Graph Algorithms – 3

Minimum Spanning Tree (MST)

- Given a connected, undirected, graph $G = (V, E)$, a *spanning tree* is an *acyclic* subset of edges $T \subseteq E$ that connects all the vertices together.

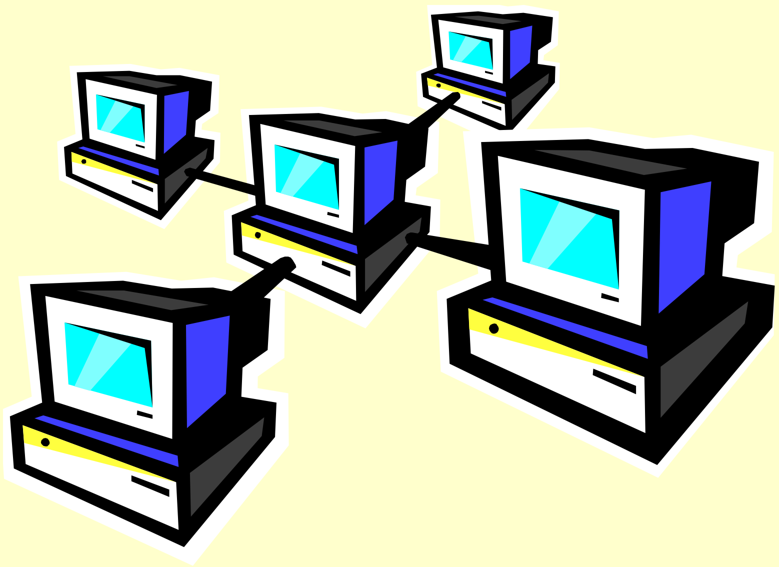
- Assuming G is weighted, we define the *cost* of a spanning tree T to be the sum of edge weights in the spanning tree.

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

- A *minimum spanning tree (MST)* is a spanning tree of minimum weight.

Applications of MST

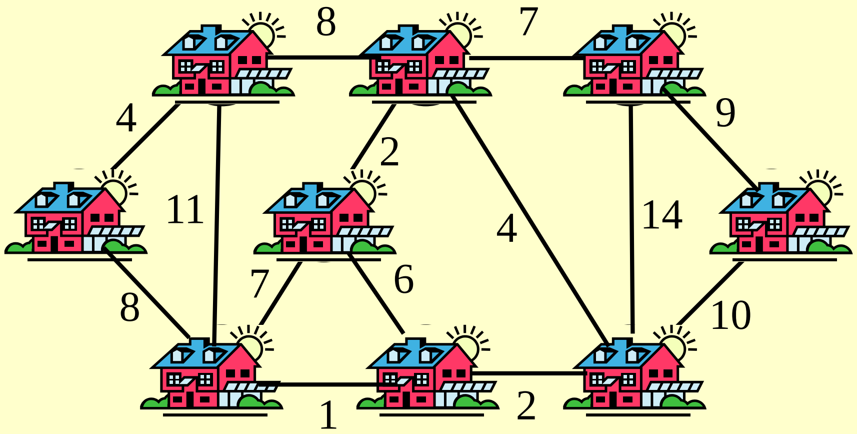
- Find the least expensive way to connect a set of nodes in,
  - » Communication networks
  - » Circuit design
  - » Layout of highway systems



Example

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:

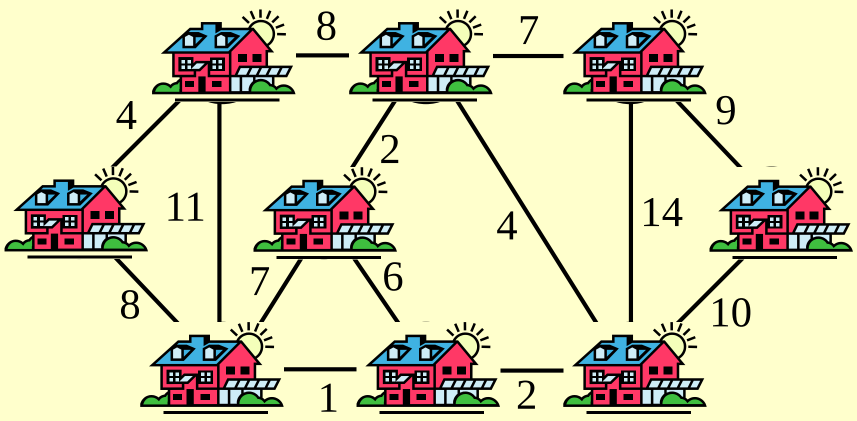
1. Everyone stays connected
   i.e., can reach every house from all other houses
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
  - » Vertices = houses, Edges = roads
- A **weight** w(u, v) on each edge (u, v) $\in$ E

