Understanding of GENERIC-MST

- As the method proceeds, the set A is always acyclic; otherwise, a minimum spanning tree including A would contain a cycle, which is a contradiction.
- At any point in the execution, the graph $G_A=(V, A)$ is a forest, and each of the connected components of G_A is a tree.
- Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains |V| trees, one for each vertex.
- Moreover, any safe edge (u,v) for A connects distinct components of G_A, since $A \cup \{(u,v)\}$ must be acyclic.

Understanding of GENERIC-MST

- The **while** loop in lines 2 - 4 of GENERIC-MST executes |V| - 1 times because it finds one of the |V|-1 edges of a minimum spanning tree in each iteration.
- Initially, when $A = \emptyset$, there are |V| trees in G_A, and each iteration reduces that number of trees by 1.
- When the forest contains only a single tree, the method terminates.

Corollary 23.2

- Let G=(V,E) be a connected, undirected graph with a real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G.
- Let $C=(V_C,E_C)$ be a connected component (tree) in the forest $G_A=(V,A)$.
- Cut(V_C, $V-V_C$)
- If (u,v) is a light edge connecting C to some other component in G_A, then (u,v) is safe for A

Corollary 23.2 (Proof)

- The cut (V_C, $V-V_C$) respects A and (u,v) is a light edge for this cut.
- Thus, (u,v) is safe for A by Theorem 23.1.

Theorem 23.1

Let G=(V,E) be a connected undirected graph with a real-valued weight function w defined on E.
Let A be a subset of E that is included in some minimum spanning tree for G.
Let (S,V-S) be any cut of G that respects A and let (u,v) be a light edge crossing (S,V-S).
Then, the edge (u,v) is safe for A

Kruskal's and Prim's Algorithms

- They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.
- In Kruskal's algorithm,
  - The set A is a forest whose vertices are all those of the given graph.
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
- In Prim's algorithm,
  - The set A forms a single tree.
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

Kruskal's Algorithm

- A greedy algorithm since at each step it adds to the forest an edge of least possible weight.
- Find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight.
- Let two trees C_1 and C_2 are connected by (u,v).
- Since (u,v) must be a light edge connecting C_1 to some other tree, Corollary 23.2 implies that (u,v) is a safe edge for C_1.

Implementation of Kruskal's Algorithm

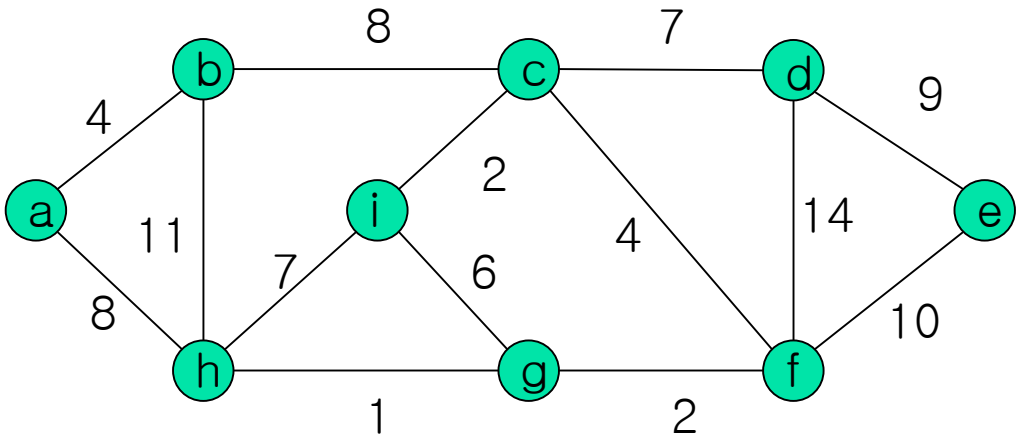
- It uses a disjoint-set data structure to maintain several disjoint sets of elements.
- Each set contains the vertices in one tree of the current forest.
- The operation FIND-SET(u) returns a representative element from the set that contains u.
- Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).
- To combine trees, Kruskal's algorithm calls the UNION procedure.

Kruskal's Algorithm

MST-KRUSKAL(G,w)

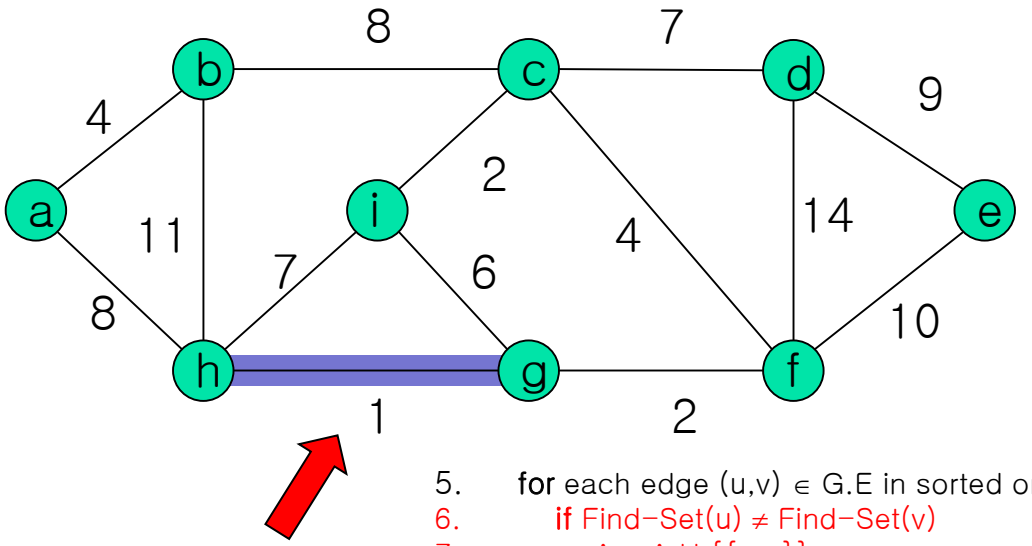
1. $A = \varnothing$
2. **for** each $v \in G.V$
3. Make-Set(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. **for** each edge $(u,v) \in G.E$ in sorted order
6. **if** Find-Set(u) $\neq$ Find-Set(v)
7. $A = A \cup \{\{u,v\}\}$
8. Union(u,v)
9. **return** A

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

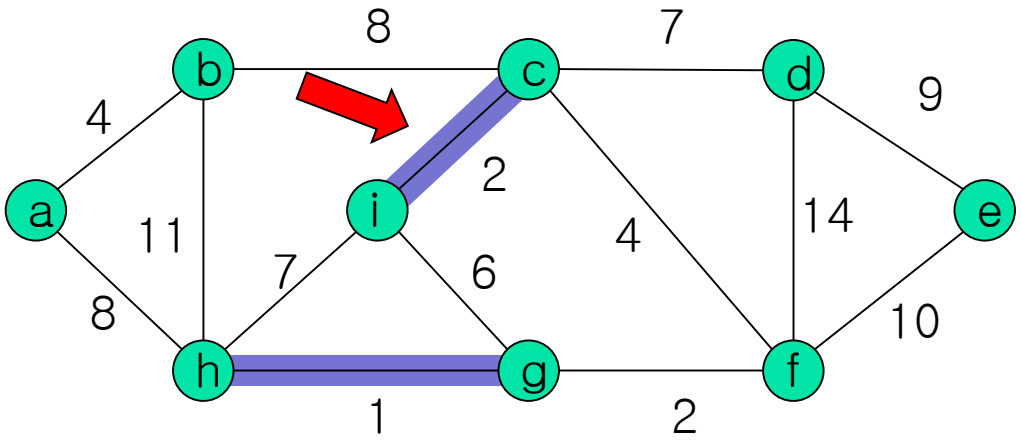
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

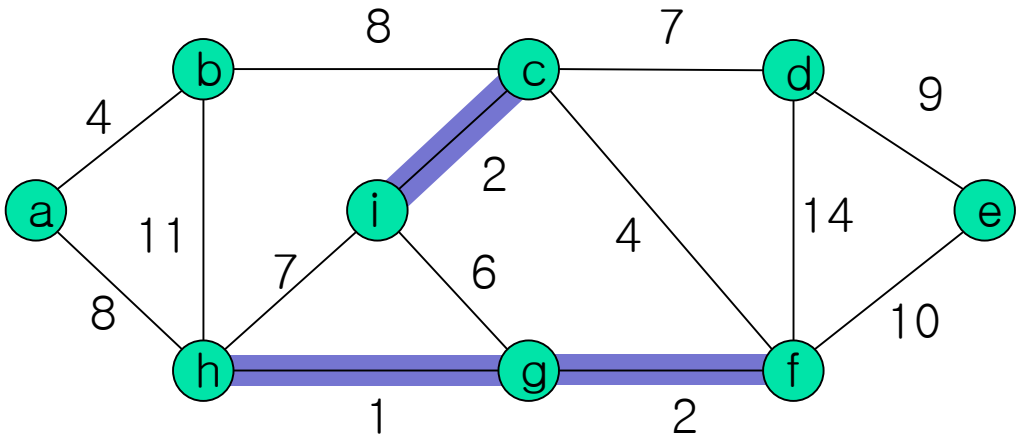
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm



