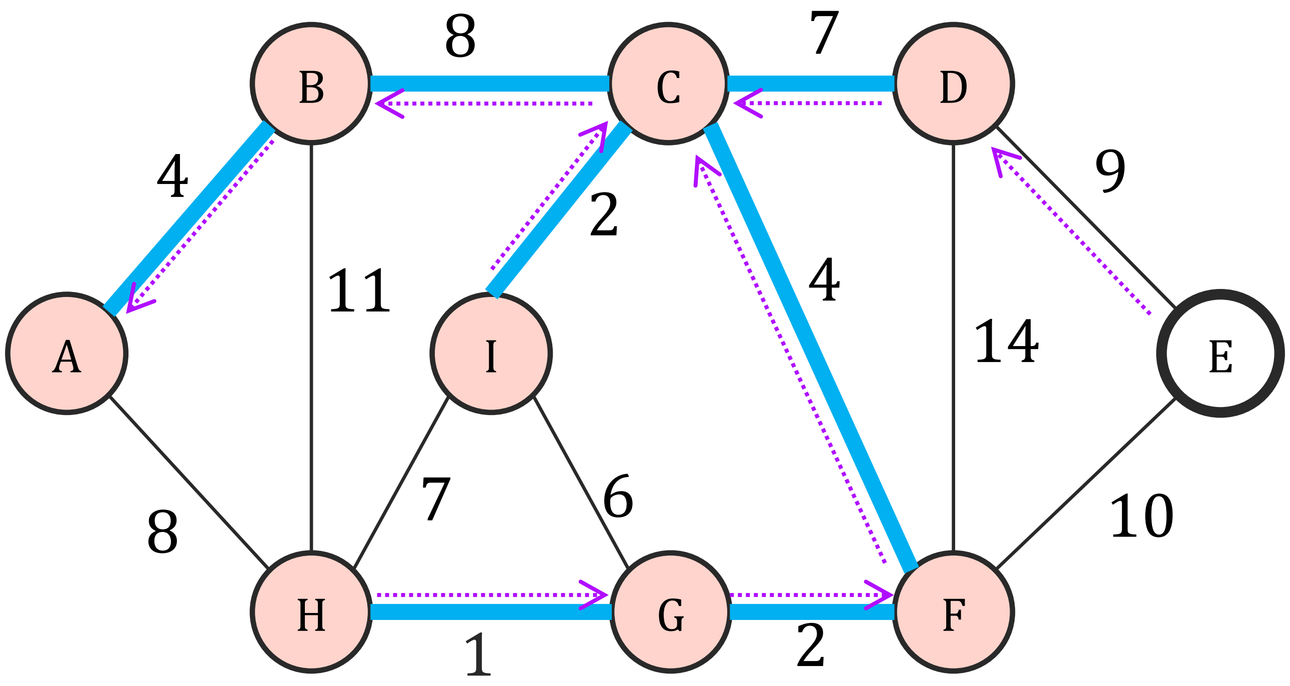


	A	B	C	D	E	F	G	H	I
dist	~~0~~	~~4~~	~~8~~	~~7~~	10	~~4~~	~~2~~	~~1~~	~~2~~
prev	~~∅~~	~~A~~	~~B~~	~~C~~	F	~~C~~	~~F~~	~~G~~	~~C~~

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 $v \in V$, $Q.\text{insert}(v, dist[v])$
 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 $z \in Q$.
    $Q.\text{decreaseKey}(z, w_{(v,z)})$
    $prev[z] \leftarrow v$
 return X

