CS 590: Algorithms

Lecture 10 & 11: Graphs & Graph’s algorithm

Minimum Spanning Trees

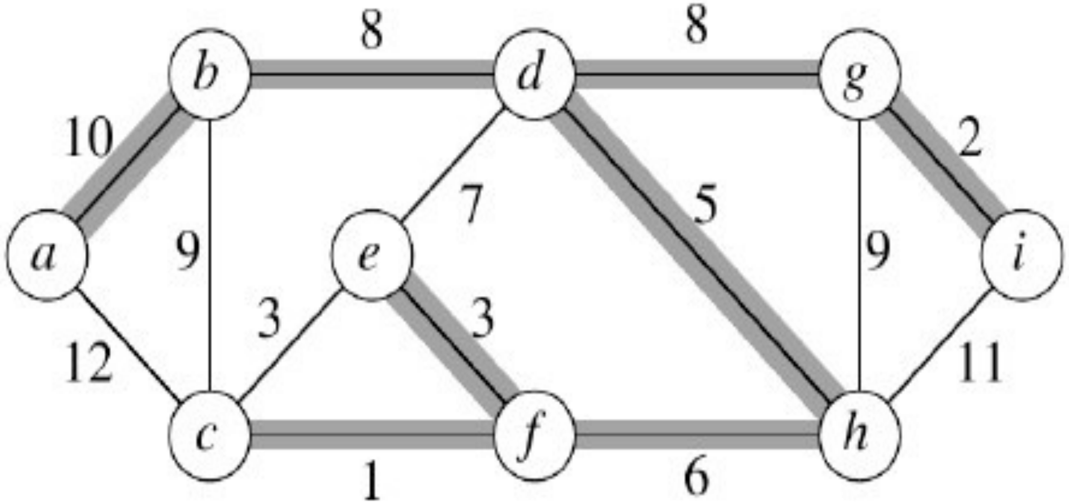
Problem:

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- **Goal**: Repair enough (and no more) roads such that
  - Everyone stays connected: can reach every house from, all other houses, and
  - Total repair cost is minimum.

Model as graph:

- Undirected graph $G = (V, E)$.
- Weight $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that
  - T connects all vertices (T is a **spanning tree**), and
  - $w(T) = \sum_{(u,v)\in T} w(u, v)$ is **minimized**.
  - $\min\big(w(T)\big)$ is called a **minimum spanning tree (MST).**

Minimum Spanning Trees



- We have more than one MST in this example.
- Replace (e,f) in the MST by (c,e) gives a different MST with the same weight.

Some properties of a MST:

- It has |V|-1 edges.
- It has no cycles.
- It might not be unique.

Minimum Spanning Trees

Building up the solution:

- We will build a set *A* of edges.
- Initially, *A* has no edges.
- As we add edges to *A*, we maintain a loop invariant: *A* is a subset of some MST.
- We add only edges that maintain the loop invariant.
- If *A* is a subset of some MST, an edge (*u*, *v*) is safe for *A* if and only if $A \cup \{(u, v)\}$ is also a subset of some MST ⇒ we will only add safe edges.

Algorithm (GENERIC-MST(G,w))

```
1 A = ∅
2 while (A is not in a spanning tree) do
3    find an edge (u, v) that is safe for A
4    A = A ∪ {(u, v)}
5 return A
```

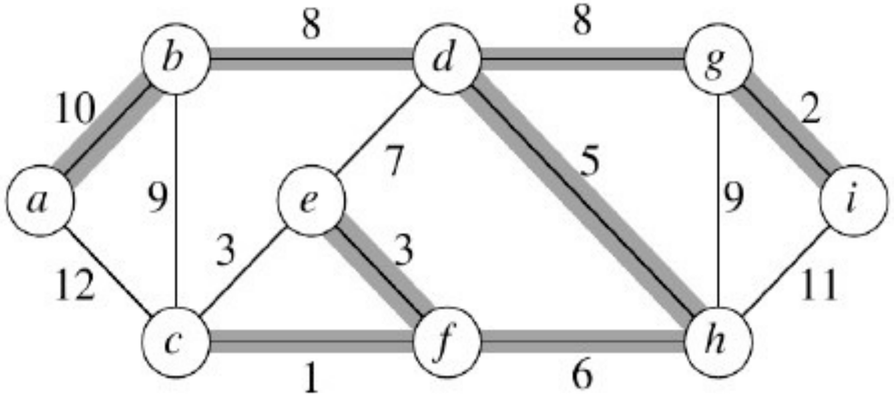
Minimum Spanning Trees

Correctness:

- Initialization: The empty set trivially satisfies the loop invariants.
- Maintenance: A remains a subset of some MST, since we add only safe edges.
- Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Find a safe edge:

- How do we find safe edges?
- In this example, the edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?



Minimum Spanning Trees

Intuitively:

- Let $S \subset V$ be any set of vertices that includes c but not f (so that f is in $V - S$).
- In any MST, there has to be one edge (at least) that connects S with $V - S$.
- Why not choose the edge with minimum weight? (would be (c, f) in our case).

Definition:

- Let $S \subset V$ and $A \subseteq E$.
  - A **cut** $(S, V - S)$ is a partition of vertices into disjoint set V and $S - V$.
  - Edge $(u, v) \in E$ **crosses** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
  - A cut **respects** A if and only if no edge in A crosses the cut.
  - An edge is a **light edge** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edges crossing it.