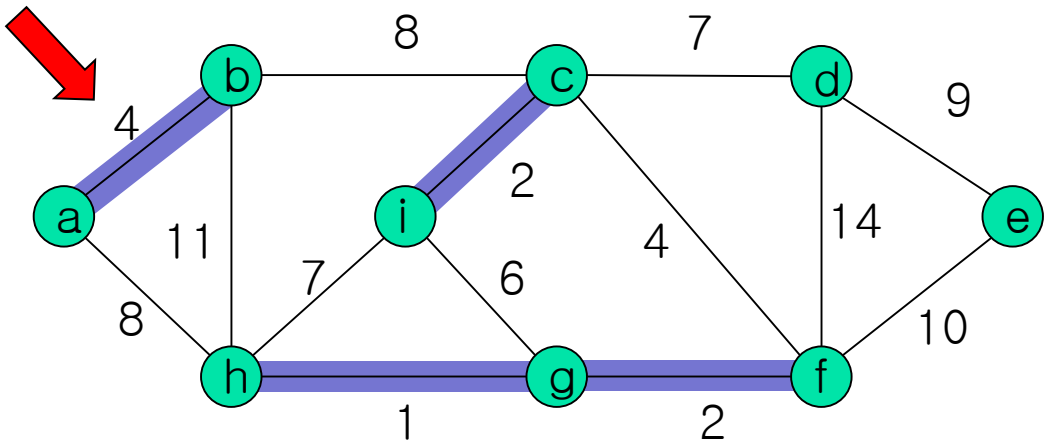

```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

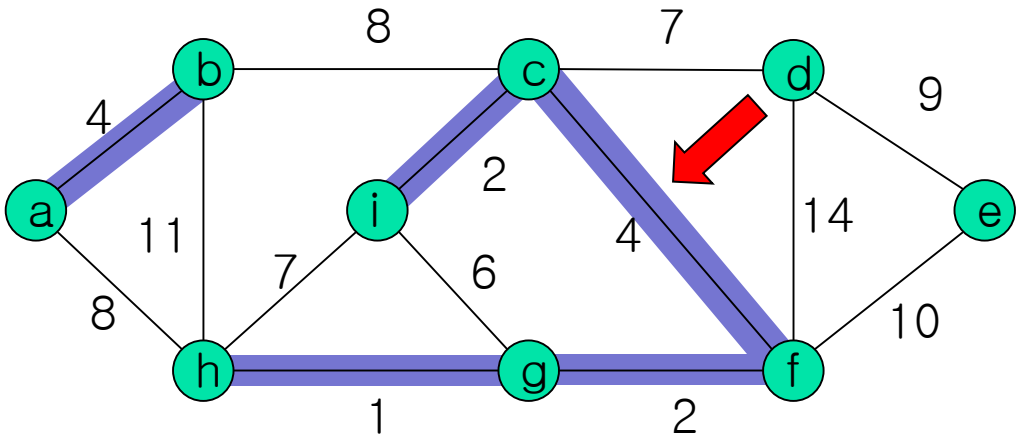
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

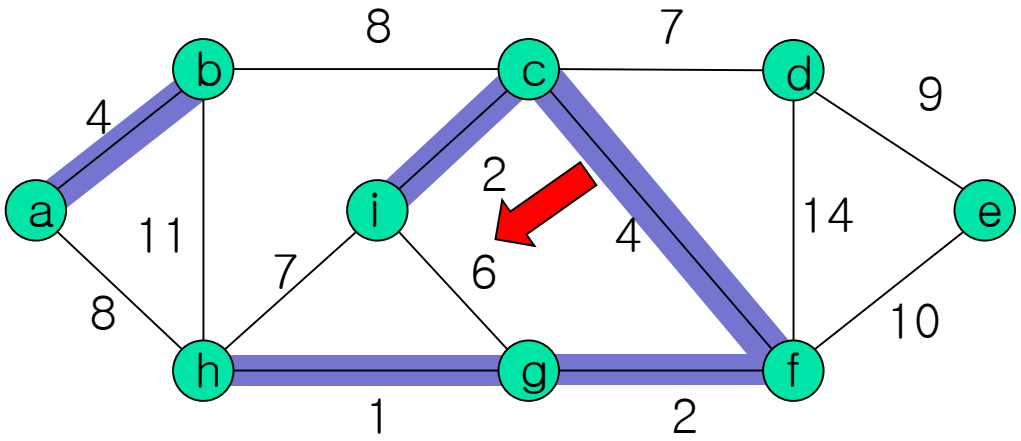
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

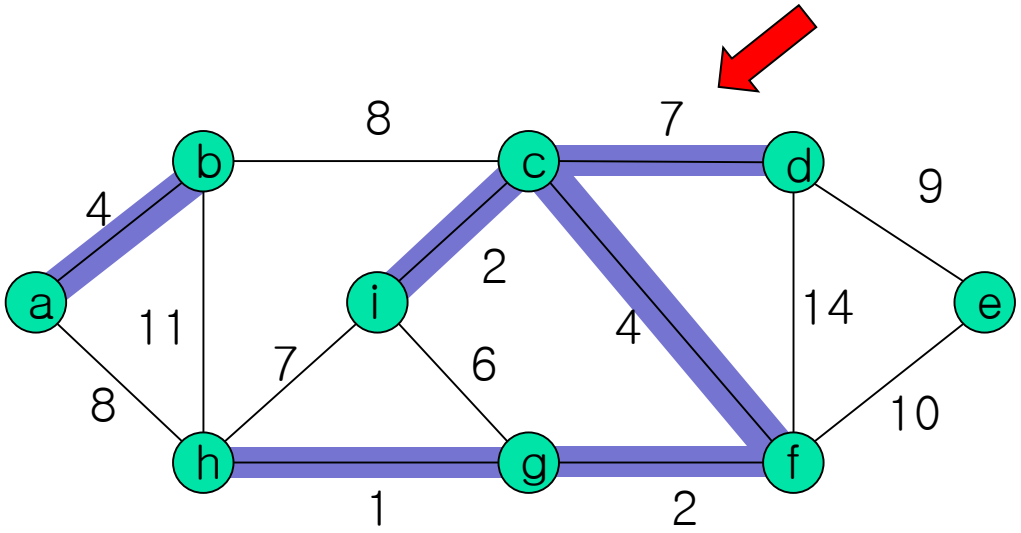
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

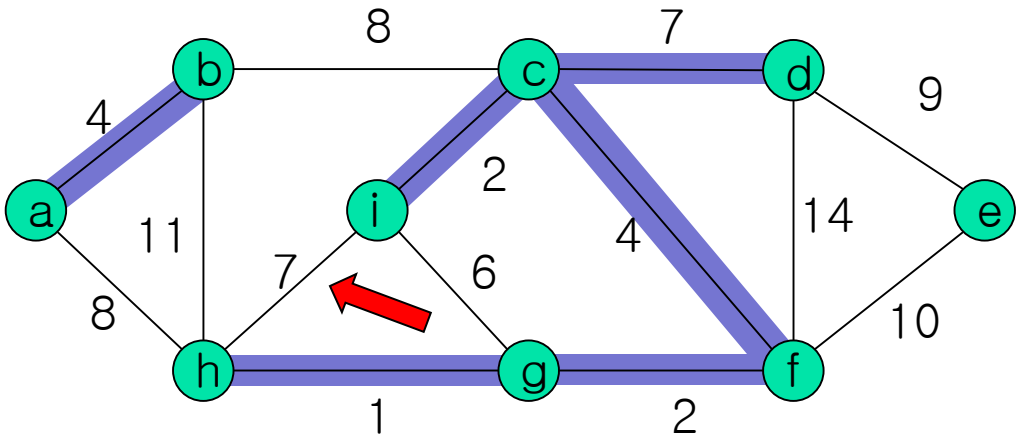
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

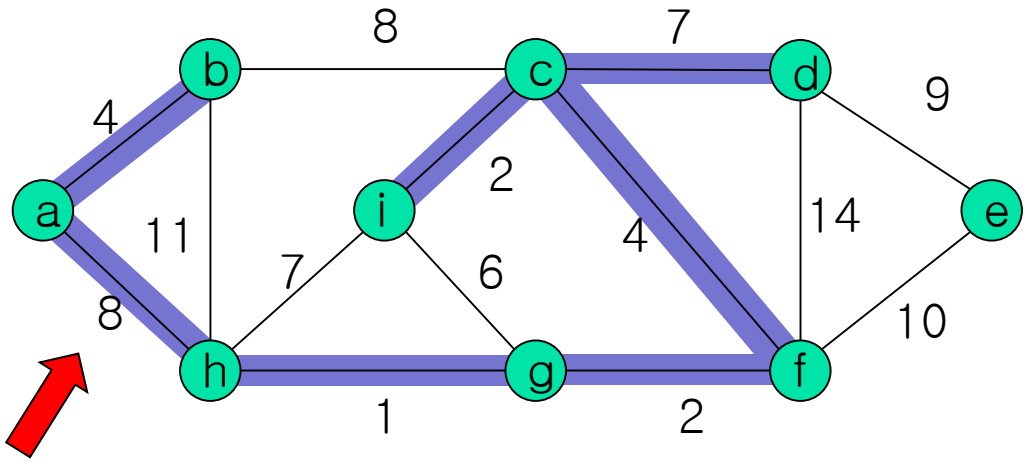
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm



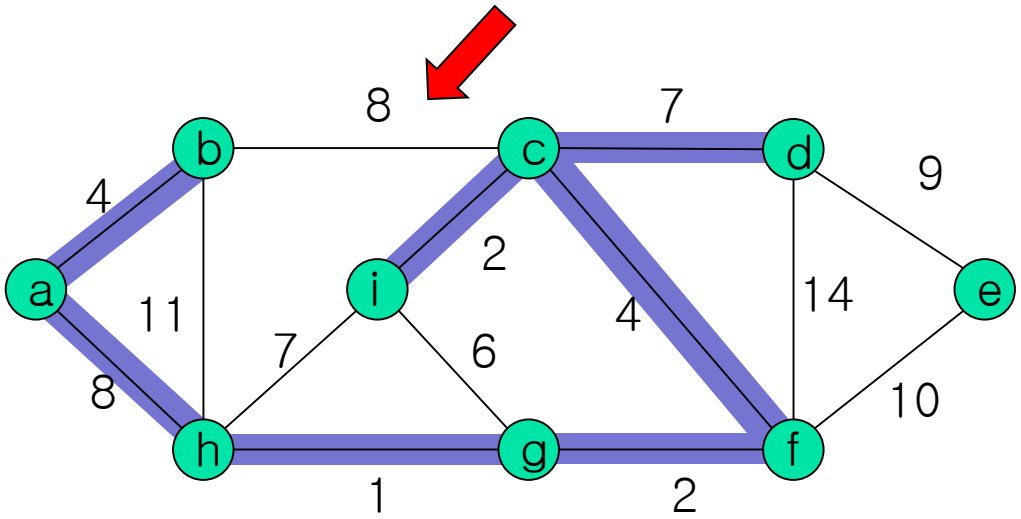