Prim's Algorithm: Efficient Implementation

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

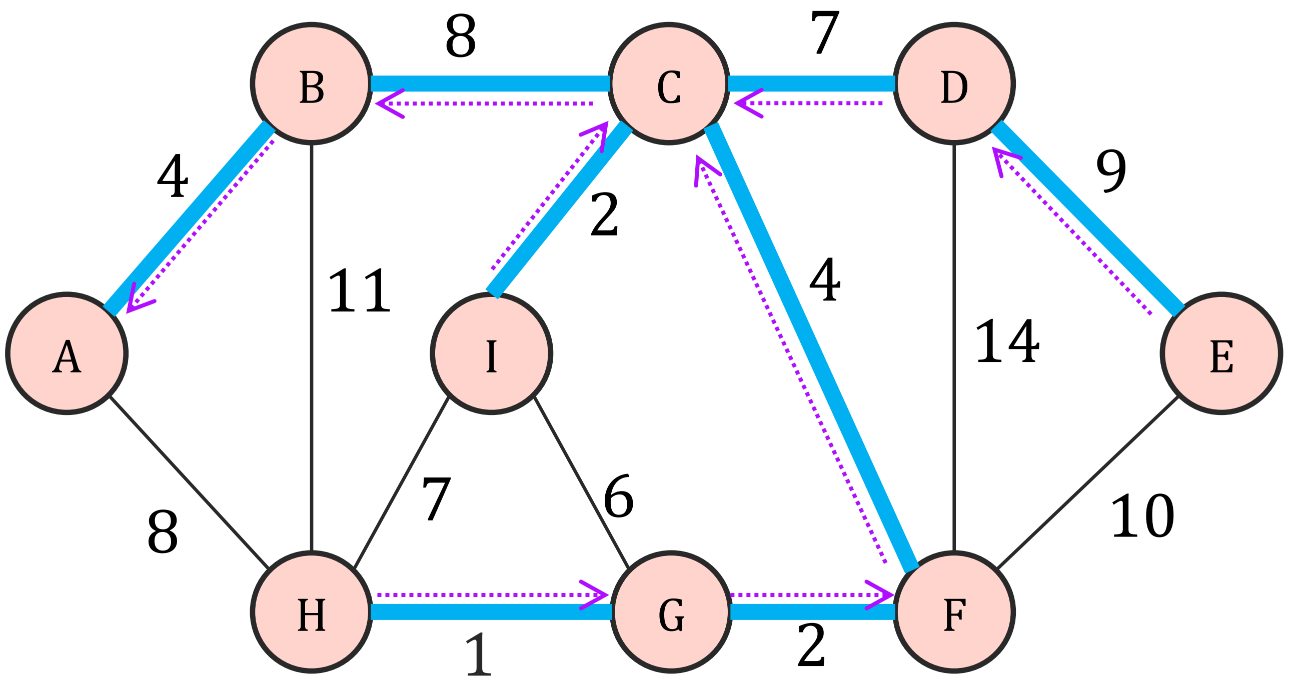


	A	B	C	D	**E**	F	G	H	I
dist	0	4	8	7	**9**	4	2	1	2
prev	∅	*A*	*B*	*C*	***D***	*C*	*F*	*G*	*C*

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	∅	*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 $v \in V$, 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

Comparing MST algorithm's runtimes

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

→ For sparse graphs ($\mathrm{m} = O(n)$), 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!

3 min break

(Please close the auditorium doors)

Next up: Last greedy algorithm!