Minimum Spanning Trees

Theorem:

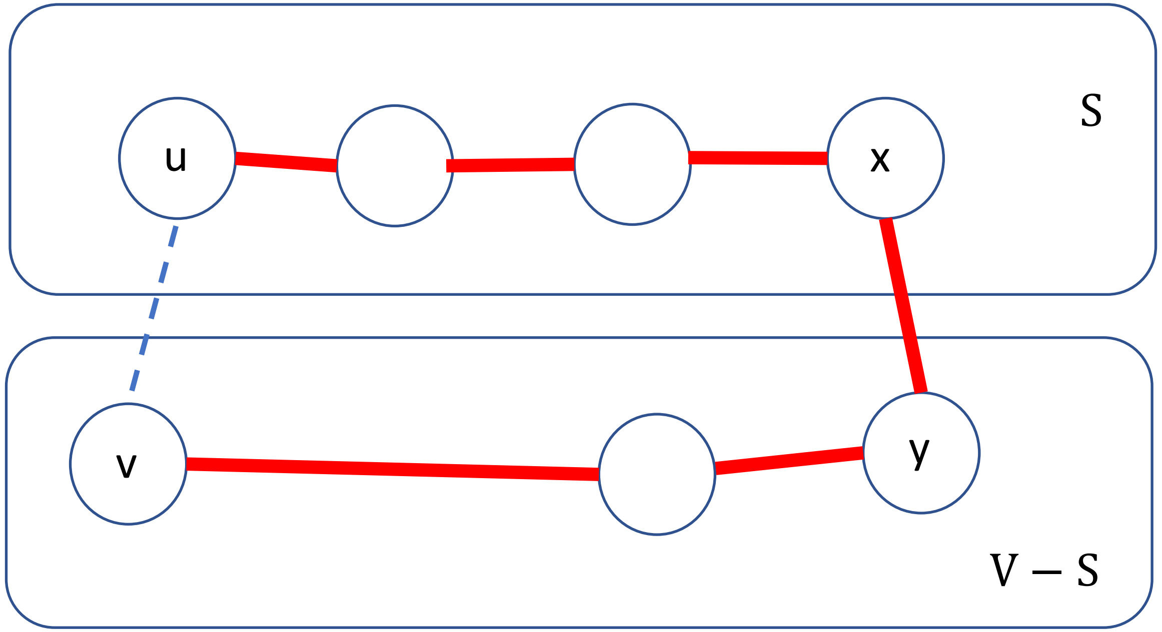
- Let A be a subset of some MST, $(\mathrm{S}, \mathrm{V}-\mathrm{S})$ be a cut that respects A, and (u, v) be a light edge crossing $(\mathrm{S}, \mathrm{V}-\mathrm{S})$. Then (u, v) is safe for A.

Minimum Spanning Trees

Proof:

- Let T be an MST that includes A.
- We are done, if T contains (u, v).
- We assume that T does not contain (u, v). We will construct a different MST T' that includes $\mathrm{A} \cup \{(\mathrm{u}, \mathrm{v})\}$.
- Recall, a tree has a unique path between each pair of vertices.
- Since T in an MST, it contains a unique path p between u and v.
- The path p must cross the cut $(\mathrm{S}, \mathrm{V} - \mathrm{S})$ at least once. Let (x, y) be an edge of p that crosses the cut. From the way (u, v) is chosen, we must have $\mathrm{w}(\mathrm{u}, \mathrm{v}) \leq \mathrm{w}(\mathrm{x}, \mathrm{y})$.

Minimum Spanning Trees



Proof:

- Since the cuts respect A, edge (x, y) is not in A.
- To form T' from T:
  - Remove (x, y) . Breaks T into two components.
  - Add (u, v). Reconnects.
  - So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Minimum Spanning Trees

- T' is a spanning tree.

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

  where $w(u, v) \leq w(x, y)$.
- Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.
- We need to show that (u, v) is safe for A:
  - $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
  - $A \cup \{(u, v)\} \subseteq T'$.
  - Since T' in an MST, (u, v) is safe for A.

Minimum Spanning Trees

In GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one.
- We can consider each component as a tree.
- Since an MST has exactly $|V| - 1$ edges, the for-loop iterates $|V|— 1$ time $\Rightarrow$ After adding $|V| - 1$ safe edges, we are down to just one component.

Minimum Spanning Trees

Corollary:

If $\mathrm{C} = (\mathrm{V_C}, \mathrm{E_C})$ is a connected component in the forest, $\mathrm{G_A} = (\mathrm{V}, \mathrm{A})$ and (u, v) is a light edge connecting C to some other component in $\mathrm{G_A}$(i.e., (u, v) is a light edge crossing the cut $(\mathrm{V_C}, \mathrm{V} - \mathrm{V_C})$), then (u, v) is safe for A.

Kruskal's Algorithm

- It leads to Kruskal's algorithm to solve the MST problem.
- Let $G = (V, E)$ is connected, undirected, weighted graph, $w: E \to \mathbb{R}$.
  - Starts with each vertex being its own component.
  - Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them.)
  - Scans the set of edges in monotonically increasing order by weight.
  - Uses a disjoint set data structure to determine whether an edge connects vertices in different components.