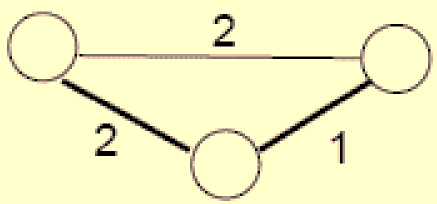
Find T $\subseteq$ E such that:

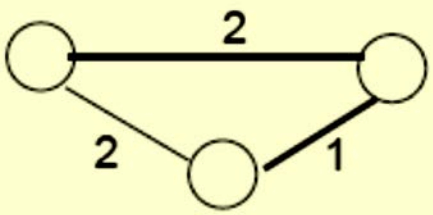
1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is minimized



Properties of Minimum Spanning Trees

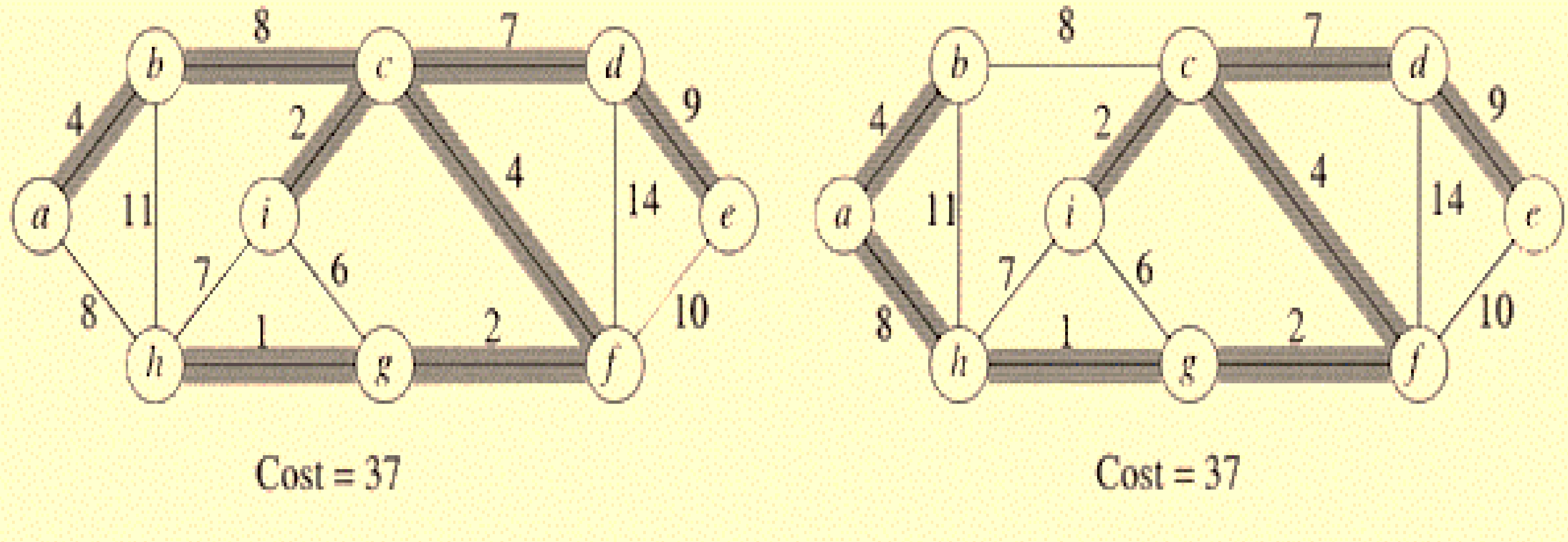
- Minimum spanning tree is **not** unique





- MST has no cycles:
  - » We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- Number of edges in a MST:
  - » |V| - 1

Examples of MST



- Not only do the edges sum to the same value, but the same set of edge weights appear in the two MSTs. NOTE: An MST may not be unique.

Generic Approaches

Two greedy algorithms for computing MSTs:

- » Kruskal's Algorithm
- » Prim's Algorithm

Facts about Trees

- A tree with n vertices has exactly n-1 edges ($|E| = |V| - 1$)
- There exists a unique path between any two vertices of a tree
- Adding any edge to a tree creates a unique cycle; breaking any edge on this cycle restores a tree

Intuition Behind Greedy MST

- We maintain in a subset of edges A, which will initially be empty, and we will add edges one at a time, until equals the MST. We say that a subset $A \subseteq E$ is *viable* if A is a subset of edges in some MST. We say that an edge $(u,v) \in E\text{-}A$ is *safe* if $A \cup \{(u,v)\}$ is viable.
- Basically, the choice (u,v) is a safe choice to add so that A can still be extended to form an MST. Note that if A is viable it cannot contain a cycle.
- A generic greedy algorithm operates by repeatedly adding any *safe edge* to the current spanning tree.