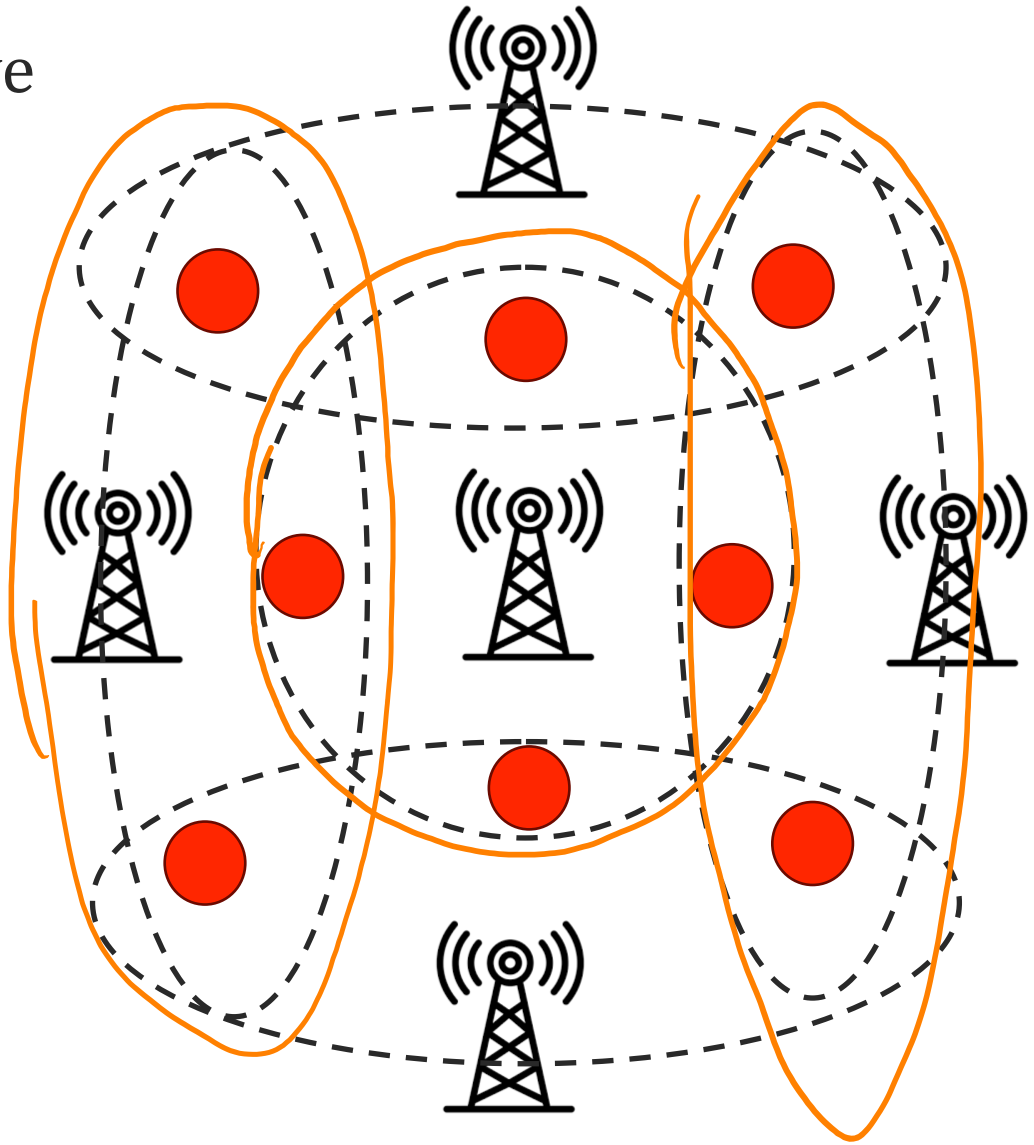
Covering

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:

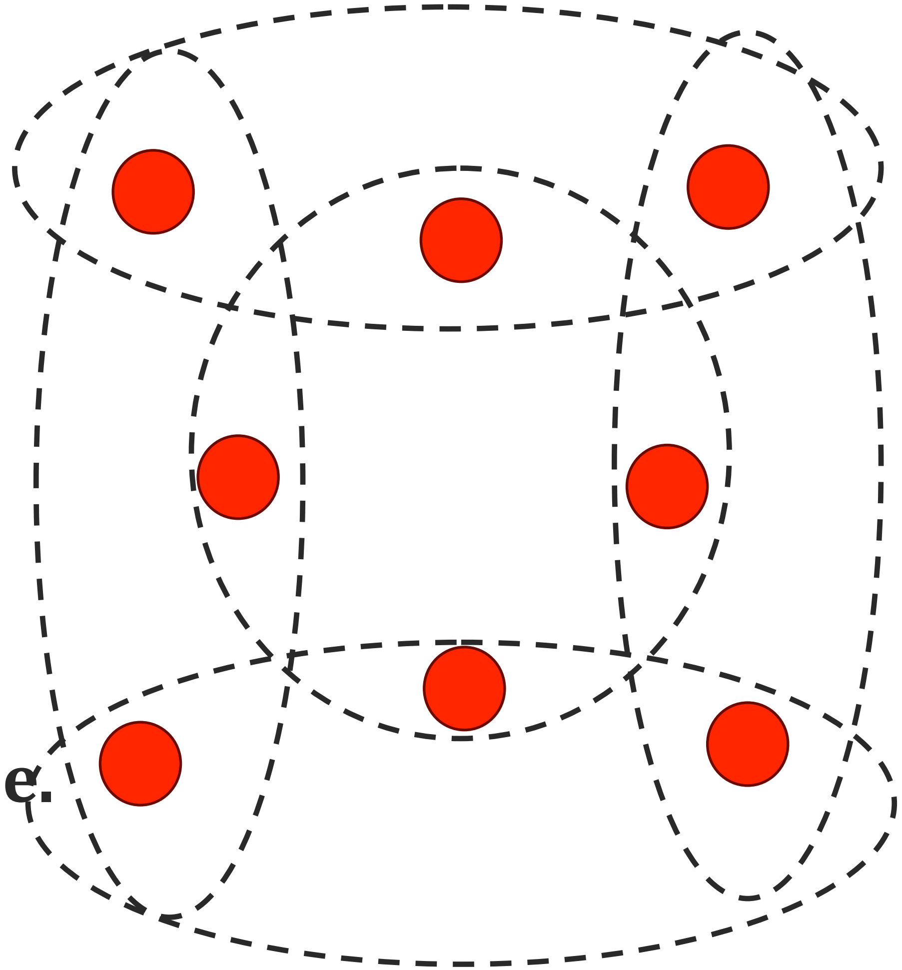
→Universe of n elements $U = \{1, \dots, n\}$, and