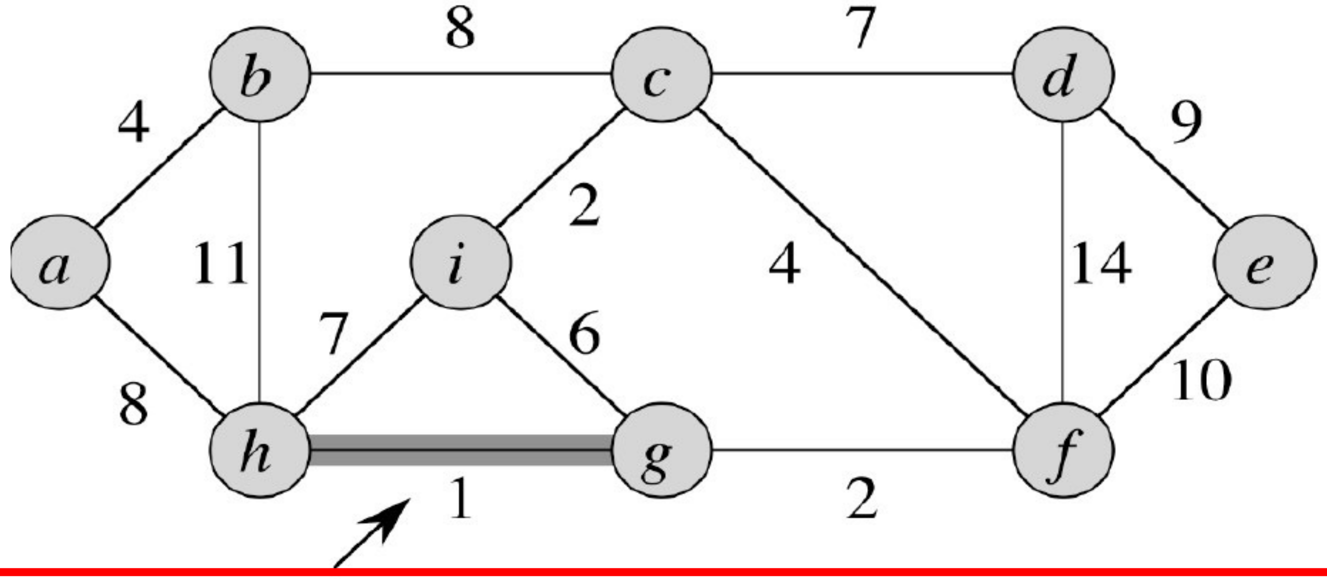
Kruskal's Algorithm

Algorithm (KRUSKAL(G,w))

```
A = ∅
foreach (vertex v ∈ G.V) do
  MAKE-SET(v)
#sort the edge of G.E into non-decreasing order by weight w
foreach ((u,v) taken from the sorted list) do
  if (FIND-SET(u)≠FIND-SET(v)) then
    A = A ∪ {(u, v)}
    UNION(u,v)
return A
```

- **MAKE-SET(*x*)** – creates a new set whose only member is *x*. Since the sets are disjoint, we require that *x* not already be in some other set.
- **UNION(*x*,*y*)** – unites the dynamic sets that contain *x* and *y* into a new set that is the union of these two sets.
- **FIND-SET(*x*)** - returns a pointer to the representative of the (unique) set containing *x*.