Generic-MST (G, w)

1. $A \leftarrow \varnothing$ // *A* trivially satisfies invariant

// lines 2-4 maintain the invariant

2. while *A* does not form a spanning tree
3. do find an edge (u,v) that is safe for *A*
4. $A \leftarrow A \cup \{(u,v)\}$

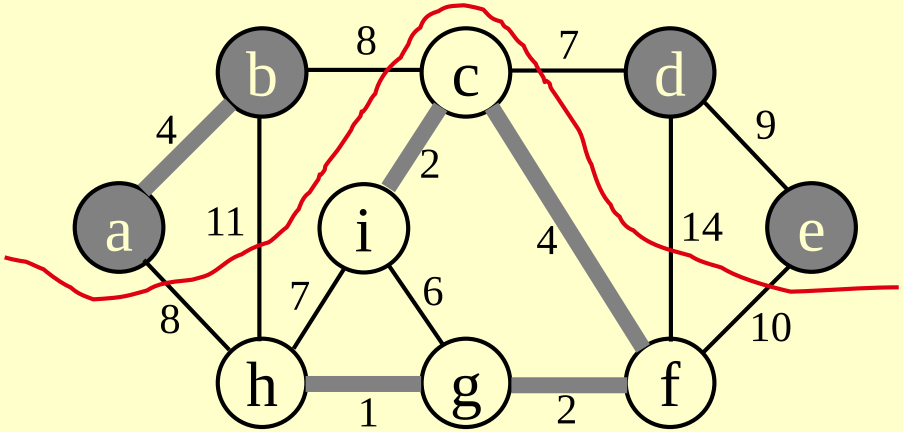
5. return *A* // *A* is now a MST

The generic method manages a set of edges A, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

Definitions

- A *cut* $(S, V\text{-}S)$ is just a partition of the vertices into 2 disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V\text{-}S$.
- Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut.
- An edge of E is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights).



When is an Edge Safe?

- If we have computed a partial MST, and we wish to know which edges can be added that do NOT induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

- Intuition says that since all edges crossing a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice.

MST Lemma

Let $G = (V, E)$ be a connected, undirected graph with real-value weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V\text{-}S)$ be any cut that respects A, and let (u,v) be a light edge crossing this cut. Then, the edge (u,v) is safe for A.

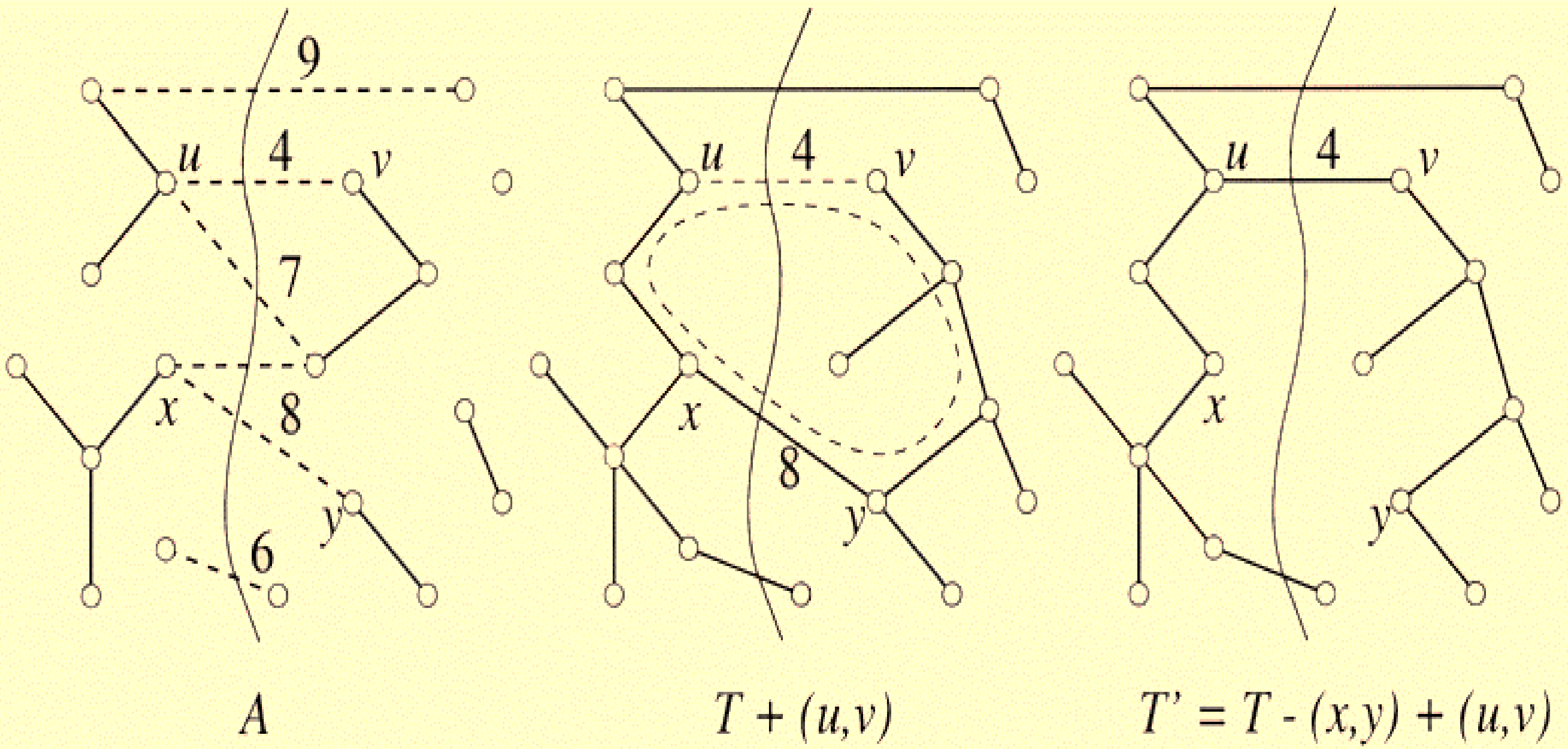
Proof: Must show that $A \cup \{(u,v)\}$ is a subset of some MST