```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

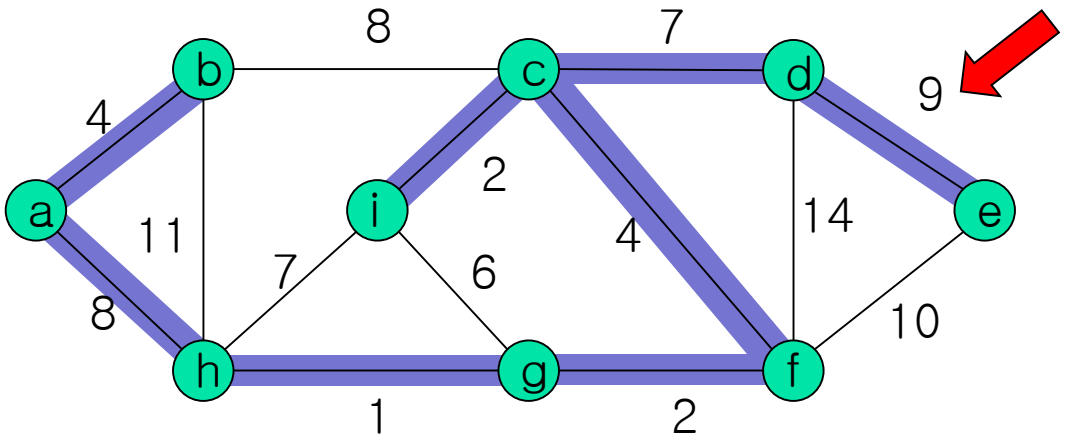
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

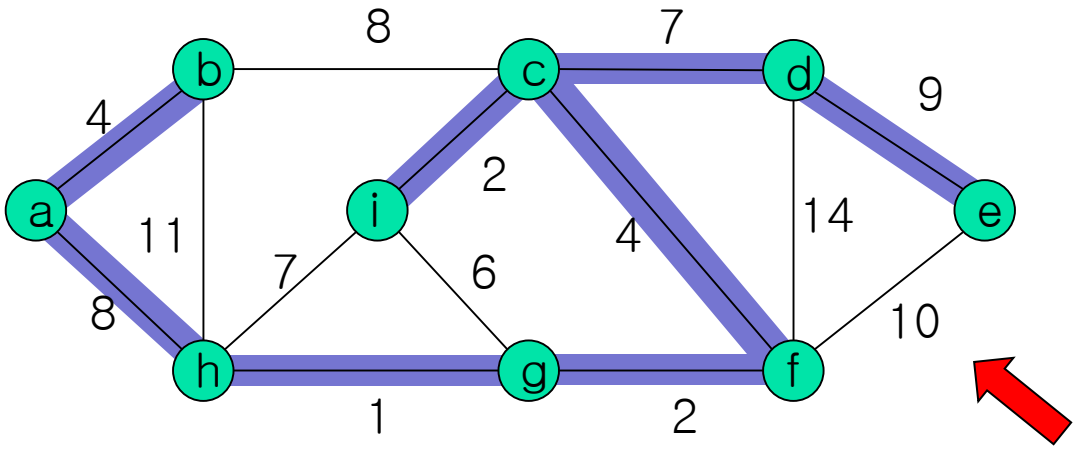
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

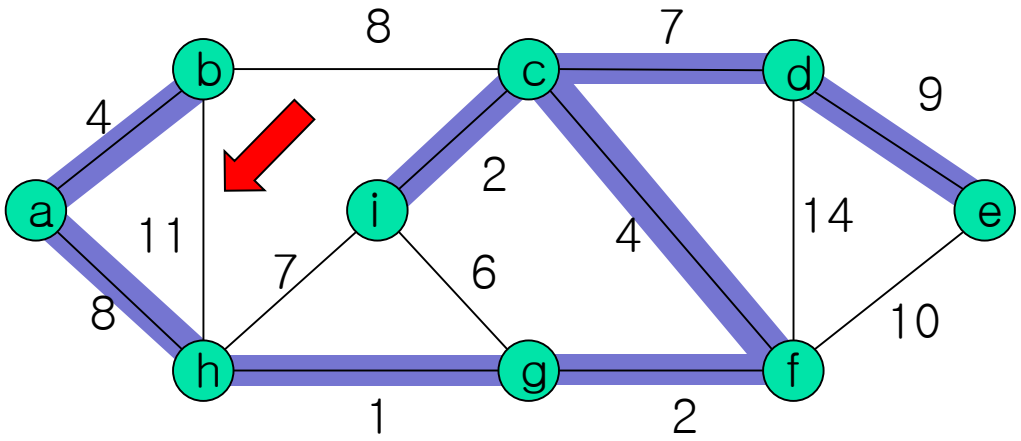
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

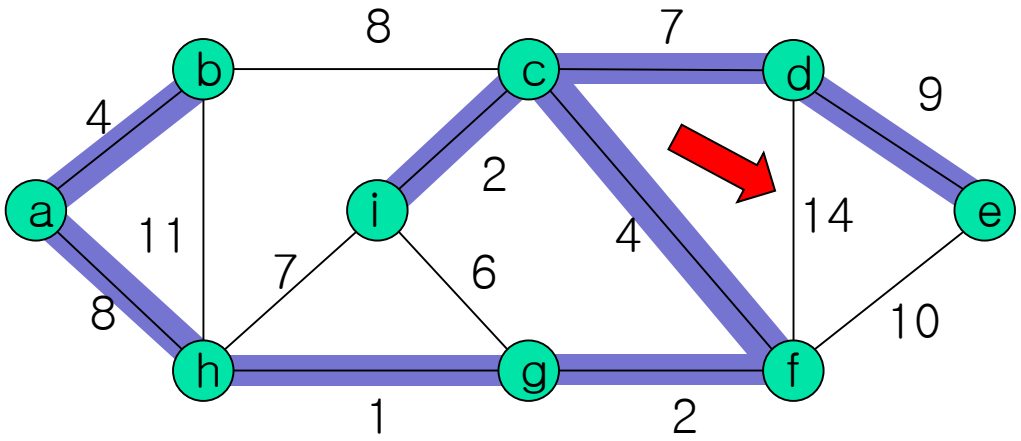
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