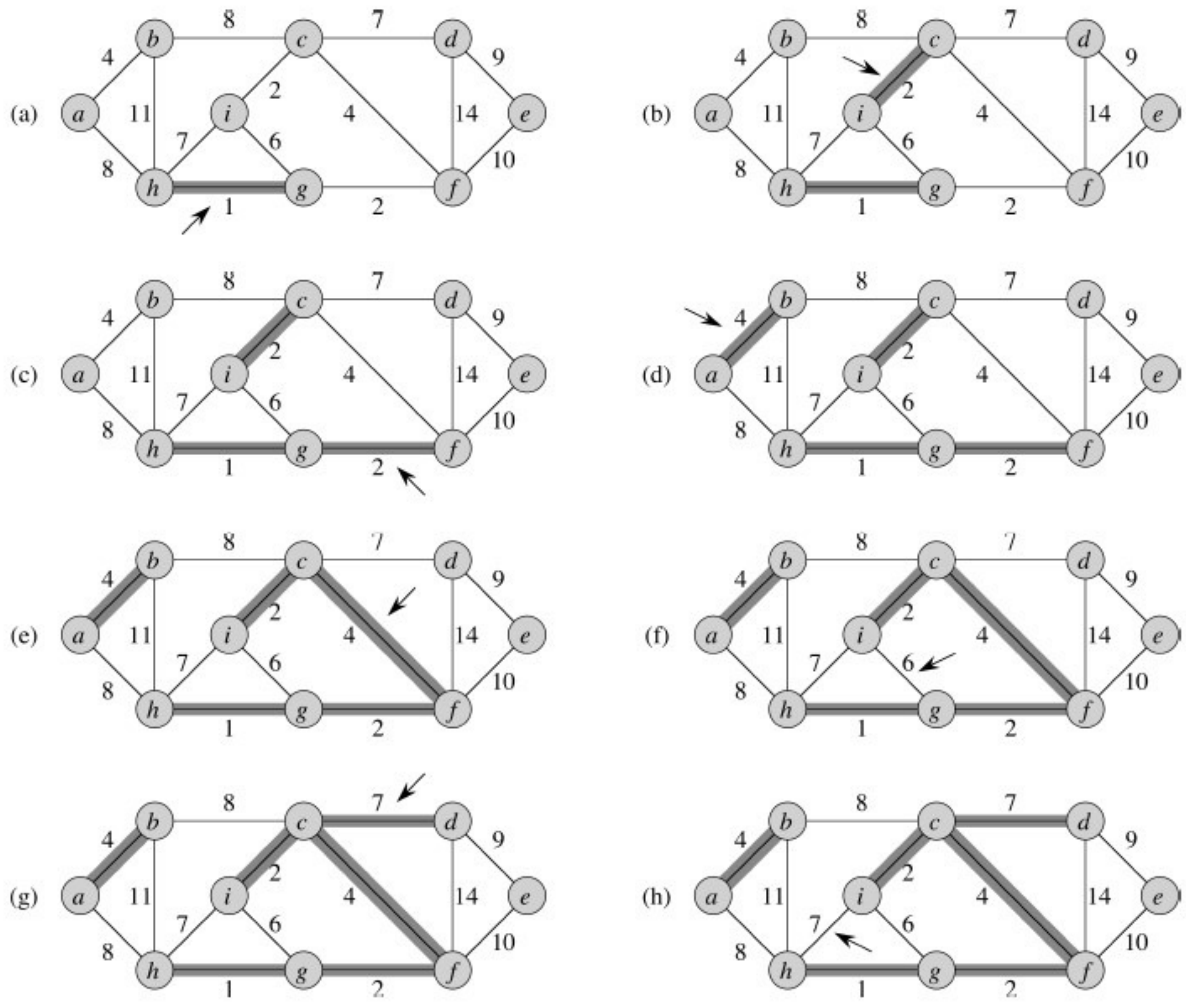
Kruskal's Algorithm



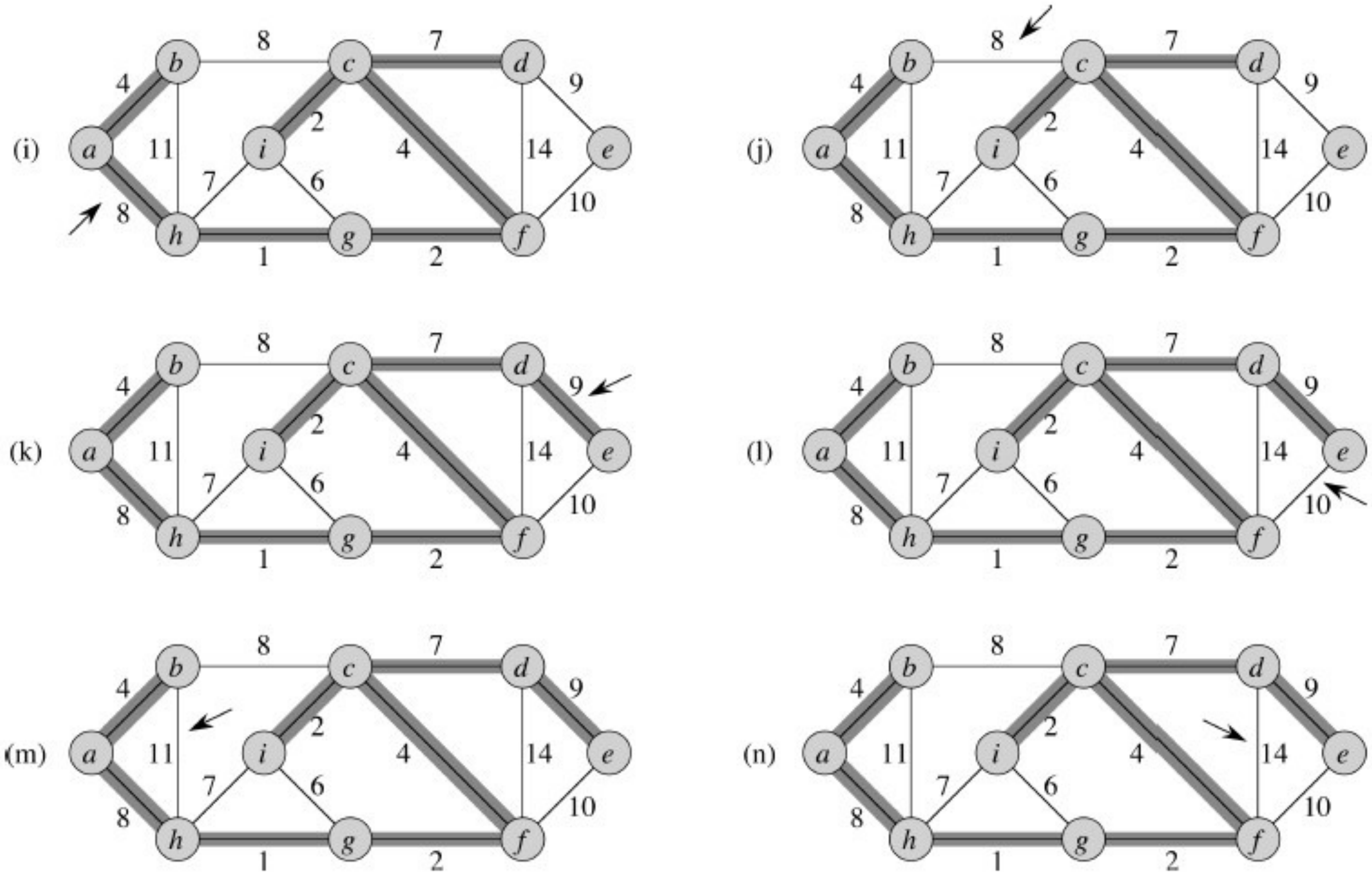
Algorithm (KRUSKAL(G,w))

```
A = ∅
foreach (vertex v ∈ G.V) do
  MAKE-SET(v)
#sort the edge of G.E into non-decreasing order by weight w
foreach ((u,v) taken from the sorted list) do
  if (FIND-SET(u)≠FIND-SET(v)) then
    A = A ∪ {(u, v)}
    UNION(u,v)
return A
```