→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.**

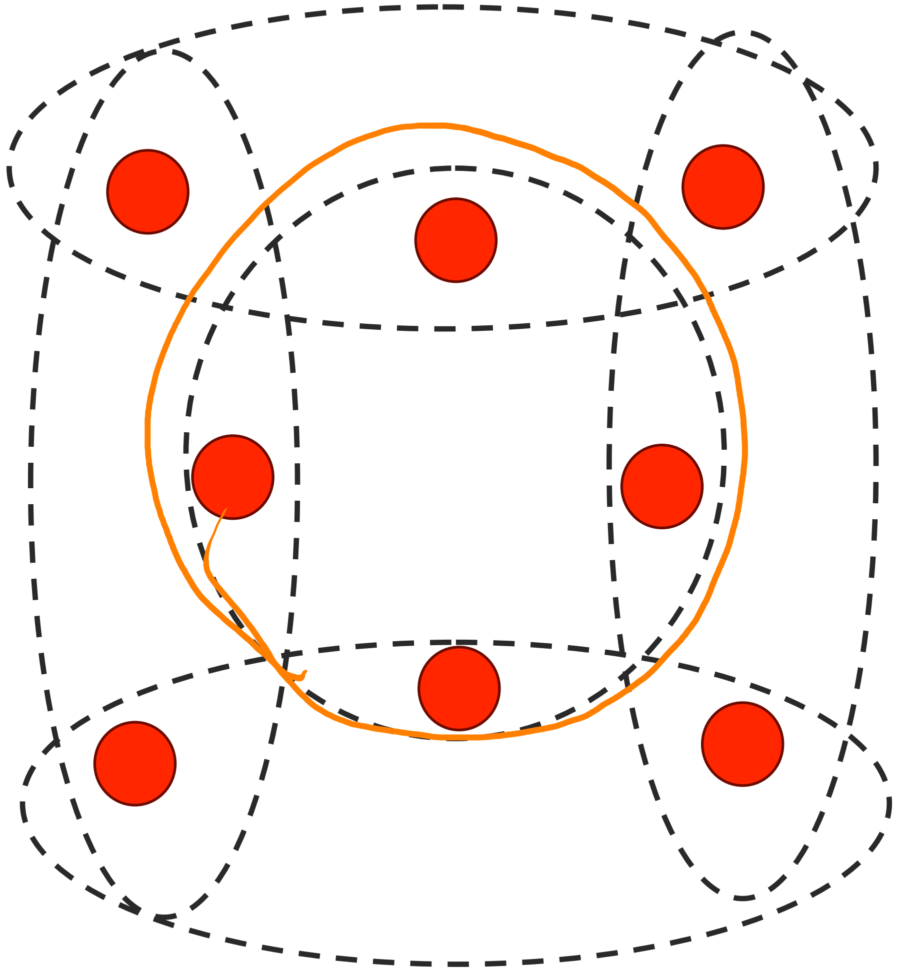
i.e., $J \subseteq \{1,2,\dots,m\}$ s.t., $\bigcup_{i \in J} S_i = U$



Greedy Algorithm

Discuss

What is a good greedy algorithm?