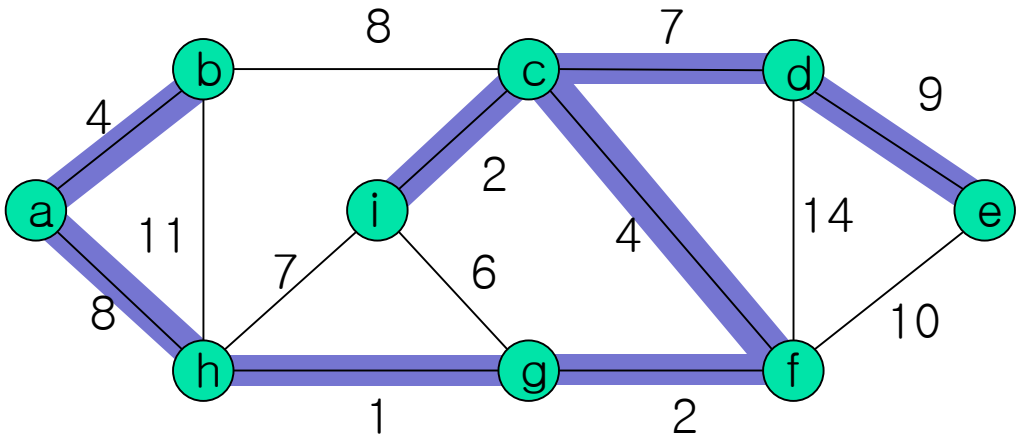
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
9.    return A
```

Kruskal's Algorithm

MST-KRUSKAL(G,w)

1. $A = \varnothing$
2. **for** each $v \in G.V$
3. Make-Set(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. **for** each edge $(u,v) \in G.E$ in sorted order
6. **if** Find-Set(u) $\neq$ Find-Set(v)
7. $A = A \cup \{\{u,v\}\}$
8. Union(u,v)
9. **return** A

O(V) Make-Set() calls

O(E lg E)

O(E) Find-Set() calls

O(V) Union() calls

Running Time of Kruskal's Algorithm

- Use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics (Section 21.3).
- Sorting the edges in line 4 is $O(|E| \lg |E|)$.
- The disjoint-set operations takes $O((|V|+|E|)\,\alpha(|V|))$ time, where α is the very slowly growing function (Section 21.4).
  - The **for** loop (lines 2–3) performs $|V|$ MAKE-SET operations.
  - The **for** loop (lines 5–8) performs $O(|E|)$ FIND-SET and UNION operations.
- Since G is connected, we have $|E| \geq |V|-1$, the disjoint-set operations take $O(|E|\alpha(|V|))$ time.
- Moreover, since $\alpha(|V|) = O(\lg |V|) = O(\lg |E|)$, Kruskal's algorithm takes $O(|E| \lg |E|)$ time.
- Observing that $|E| < |V|^2 \Rightarrow \lg |E| = O(\lg V)$, the total running time of Kruskal's algorithm becomes $O(E \lg V)$.

Running Time of Kruskal's Algorithm

- In a nut shell,
  - Sort edges: O(|E| lg |E|)
  - Disjoint-set operations
    - O(|V|+|E|) operations ⇒ O((|V|+|E|) α(|V|)) time
    - |E|≥|V|-1 ⇒ O(E| α(|V|)) time
    - Since α(n) can be upper bounded by the height of the tree, α(|V|)=O(lg |V|)=O(lg |E|).
- Thus, the total running time of Kruskal's algorithm is is O(|E| lg |E|)
- By observing that |E| < $|V|^2$ ⇒ lg |E| = O(lg |V|), it becomes O(|E| lg |V|).

O(|V|) Make-Set() calls
O(|E|) Find-Set() calls
O(|V|) Union() calls

Prim's Algorithm

- Special case of the generic minimum-spanning-tree method.
- A greedy algorithm since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.
- The edges in the set A always form a single tree.
- Each step adds to the tree A a light edge that connects A to an isolated vertex – one on which no edge of A is incident.
- By Corollary 23.2, this rule adds only edges that are safe for A

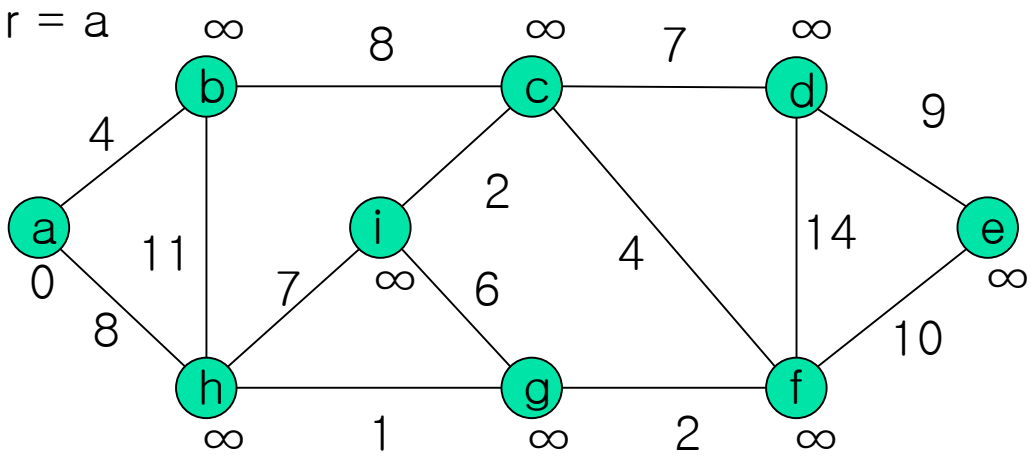
Implementation of Prim's algorithm

- Input is a connected Graph G and the root r of the minimum spanning tree.
- During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.
- For each v, v.key is the minimum weight of any edge connecting v to a vertex in the tree.
- By convention, v.key = ∞ if there is no such edge.
- The attribute $v.\pi$ names the parent of v in the tree.
- The algorithm maintains the set A from Generic-MST as $A=\{(v, v.\pi):v\in V-\{r\}-Q\}$.
- It terminates when the min-priority queue Q is empty.

Prim's Algorithm

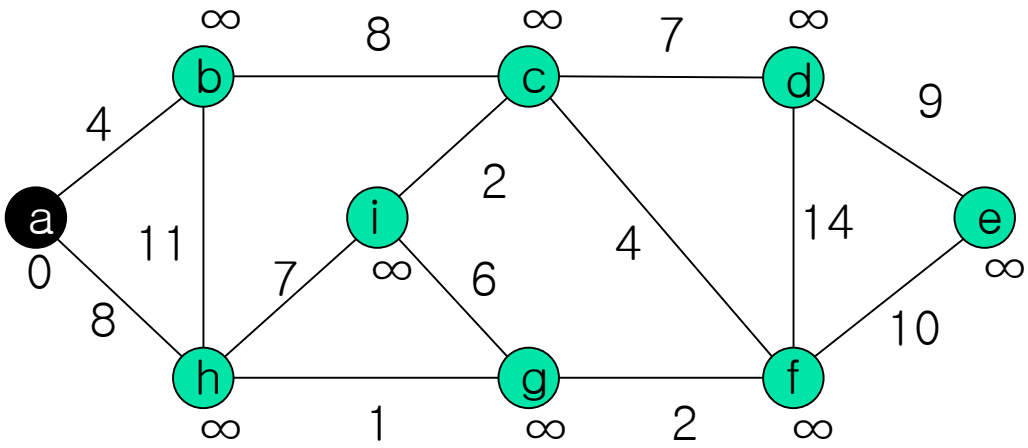
```
MST-PRIM(G, w, r)
1.  for each u ∈ G.V
2.      u.key = ∞
3.      u.π=NIL
4.  r.key = 0
5.  Q = G.V
6.  while Q ≠ Ø
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



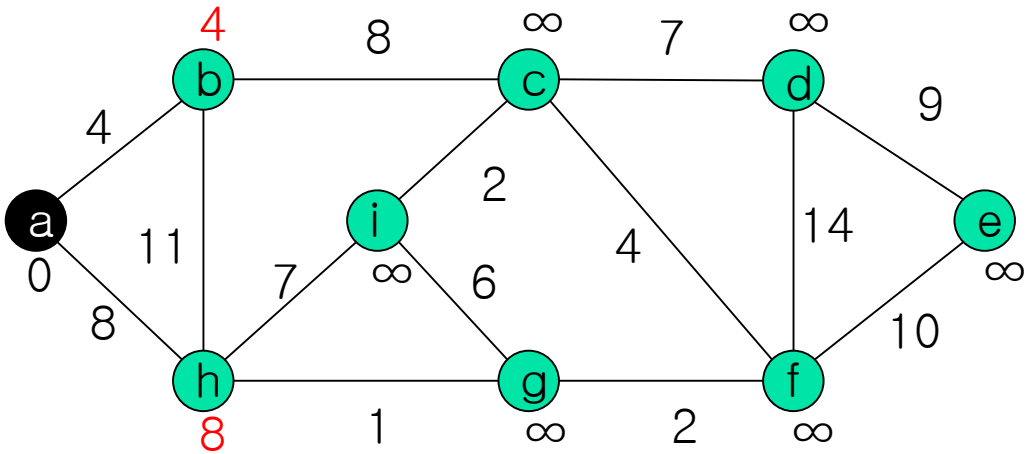

```
1.   for each u ∈ G.V
2.       u.key = ∞
3.       u.π=NIL
4.   r.key = 0
5.   Q = G.V
```