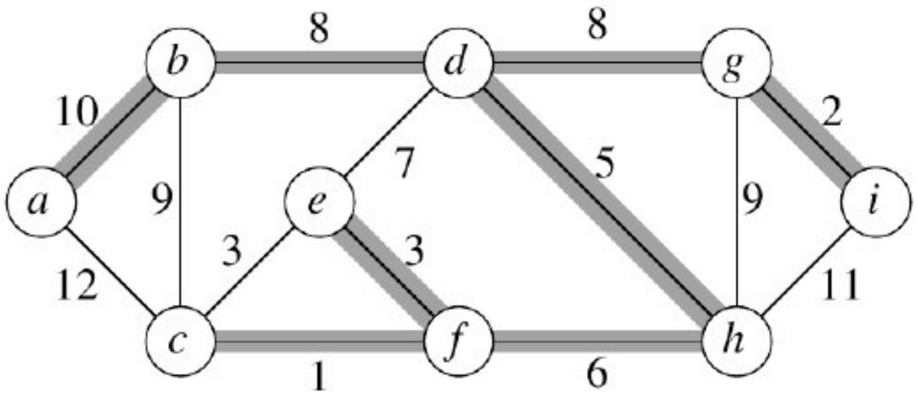
Kruskal's Algorithm



Kruskal’s Algorithm



Kruskal's Algorithm



- All edges are safe except (c, e), (e, d), (b, c), and (g, h).
- If edges (c, e) were examined before (e, f), then (c, e) would have been safe, and (e, f) would have been rejected.

Kruskal's Algorithm

Algorithm (KRUSKAL(G,w))

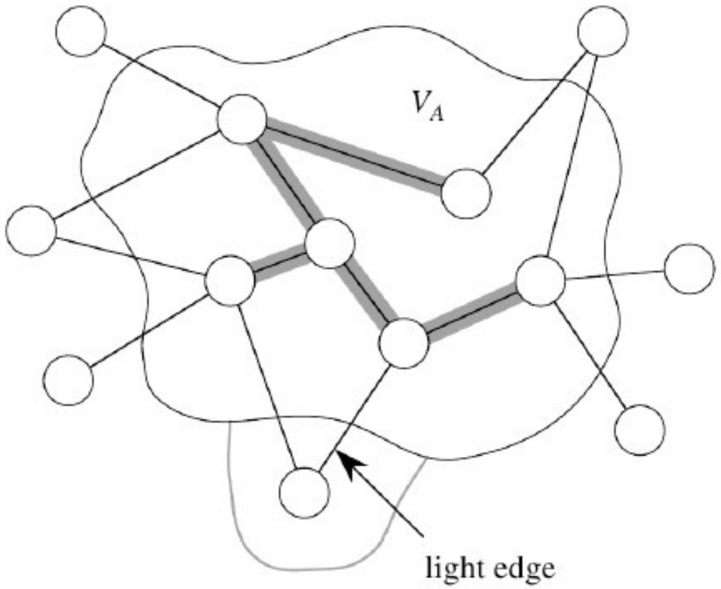
```
1  A = ∅
2  foreach (vertex v ∈ G.V) do
3    MAKE-SET(v)
4  sort the edge of G.E into non-decreasing order by weight w
5  foreach ((u,v) taken from the sorted list) do
6    if (FIND-SET(u)≠FIND-SET(v)) then
7      A = A ∪ {(u, v)}
8      UNION(u,v)
9  return A
```

- ✓ Line 1: $O(1)$
- ✓ Line 2-3: $|V|$
- ✓ Line 4: $O(E \lg E)$
- ✓ Line 5-8: $O(E)$

- ❑ The total running time is $O(V+E)+O(E \lg E)$
- ❑ G is connected → $|V|-1 \leq |E| \leq |V|^2$.
  - ❑ The total running time is $O(E)+O(E \lg E)=O(E \lg E)$
  - ❑ or $O(E \lg V)$ since $\lg E = O(2 \lg V) = O(\lg V)$.