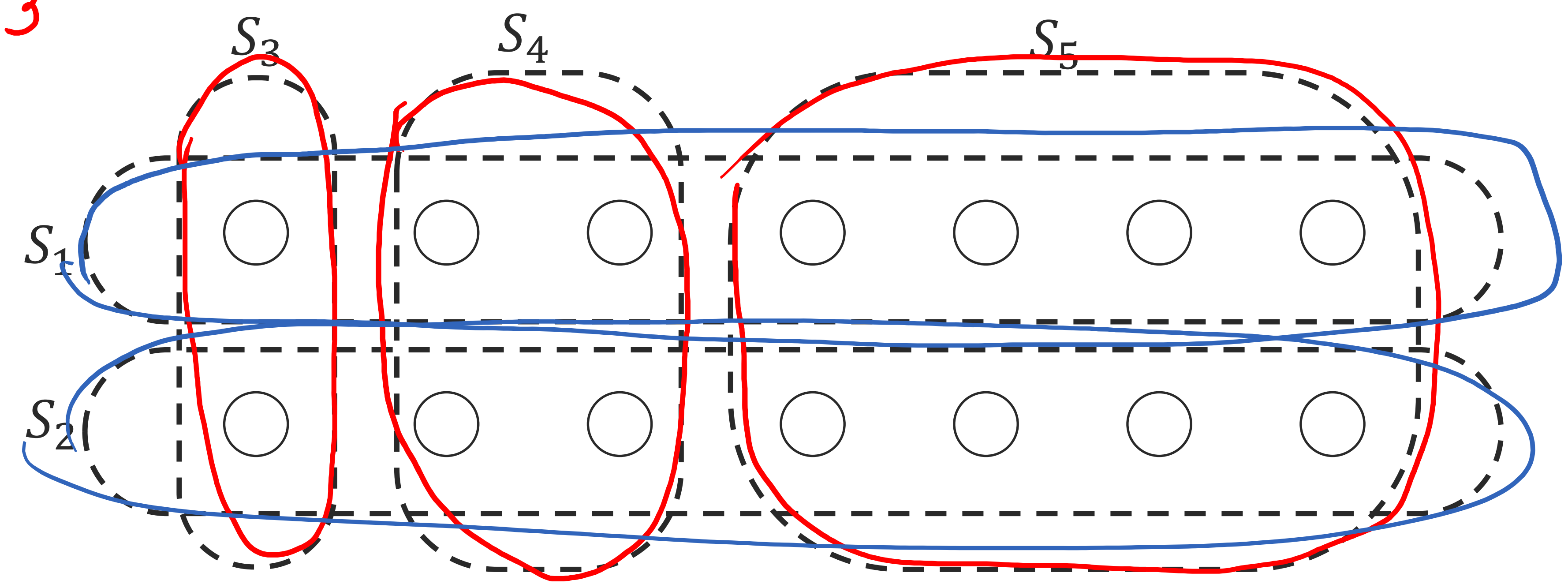
Greedy Algorithm for Set Cover

A suggested greedy algorithm:

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

$|\text{Greedy}| = 3$

S_5
S_4
S_3



$|\text{Opt}| = 2$

S_1
S_2

Greedy is not optimal for Set Cover

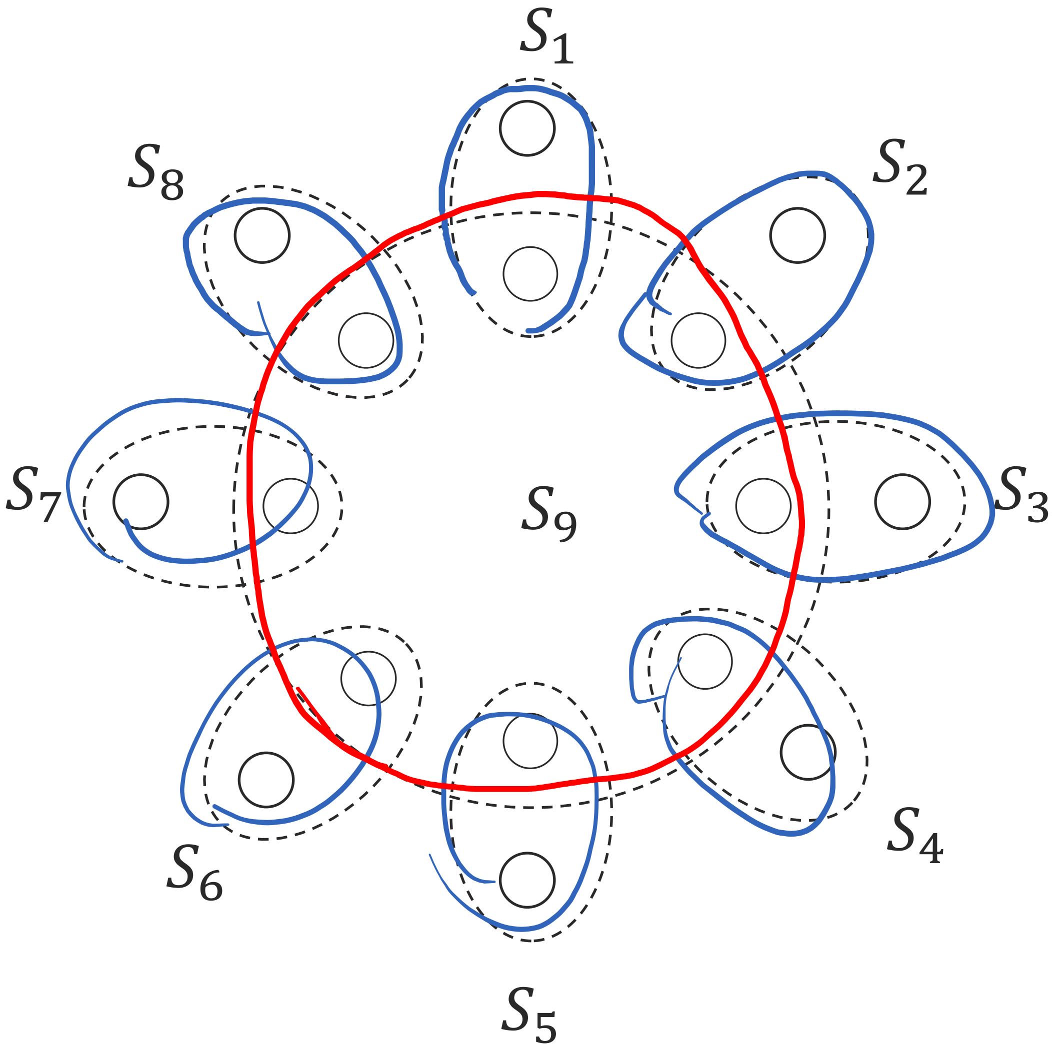
One other example where this greedy algorithm is not optimal

$|Greedy| = 9$

$S_9, S_1, \dots, S_8$

$|Opt| = 8$

$S_1 \dots S_8$

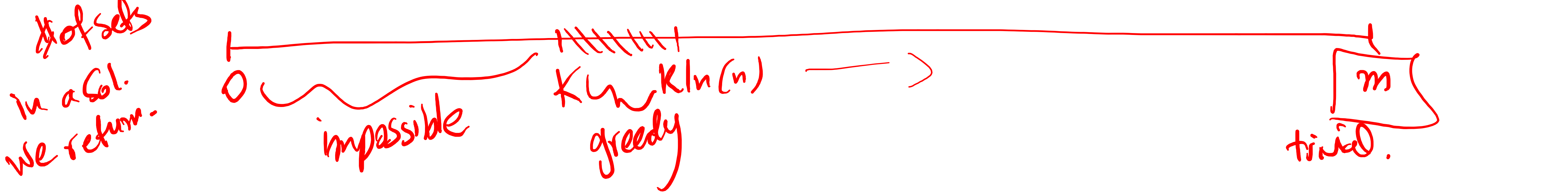


$$\frac{|Greedy|}{Opt} = 9/8$$

Greedy is Approximately Optimal

Claim: For any instance of the Set Cover problem. If the optimal solution uses k sets, the Greedy algorithm uses at most $k \ln(n)$ sets.

take all $S_1 \cdots S_m$



Proof: Let n_t be the number of elements not covered after t step of the Greedy algorithm. (E.g., $n_0 = n$).