Method:

- Find arbitrary MST T containing A
- Use a cut-and-paste technique to find another MST T that contains $A \cup \{(u,v)\}$

This cut-and-paste idea is an important proof technique.

MST Lemma

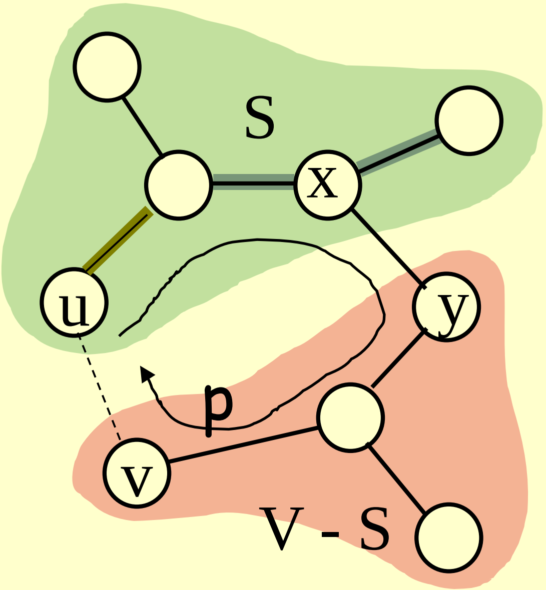


Step 1

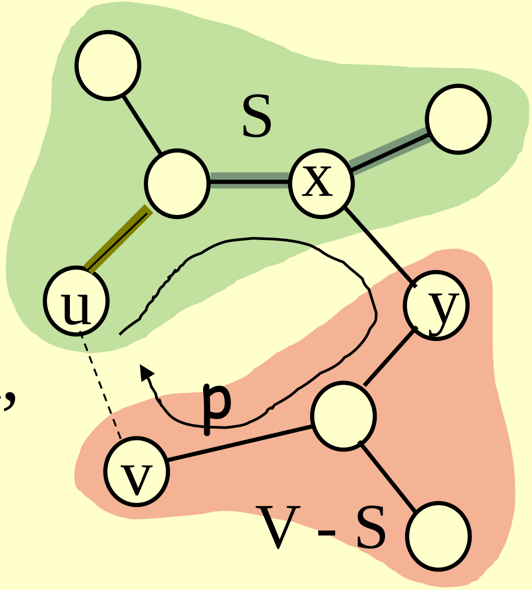
- Let T be any MST for G containing A.
  - » We know such a tree exists because A is viable.
- If (u, v) is in T then we are done as (u, v) must be a safe edge for A.

Constructing *T'*

- **If (*u*, *v*) is not in *T***, then add it to *T*, thus creating a cycle. Since *u* and *v* are on opposite sides of the cut, and since any cycle must cross the cut an even number of times, there must be at least one other edge (*x*, *y*) in *T* that crosses the cut.
- The edge (*x*, *y*) is not in *A* (because the cut respects *A*). By removing (*x*,*y*) we restore a spanning tree, *T'*.
- $T' = T - \{(x,y)\} \cup \{(u,v)\}$
- Now must show
  - » *T'* is a *minimum* spanning tree
  - » $A \cup \{(u,v)\}$ is a subset of *T'*



Conclusion of Proof



- *T'* is an MST: We have

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

  Since (u,v) is a light edge crossing the cut, we have $w(u,v) \leq w(x,y)$.
  Thus $w(T') \leq w(T)$.
  So *T'* is also a minimum spanning tree.