Prim's Algorithm



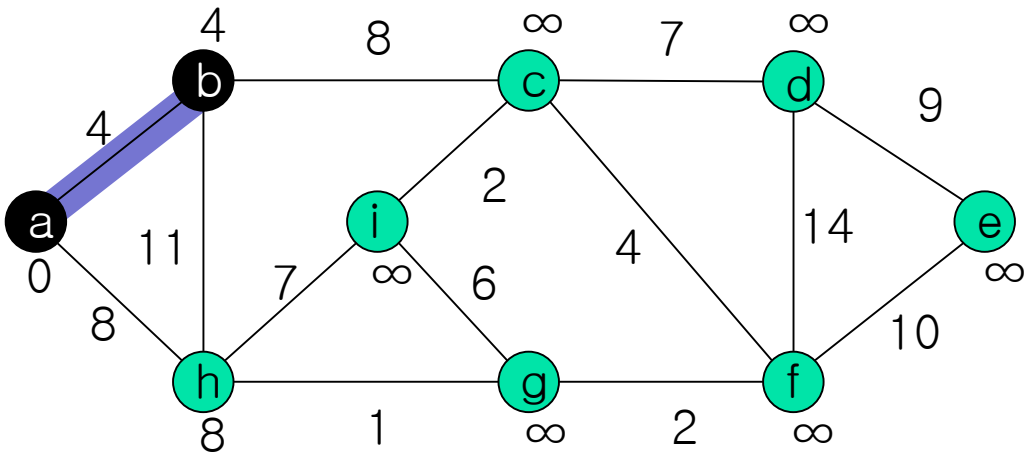

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



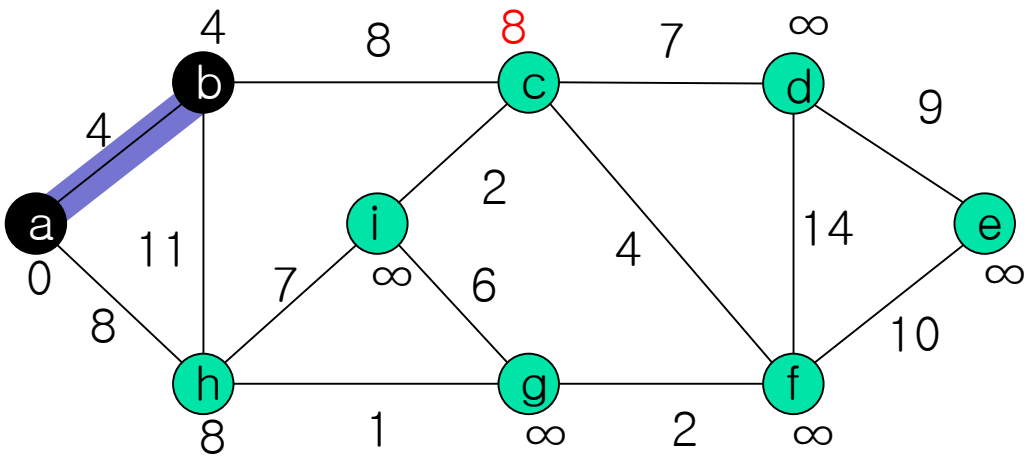

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



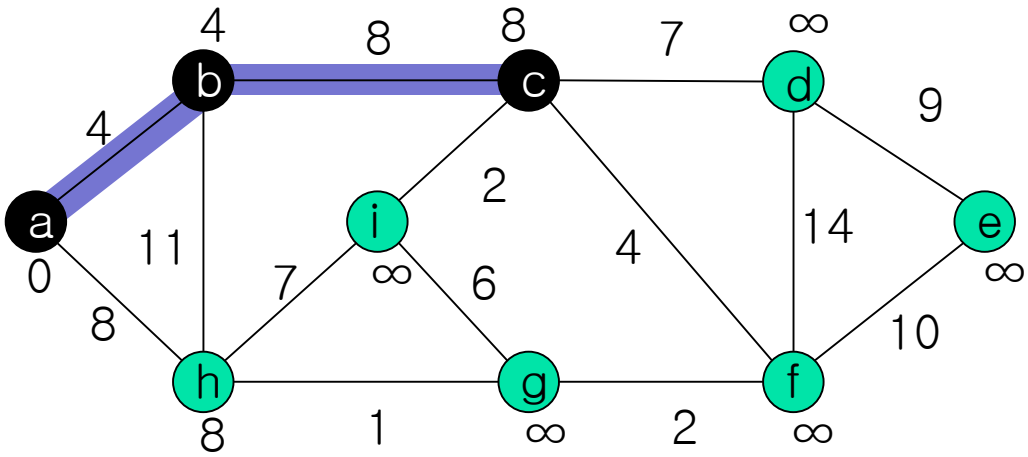

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



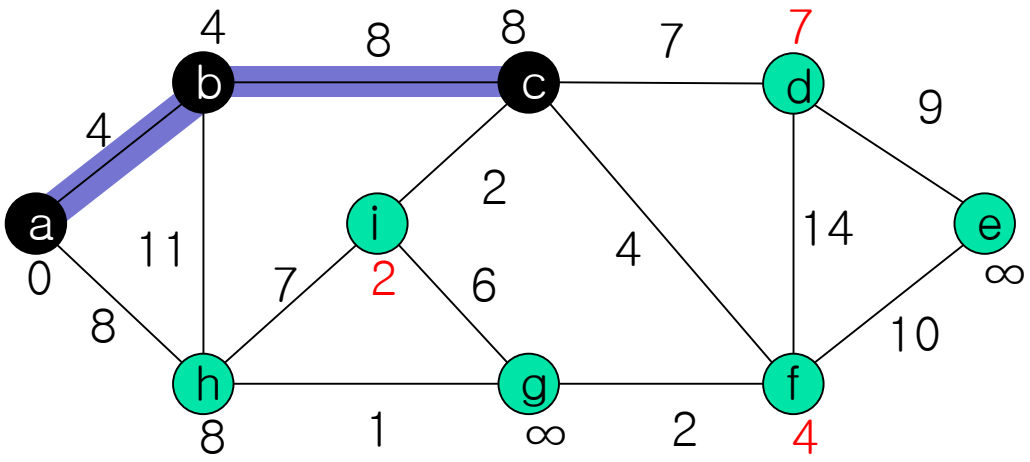

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



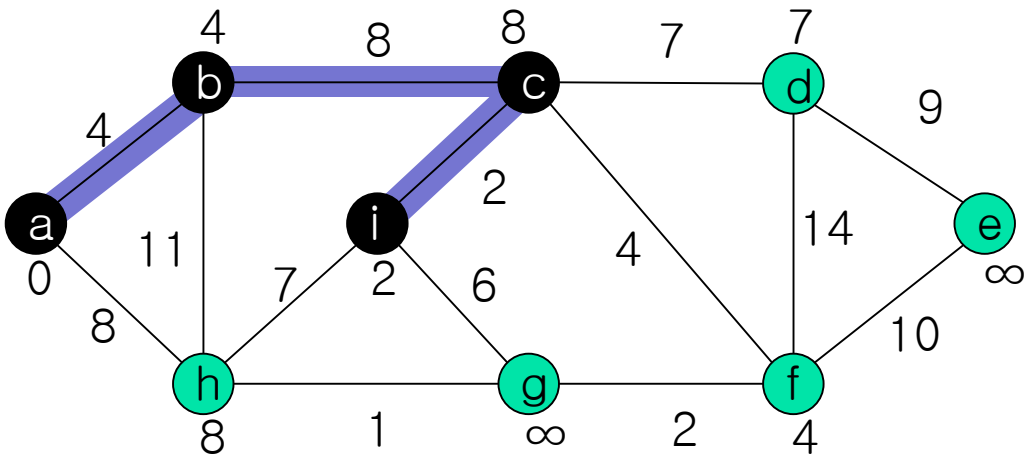

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



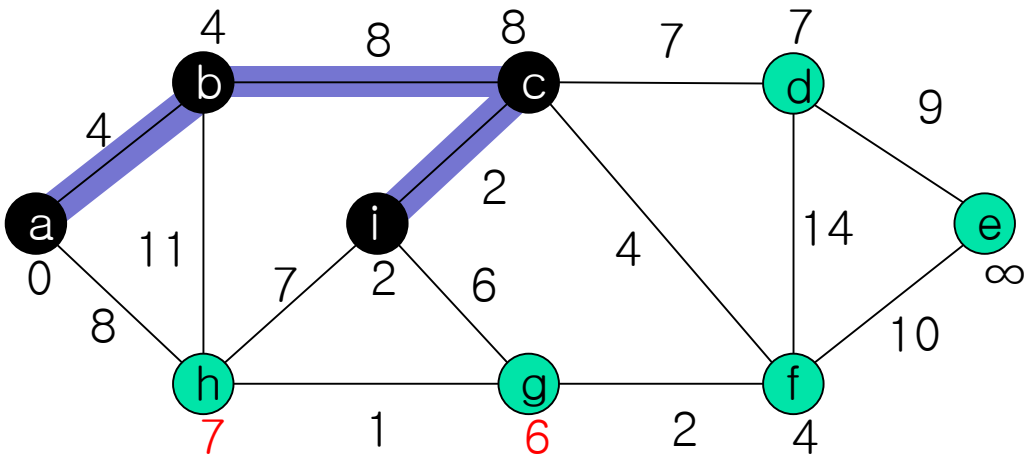

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



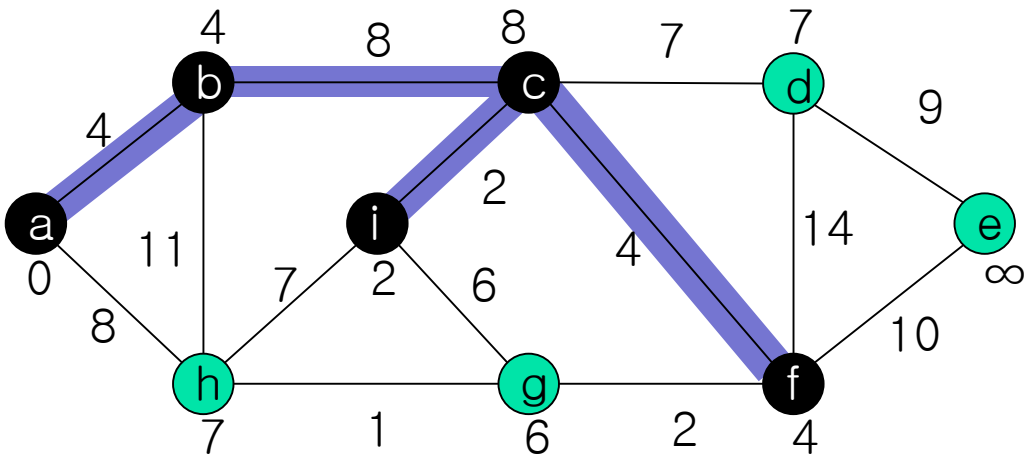