Our goal: Show that for $\boxed{t = k \ln(n)}$, $\boxed{n_t < 1}$.
→ If we achieve this goal; then we have $n_t = 0$. i.e., all elements of the set are covered by Greedy after $k \ln(n)$ rounds.

Greedy is Approximately Optimal

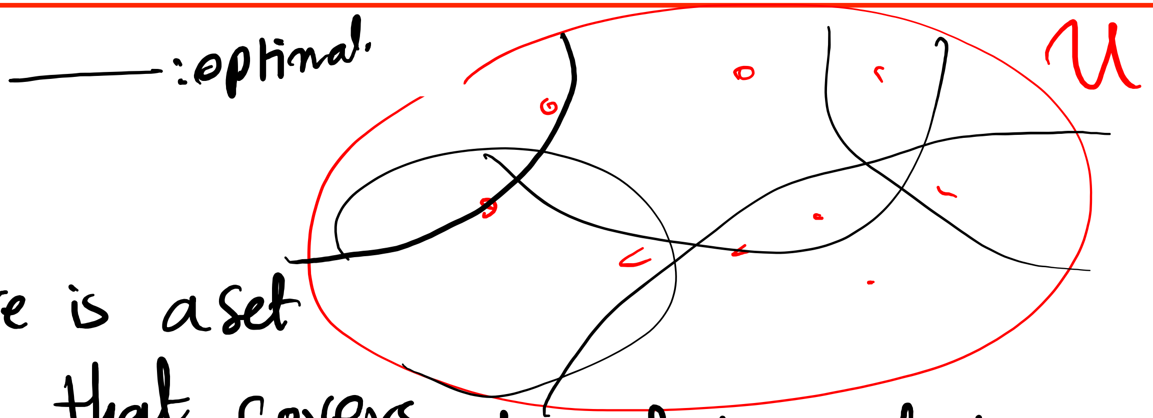
Let n_t be the number of elements not covered after t step of the Greedy algorithm.

Our goal: Show that for $t = k\ln(n)$, $n_t < 1$.

Subclaim 1: $n_1 \leq n_0 - \frac{n_0}{k}$

Proof:

Optimal: opt chooses k sets to cover U $\Rightarrow$ there is a set S that covers at least $\frac{n}{k}$ of the elements.



Greedy: Solutions picks the largest $\Rightarrow$ Greedy also covers at least $\frac{n}{k}$ elements

$n_0 = n$

$\hookrightarrow \underbrace{n_1}_{\text{still uncovered by greedy}} \leq n_0 - \frac{n_0}{k}$,

Greedy is Approximately Optimal

Let n_t be the number of elements not covered after t step of the Greedy algorithm.

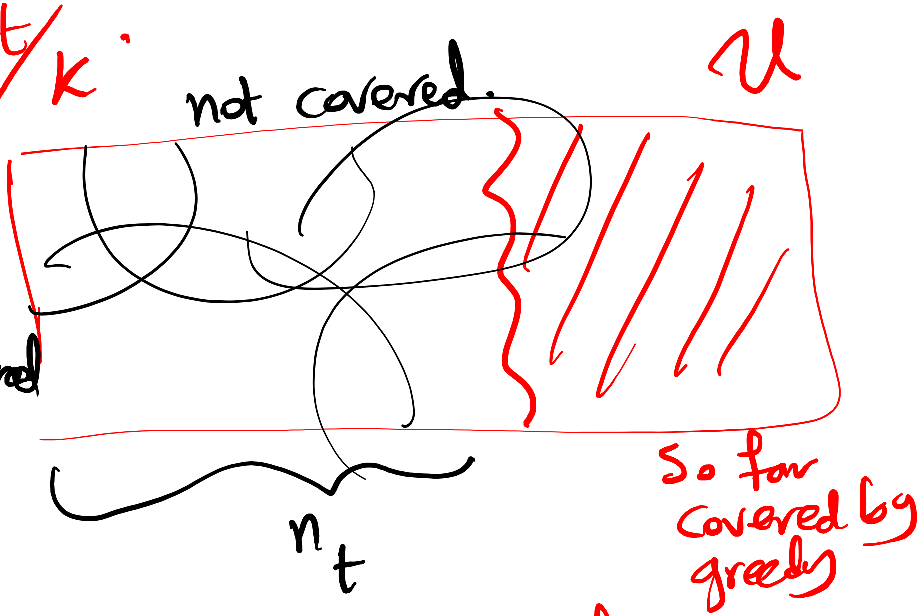
Our goal: Show that for $t = k \ln(n)$, $n_t < 1$.