Prim’s Algorithm

- View as a tree.
- Build one tree, and A is always a tree.
- We start from an arbitrary “root”, r.
- At each step, we find a light edge crossing a cut $(V_A, V - V_A)$, where V_A is the vertices that A is incident on and we add this edge to A.



Prim's Algorithm

How can we find the light edge quickly?

- **Use a priority queue $\mathbf{Q}$:**
  - Each object is a vertex in $V - V_A$
  - A key of v is the minimum weight of any edge (u, v), where u ∈ V_A.
  - We use EXTRACT-MIN to return the vertex v such that there exists u ∈ V_A and (u, v) is a light edge crossing $(V_A, V - V_A)$.
  - We give the v's key value ∞ if v is not adjacent to any vertices in V_A.

- **The edges of $\mathbf{A}$ will form a rooted tree with root $\mathbf{r}$.**
  - We give the root as an input to the algorithm, but it can be any vertex.
  - Each vertex knows its parent in the tree by the attribute $v.\pi$ =parent of v.
    - $v.\pi$ =NIL if $v = r$ or v has no parent.
  - As the algorithm progresses, $A = \{(v, v.\pi): v \in V - \{r\} - Q\}$.
  - When the algorithm terminates, $V_A = V \Rightarrow Q = \emptyset$, so MST is $A = \{(v, v.\pi): v \in V - \{r\}\}$.

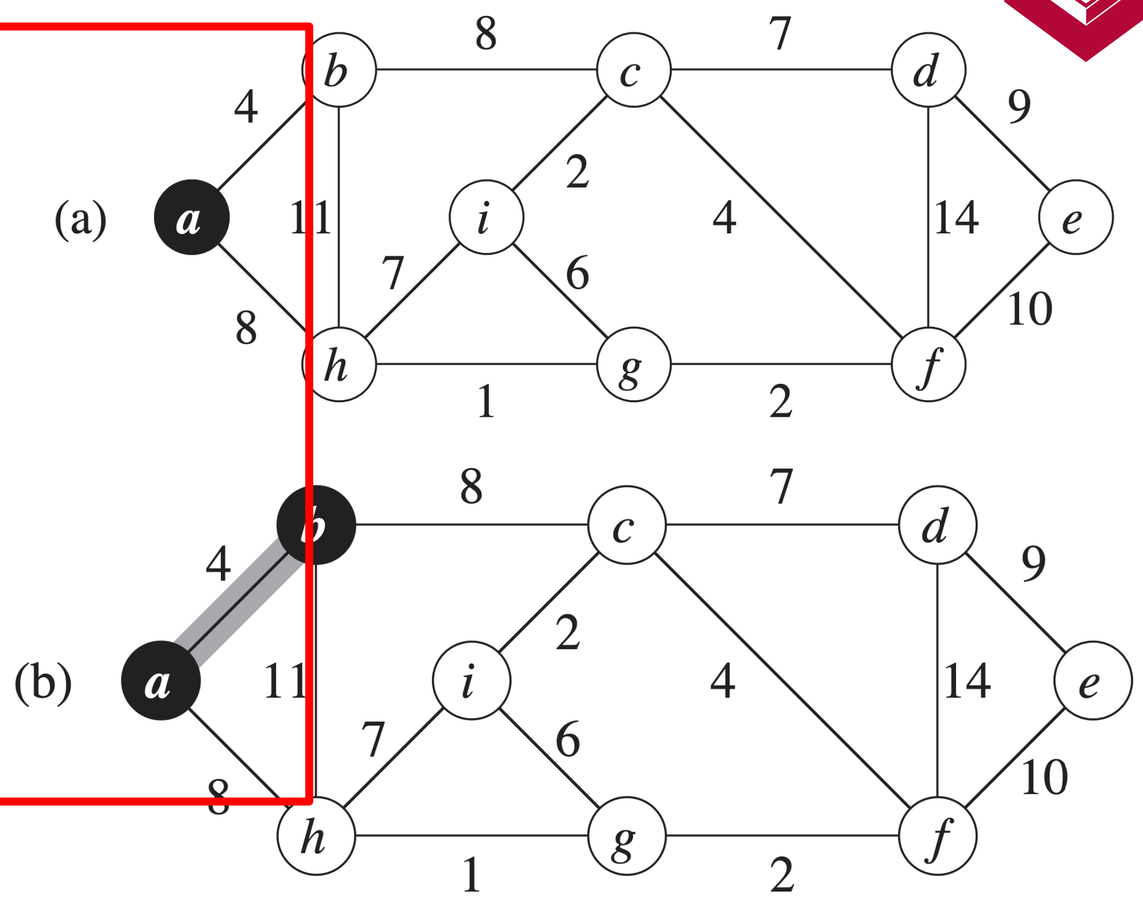
Prim's Algorithm

Algorithm (PRIM(G, w, r))

```
Q = ∅
foreach ( vertex u ∈ G.V ) do
  u.key = ∞
  u.π = NIL
  INSERT (Q, u)
DECREASE-KEY(Q, r, 0)          //r.key=0
while ( Q ≠ ∅ ) do
  u = EXTRACT-MIN( Q )
  foreach (v ∈ G.Adj[u]) do
    if (v ∈ Q and w(u,v)<v.key) then
      v.π = u
      DECREASE-KEY (Q, v, w(u,v))
```