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



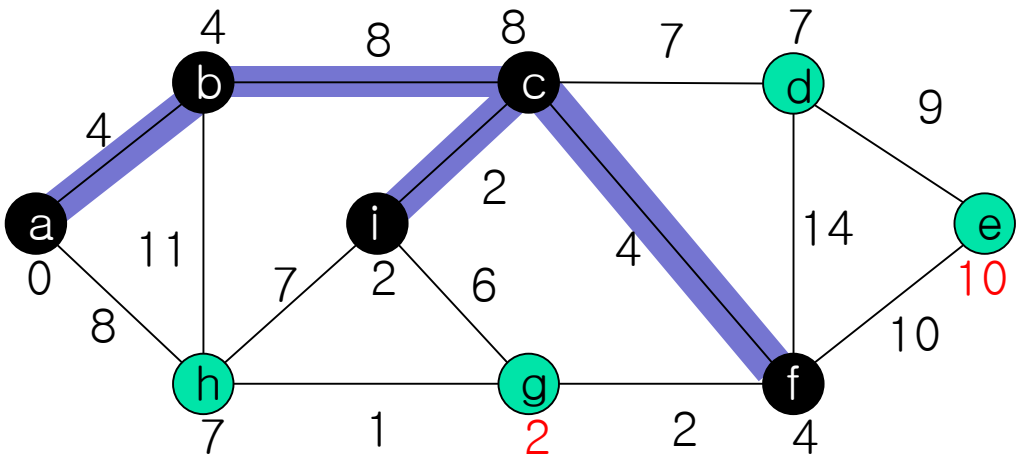

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



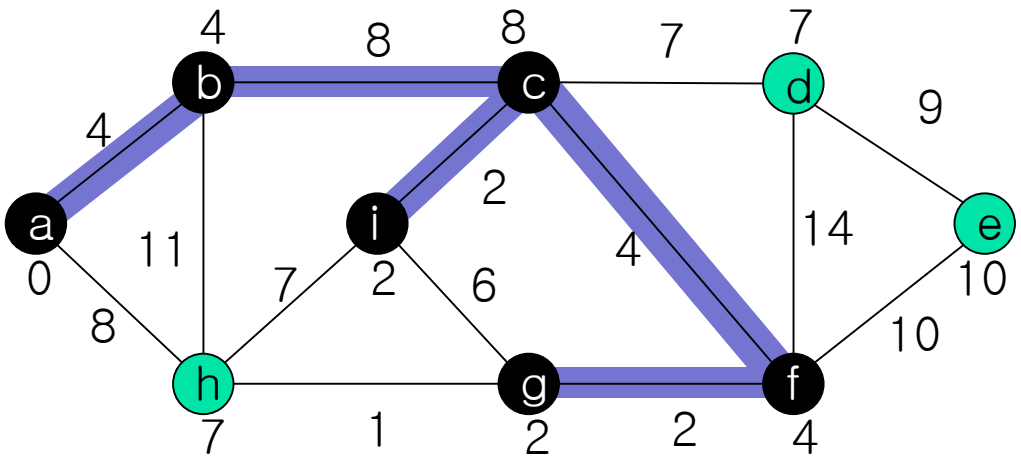

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



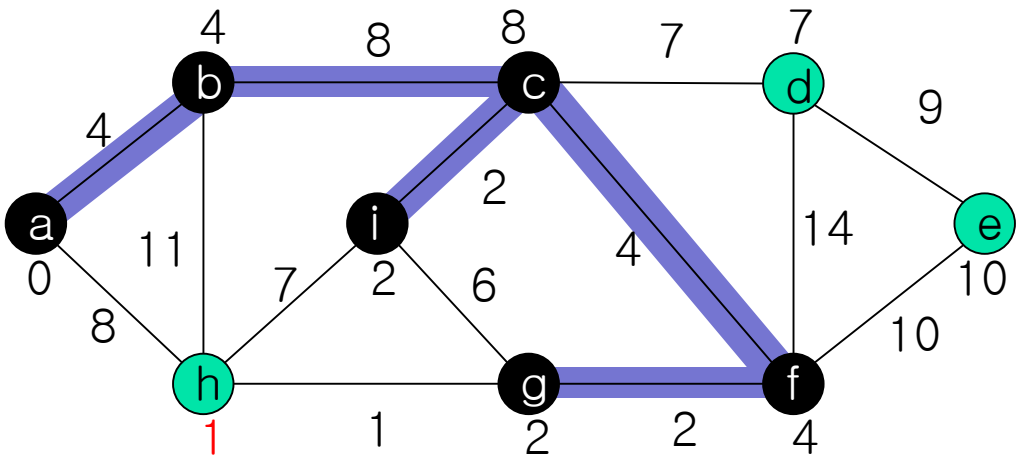

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



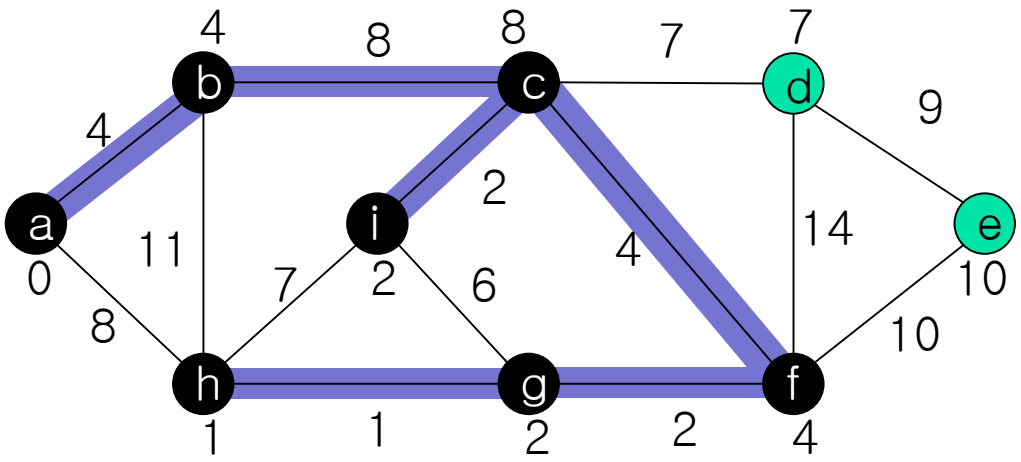

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



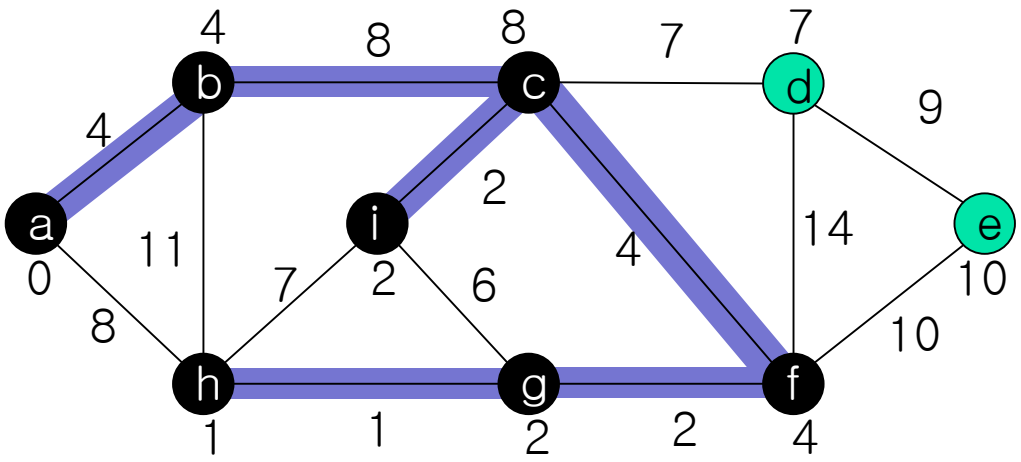

```
7.      u = Extract-Min(Q)
8.          for each v ∈ G.Adj[u]
9.              if v ∈ Q and w(u,v) < v.key
10.                 v.π = u
11.                 v.key = w(u,v)
```