- $A \cup \{(u,v)\} \subseteq T'$: Remember that (x, y) is not in A and $A \subseteq T$,
  Thus $A \subseteq T - \{(x, y)\}$, and thus

  $$A \cup \{(u,v)\} \subseteq T - \{(x, y)\} \cup \{(u,v)\} = T'$$

Basics of Kruskal's Algorithm

- Attempts to add edges to *A* in increasing order of weight (lightest edge first)
  - » If the next edge does not induce a cycle among the current set of edges, then it is added to *A*.
  - » If it does, then this edge is passed over, and we consider the next edge in order.
  - » As this algorithm runs, the edges of *A* will induce a forest on the vertices and the trees of this forest are merged together until we have a single tree containing all vertices.

Detecting a Cycle

- We can perform a DFS on subgraph induced by the edges of *A*, but this takes too much time.
- Use "disjoint set UNION-FIND" data structure. This data structure supports 3 operations:

  Create-Set(u): create a set containing u.

  Find-Set(u): Find the set that contains u.

  Union(u, v): Merge the sets containing u and v.

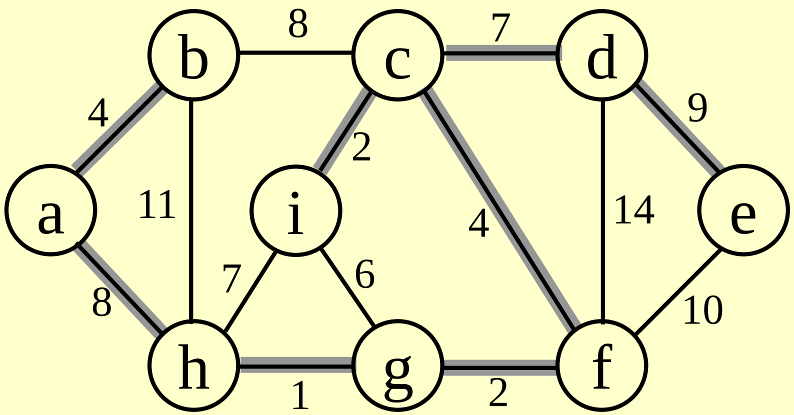
  Each can be performed in $O(lg\ n)$ time.
- The vertices of the graph will be elements to be stored in the sets; the sets will be vertices in each tree of *A* (stored as a simple list of edges).

MST-Kruskal(G, w)

1. $A \leftarrow \emptyset$ // initially A is empty
2. for each vertex $v \in V[G]$ // line 2-3 takes $O(V)$ time
3.   do Create-Set(v) // create set for each vertex
4. sort the edges of E by nondecreasing weight w
5. for each edge $(u,v) \in E$, in order by nondecreasing weight
6.   do if Find-Set(u) $\neq$ Find-Set(v) // u&v on different trees
7.   then $A \leftarrow A \cup \{(u,v)\}$
8.   Union(u,v)
9. return A

Total running time is $O(E\ lg\ E)$.