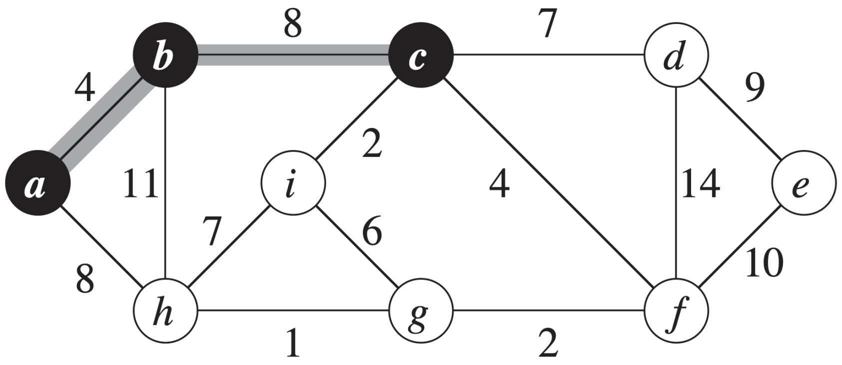
Prim’s Algorithm

```
Algorithm (PRIM( G, w, r))
  Q = ∅
 foreach ( vertex u ∈ G.V ) do
   u.key = ∞
   u.π = NIL
   INSERT (Q, u)
 DECREASE-KEY(Q, r, 0)          //r.key=0
 while ( Q ≠ ∅ ) do
   u = EXTRACT-MIN( Q )
   foreach (v ∈ G.Adj[u]) do
     if (v ∈ Q and w(u,v)<v.key) then
       v.π = u
       DECREASE-KEY (Q, v, w(u,v))
```

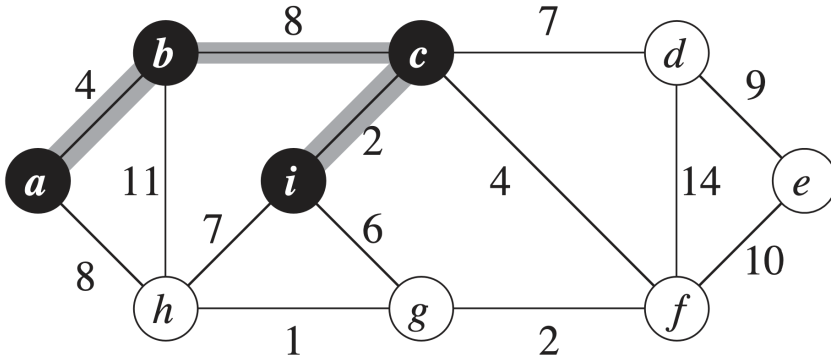


Prim’s Algorithm

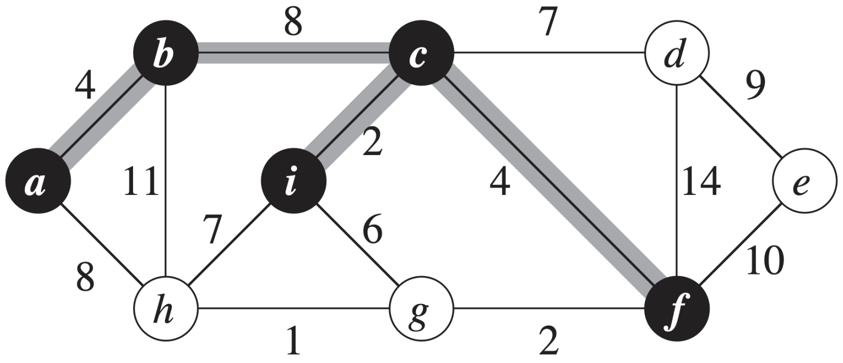
(c)



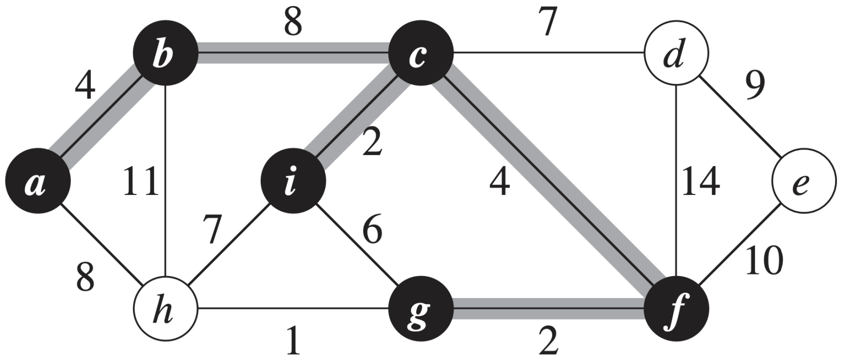
(d)