Prim's Algorithm



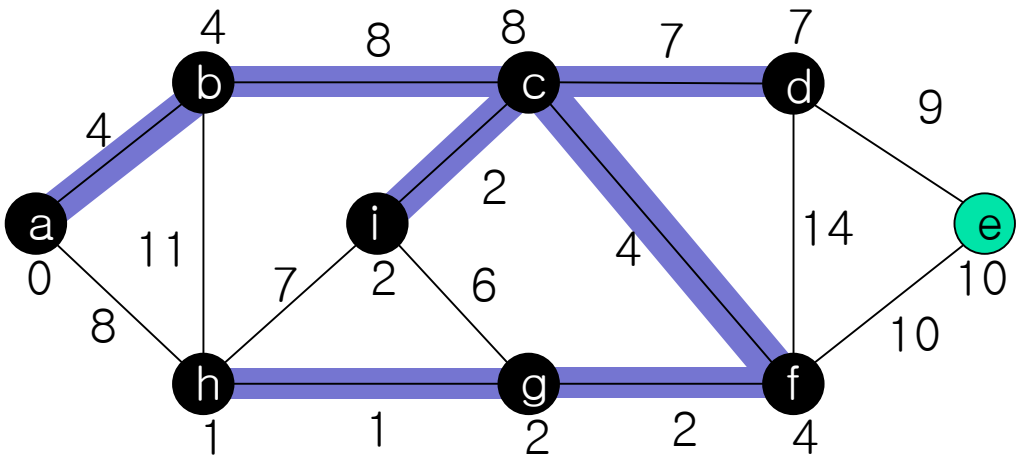

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



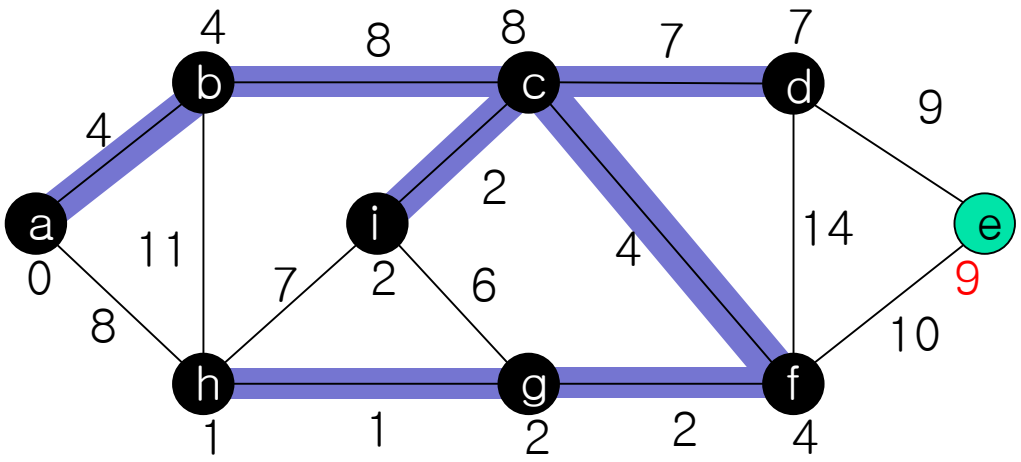

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



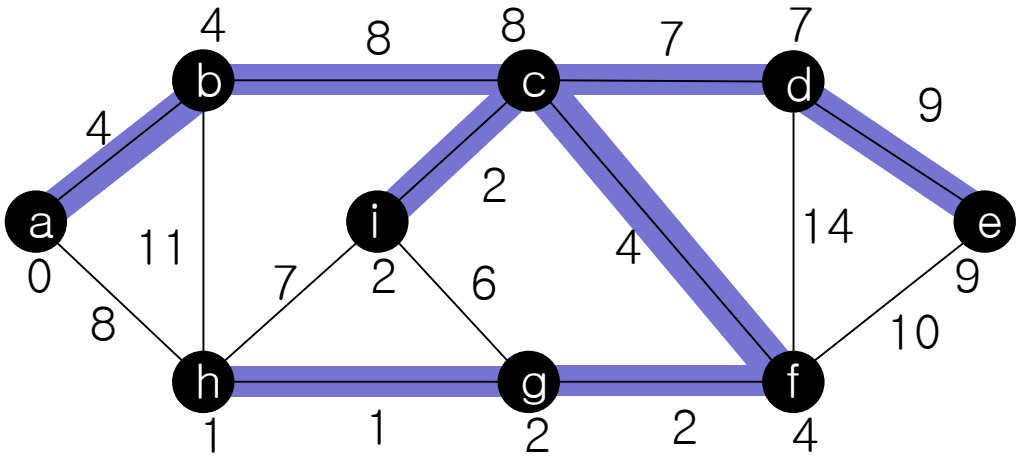

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



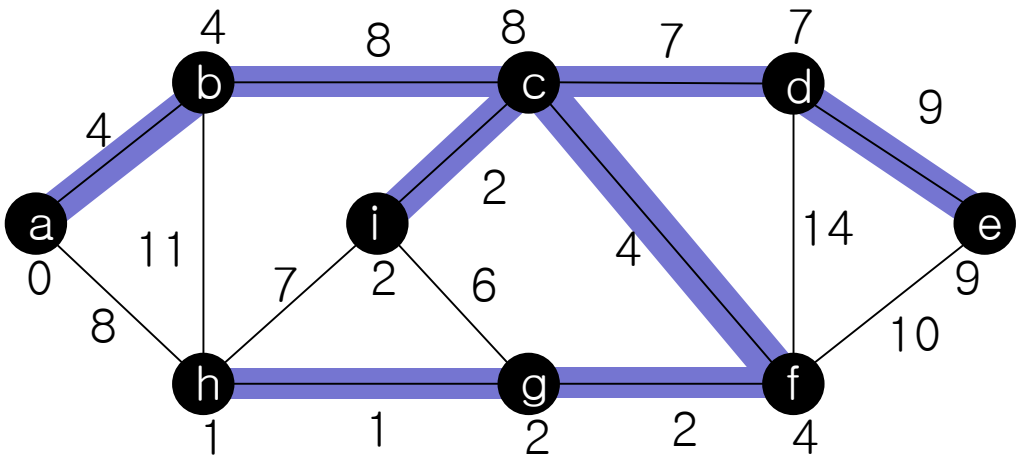

```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm




```
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π = u
11.             v.key = w(u,v)
```

Prim's Algorithm



6. while Q ≠ ∅
7. u = Extract-Min(Q)

Running Time of Prim's Algorithm

- The running time of Prim's algorithm depends on how we implement the min-priority queue Q.
- If we implement Q as a binary min-heap,
  - EXTRACT-MIN takes O(lg |V|) time.
  - DECREASE-KEY takes O(lg |V|) time.
- If we implement Q as a simple array,
  - EXTRACT-MIN takes O(|V|) time.
  - DECREASE-KEY O(1) time.
- If we implement Q as a Fibonacci heap,
  - EXTRACT-MIN takes O(lg V|) amortized time.
  - DECREASE-KEY O(1) amortized time.

Prim's Algorithm

- Total time using a binary heap is O(V lg V + E lg V) = O(E lg V).

```
MST-PRIM(G, w, r)
1.   for each u ∈ G.V
2.       u.key = ∞
3.       u.π=NIL
4.   r.key = 0
5.   Q = G.V
6.   while Q ≠ Ø
7.       u = Extract-Min(Q)
8.       for each v ∈ G.Adj[u]
9.           if v ∈ Q and w(u,v) < v.key
10.              v.π = u
11.              v.key = w(u,v)
```

O(|V|)

Q: Implement as a binary min-heap

O(|V| lg |V|)

O(|E| lg |V|)
DECREASE-KEY O(E) times

Prim's Algorithm

- Total time using a simple array is $O(E + V^2)$.