Example

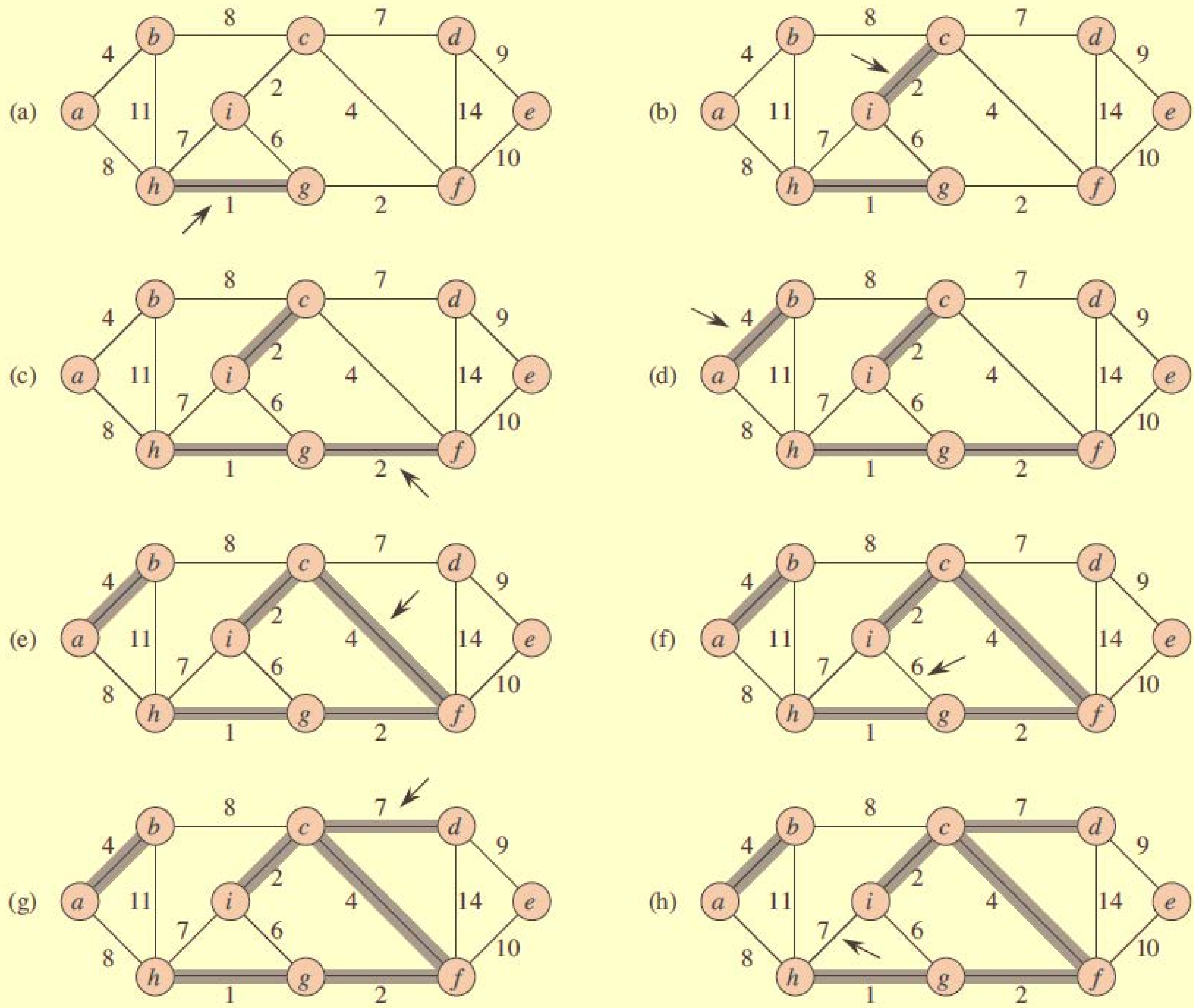


1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

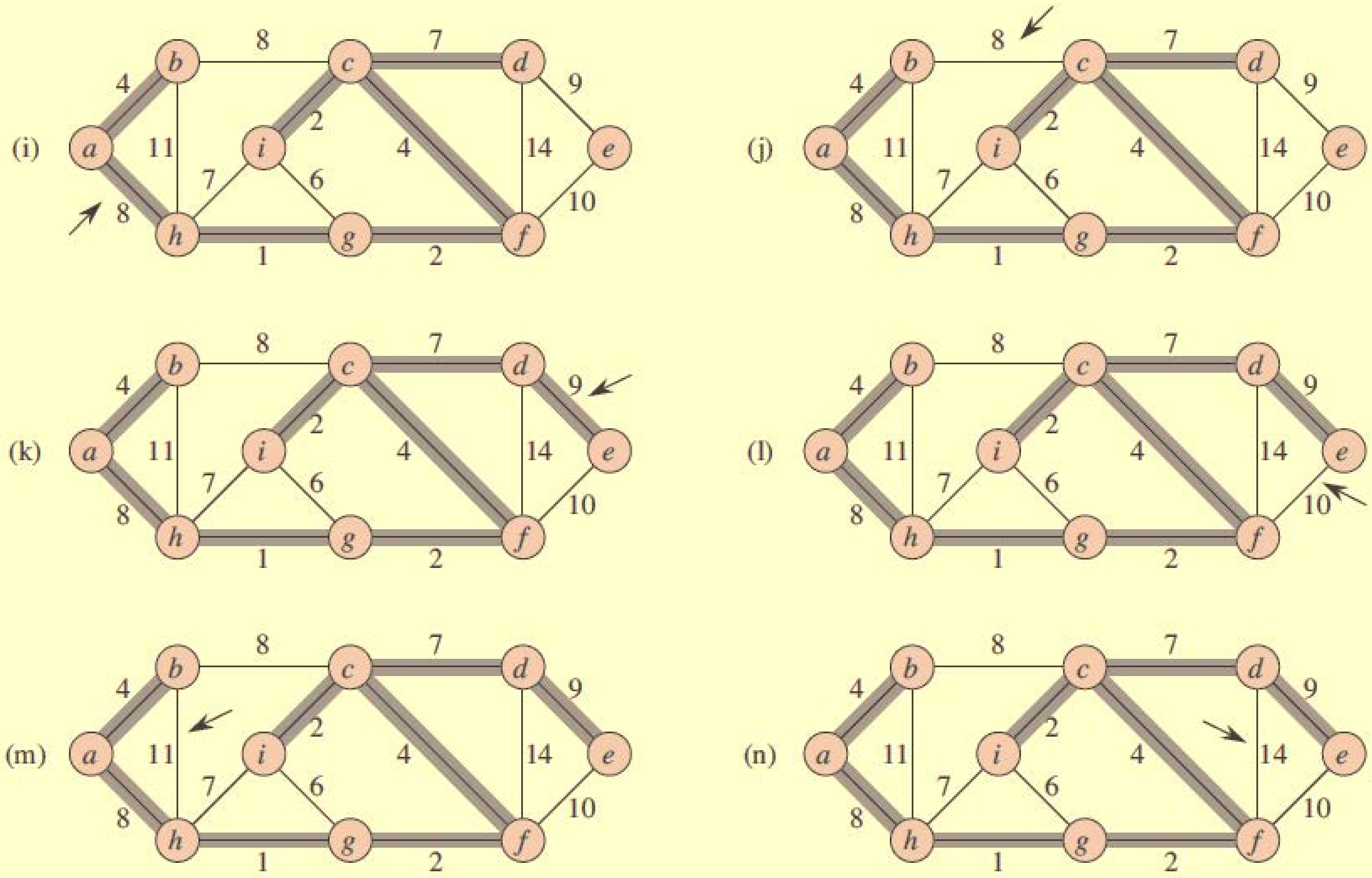
{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1.	Add (h, g)	{g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2.	Add (c, i)	{g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3.	Add (g, f)	{g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4.	Add (a, b)	{g, h, f}, {c, i}, {a, b}, {d}, {e}
5.	Add (c, f)	{g, h, f, c, i}, {a, b}, {d}, {e}
6.	Ignore (i, g)	{g, h, f, c, i}, {a, b}, {d}, {e}
7.	Add (c, d)	{g, h, f, c, i, d}, {a, b}, {e}
8.	Ignore (i, h)	{g, h, f, c, i, d}, {a, b}, {e}
9.	Add (a, h)	{g, h, f, c, i, d, a, b}, {e}
10.	Ignore (b, c)	{g, h, f, c, i, d, a, b}, {e}
11.	Add (d, e)	{g, h, f, c, i, d, a, b, e}
12.	Ignore (e, f)	{g, h, f, c, i, d, a, b, e}
13.	Ignore (b, h)	{g, h, f, c, i, d, a, b, e}
14.	Ignore (d, f)	{g, h, f, c, i, d, a, b, e}