Subclaim 2: For any t, $n_{t+1} \le n_t(1 - 1/k) = n_t - n_t/k$.

optimal Sol: Covers all of the n_t points. — opt

↳ there is a set S, s.t.

S covers $\geq n_t/k$ of the so far uncovered elements.



Picked set

↳ Greedy: largest # of uncovered points.

↳ Greedy picks some set covering $\geq \frac{n_t}{k}$ of the so far uncovered points.

$n_{t+1} \le n_t - n_t/k = n_t(1 - 1/k)$

Greedy is Approximately Optimal

Let n_t be the number of elements not covered after t step of the Greedy algorithm.

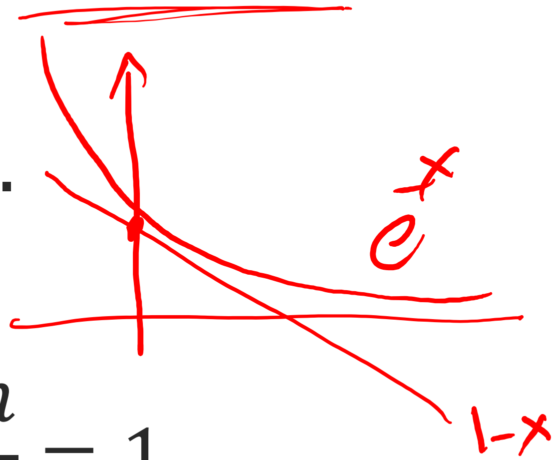
Our goal: Show that for $\boxed{t = k\ln(n),\, n_t < 1.}$

Repeatedly applying subclaim 2, we have that for any t

$$n_t \le n_{t-1}\left(1-\frac{1}{k}\right) \le n_{t-2}\left(1-\frac{1}{k}\right)^2 \le \cdots \le n_0\left(1-\frac{1}{k}\right)^t = n\left(1-\frac{1}{k}\right)^t$$

Final subclaim: $n\left(1-\frac{1}{k}\right)^{\boxed{k\ln(n)}} < 1.$

Proof: We use a mathematical fact that for any $x \ne 0$, $\underline{1 - x < e^{-x}}$.



$$n\underbrace{\left(1-\frac{1}{k}\right)}_{e^{-1/k}}^{k\ln(n)} < n e^{\frac{-1}{\cancel{k}} \times \cancel{k} \ln n} = n\underbrace{e^{-\ln(n)}}_{1/n} = \frac{n}{n} = 1$$

Approximation Factor

We showed that Greedy does not find the optimal set cover.

We also showed that Greedy outputs $\leq k \ln(n)$ sets, where $k = \text{OPT}$ is the number of sets used in the optimal solution.

"Greedy has an **approximation factor** of $\ln(n)$ for Set Cover"

Formally, approximation factor of an algorithm (for minimizing cost) is

$$\frac{\text{Cost(Alg(input x))}}{\text{Cost(optimal solution for input x)}} = \frac{K \ln(n)}{K} = \ln(n)$$

What is the best polynomial time approximation algorithm for Set Cover?
Greedy! Meaning, **approximation factor** $< \ln(n)$ is not achievable in polynomial time.

Show at home: Greedy's approximation factor is no better than $\ln(n)$.
→ Generalize our first "bad" example showing Greedy is not optimal.

Wrap up

Done with being greedy!

→ Mastering proof by induction!

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

Next time

- Dynamic Programming