```
MST-PRIM(G, w, r)
 1.  for each u ∈ G.V
 2.      u.key = ∞
 3.      u.π=NIL
 4.  r.key = 0
 5.  Q = G.V
 6.  while Q ≠ Ø
 7.      u = Extract-Min(Q)
 8.      for each v ∈ G.Adj[u]
 9.          if v ∈ Q and w(u,v) < v.key
10.              v.π = u
11.              v.key = w(u,v)
```

O(|V|)

Q: Implement as an array

$O(|V|^2)$

O(|E|)

Prim's Algorithm

- Total time using a Fibonacci Heap is O(E + V lg V).

```
MST-PRIM(G, w, r)
1.   for each u ∈ G.V
2.       u.key = ∞
3.       u.π=NIL
4.   r.key = 0
5.   Q = G.V
6.   while Q ≠ Ø
7.       u = Extract-Min(Q)
8.       for each v ∈ G.Adj[u]
9.           if v ∈ Q and w(u,v) < v.key
10.              v.π = u
11.              v.key = w(u,v)
```

O(|V|)

Q: Implement as a Fibonacci mean-heap

O(|V| lg |V|)

O(|E|)

Any Question?

Introduction to Algorithms (Chapter 24: Single-source Shortest Paths)

Kyuseok Shim

Electrical and Computer Engineering
Seoul National University

Single-source Shortest-Paths Problem

- Given a weighted directed graph G=(V,E) with weight function w:E→R mapping edges to real-valued weights, find the minimum-weight path from a given source vertex s to another vertex v.
  - The weight w(p) of a path p=<v_0,v_1,…,v_k> is the sum of the weights of its constituent edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
  - The shortest-path weight δ(s,v) from s to v by

$$\delta(s,v) = \begin{cases} \min\{w(p): s \rightsquigarrow v\} & \text{if there is a path } p \text{ from u to v} \\ \infty & \text{otherwise} \end{cases}$$

Single-source Shortest-Paths Problem

- The algorithm for the single-source shortest path problem can solve many other problems, including the following variants:
  - Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v.
    - If we solve the single-source problem with source vertex u,

    we solve this problem too.
    - All known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms
  - All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.
    - Although we can solve this problem by running a single source algorithm once from each vertex, we usually can solve it faster.
    - Details are in next chapter.

Optimal Substructure Property

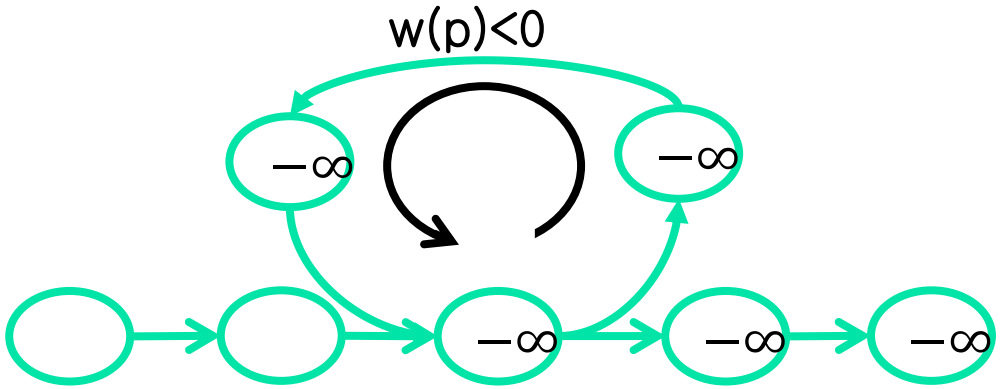
- A shortest path between two vertices contains other shortest paths within it.
- Lemma 24.1 (Subpaths of shortest paths are shortest paths)
  - Consider a weighted graph G, with weight function $w:E \rightarrow R$,
  - Let $p=<v_0,v_1,v_2,v_4,\ldots,v_{k-1},v_k>$ be a shortest path from vertex v_0 to vertex v_k.
  - For any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij}=<v_i,v_{i+1},\ldots,v_j>$ be the subpath of p from vertex v_i to vertex v_j.
  - Then, p_{ij} is a shortest path from v_i to v_j.
- Proof is straightforward.

Shortest Path Properties

- Some instances of the single-source shortest-paths problem may include edges whose weights are negative.
- If the graph G=(V,E) contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s,v)$ remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined.

Shortest Path Properties

- No path from s to a vertex on the cycle can be a shortest path—we can always find a path with lower weight by following the proposed shortest path and then traversing the negative-weight cycle.
- If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$.

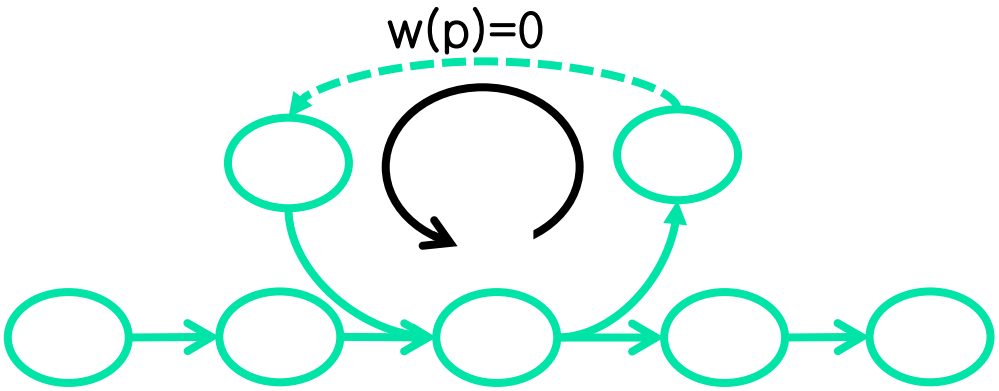


Can a Shortest Path Contain a Cycle?

- As we have just seen, it cannot contain a negative-weight cycle.
- Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.
- We can remove a 0-weight cycle from any path to produce another path whose weight is the same.
- Thus, if there is a shortest path from a source vertex s to a destination vertex that contains a 0-weight cycle, then there is another shortest path from s to without this cycle.

Shortest Path Properties

- Without loss of generality, we can assume that when we are finding shortest paths, they have no cycles.
- Since any acyclic path in a graph G=(V,E), contains at most |V| distinct vertices, it also contains at most |V|-1 edges.
- If have only 0-weight cycles, we can restrict our attention to shortest paths of at most |V|-1 edges.



Representing Shortest Paths

- We represent shortest paths similarly to how we represented breadth-first trees.
- Shortest-paths tree:
  - Predecessor subgraph $G_\pi = (V_\pi, E_\pi)$
    - $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$
    - $E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$
  - V_π is the set of vertices of G with non-NIL predecessors, plus the source s.
  - E_π is the the set of edges induced by the π values for vertices in V_π.
  - G_π forms a rooted tree with root s containing a shortest path from the source s to every vertex that is reachable from s.

A Shortest-path Tree

- Let G=(V, E) be a weighted, directed graph with weight function w:E→R.
- Assume that G contains no negative-weight cycles reachable from the source vertex s ∈ V, so that shortest paths are well defined.
- A shortest-path tree rooted at s is a directed subgraph G′=(V′, E′), where V′⊂V and E′ ⊂ E, such that
  - V′ is the set of vertices reachable from s in G.
  - G′ forms a rooted tree with root s.
  - For all v ∈ V′, the unique simple path from s to v in G′ is a shortest path from s to v in G.

Relaxation

- The algorithms in this chapter use the technique of relaxation.
- For all $v \in V$, we maintain an attribute v.d which is an upper bound on the weight of a shortest path from source s to v.
- We call v.d a shortest-path estimate.

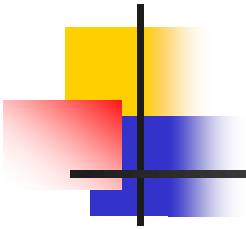
INITIALIZE-SINGLE-SOURCE(G,s)

1. **for** each vertex $v \in G.V$
2. $\quad v.d = \infty$
3. $\quad v.\pi = NIL$
4. $s.d = 0$

Relaxation

- The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating v.d and $v.\pi$.
- The following code performs a relaxation step on edge (u,v) in O(1) time.

```
RELAX(u, v, w)
1.    if  v.d>u.d + w(u,v)
2.          v.d = u.d + w(u,v)
3.          v.π = u
```



Relaxation

- Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.
- Moreover, relaxation is the only means by which shortest path estimates and predecessors change.
- The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges.
  - The Bellman-Ford algorithm relaxes each edge $|V|-1$ times.
  - Dijkstra's algorithm for directed acyclic graphs relaxes each edge exactly once.

Initialize

INITIALIZE-SINGLE-SOURCE(G,s)

1. **for** each vertex $v \in G.V$
2. $v.d = \infty$
3. $v.\pi = NIL$
4. $s.d = 0$

Relaxation

RELAX(u, v, w)

1. **if** v.d>u.d + w(u,v)
2. v.d = u.d + w(u,v)
3. v.π = u