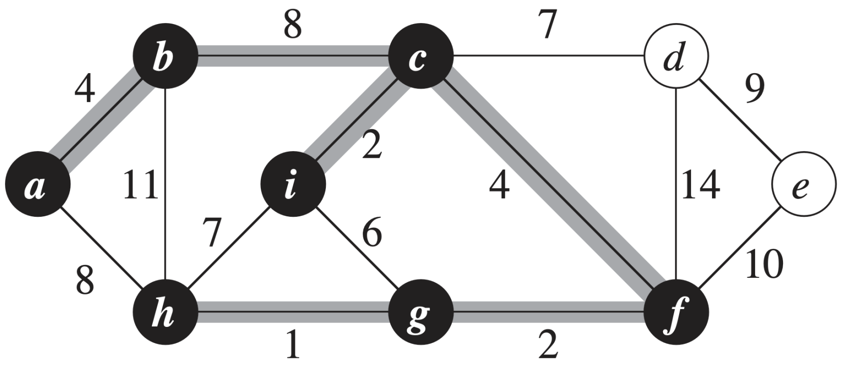
(e)

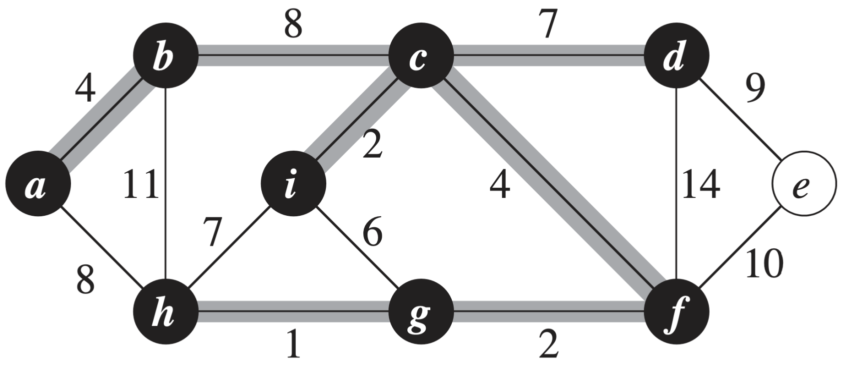


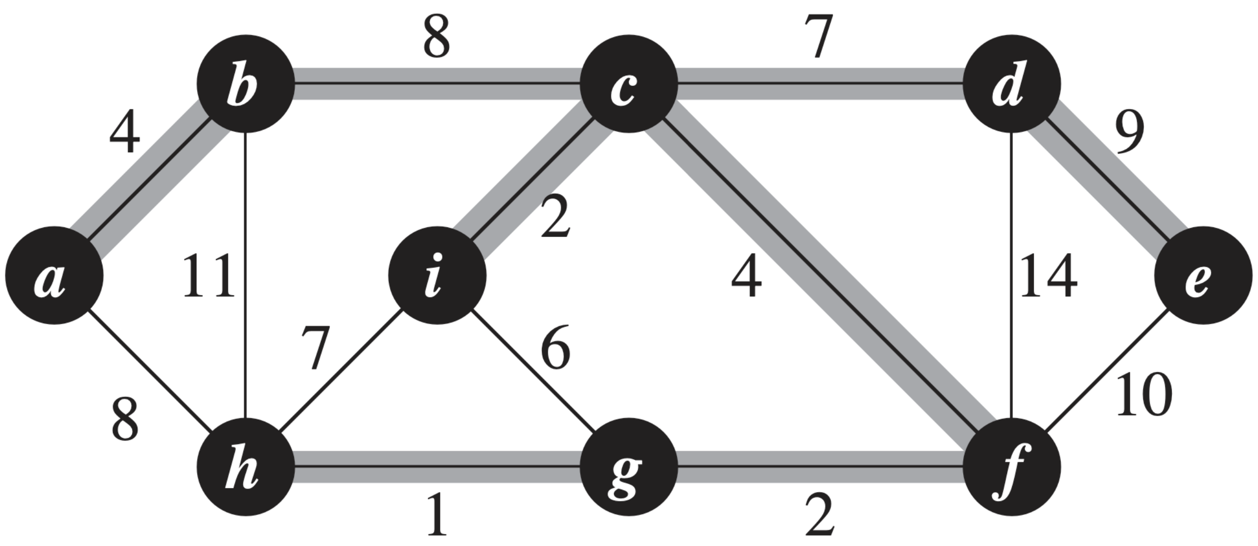
(f)



Prim’s Algorithm

(g)


(h)


(i)


Prim's Algorithm

Analysis: depends on the priority queue.

- Suppose Q is a binary heap:
  - Initialize Q and first for loop: $O(V \lg V)$
  - Decrease key of r: $O(\lg V)$
  - while loop:
    - |V| EXTRACT-MIN calls
    - $\Rightarrow O(V \lg V) \leq |E|$ DECREASE-KEY calls
    - $\Rightarrow O(E \lg V)$
  - $\Rightarrow O(E \lg V)$.
- We could do DECREASE-KEY in $O(1)$ amortized time.
  - Then $\leq |E|$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$

Algorithm (PRIM(G, w, r))

```
Q = ∅
foreach ( vertex u ∈ G.V ) do
  u.key = ∞
  u.π = NIL
  INSERT (Q, u)
DECREASE-KEY(Q, r, 0)          //r.key=0
while ( Q ≠ ∅ ) do
  u = EXTRACT-MIN( Q )
  foreach (v ∈ G.Adj[u]) do
    if (v ∈ Q and w(u,v)<v.key) then
      v.π = u
      DECREASE-KEY (Q, v, w(u,v))
```