Example: Kruskal's Algorithm



Example: Kruskal’s Algorithm

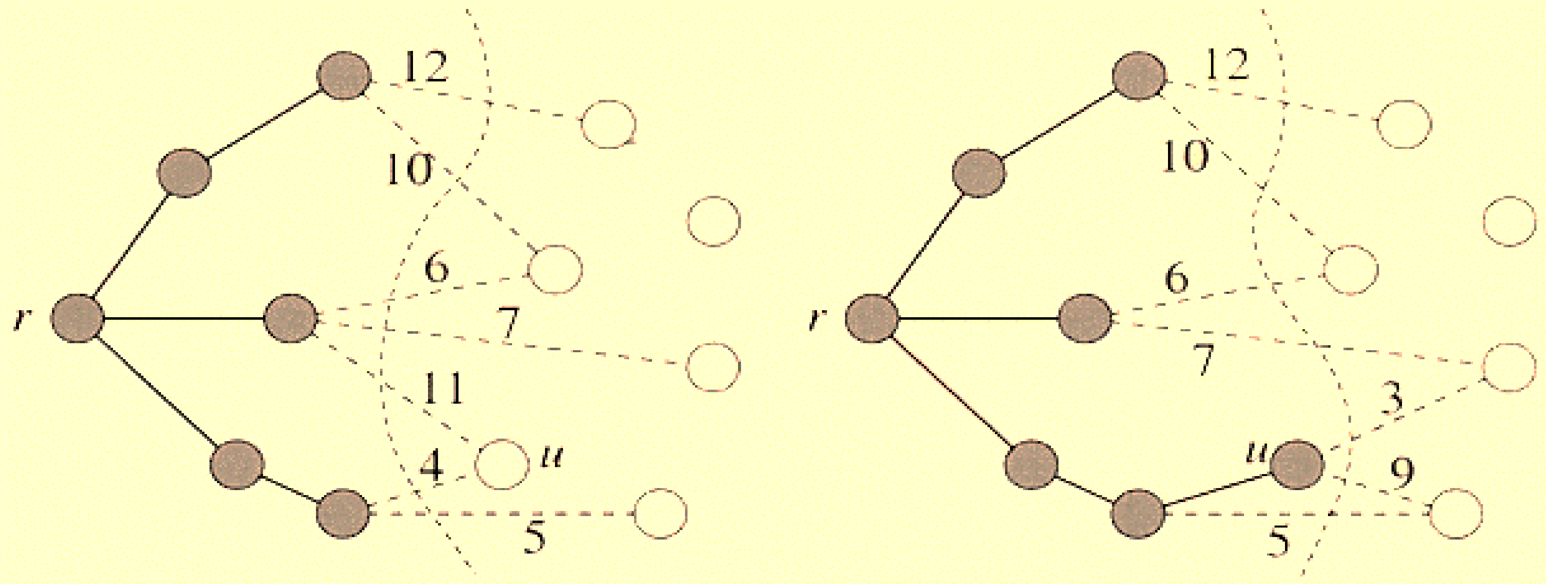


Analysis of Kruskal

- Lines 1-3 (initialization): O(V)
- Line 4 (sorting): O(E lg E)
- Lines 6-8 (set-operation): O(E log E)
- Total: O(E log E)

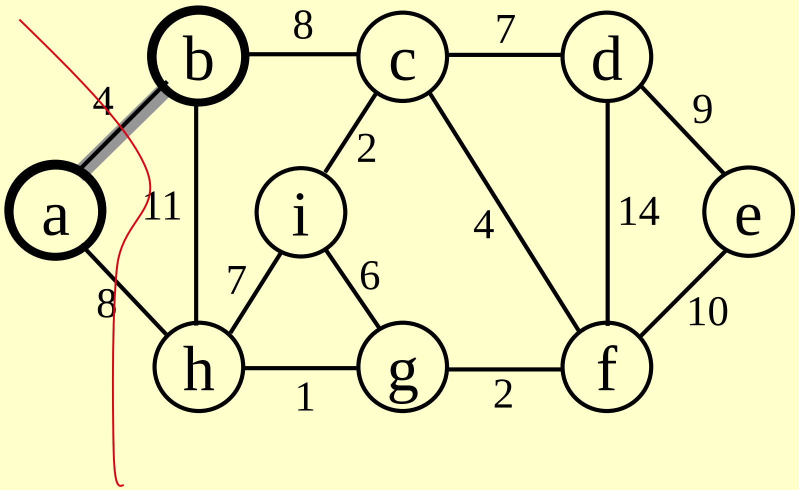
Intuition behind Prim's Algorithm

- Consider the set of vertices S currently part of the tree, and its complement (V-S). We have a cut of the graph and the current set of tree edges A is respected by this cut.
- Which edge should we add next? *Light edge!*



Basics of Prim 's Algorithm

- The edges in set A always form a single tree
- Starts from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - » Find a light edge crossing (V_A, $V - V_A$)
  - » Add this edge to A
  - » Repeat until the tree spans all vertices



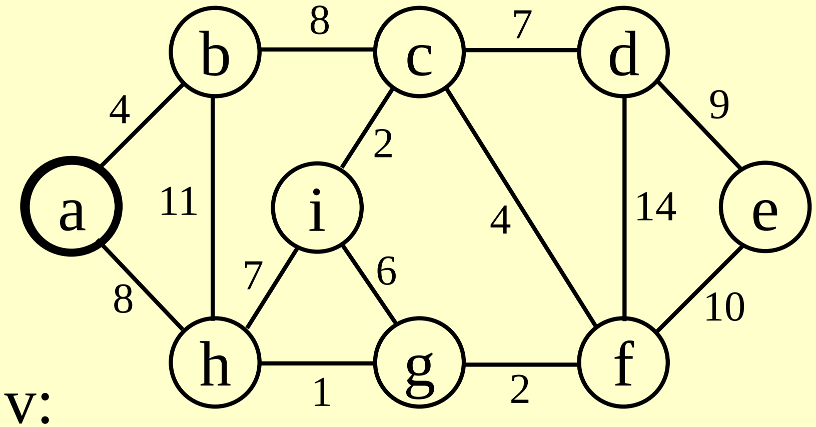
- Implementation Issues:
  - » How to update the cut efficiently?
  - » How to determine the light edge quickly?

Implementation: Priority Queue

- Priority queue implemented using heap can support the following operations in $O(lg\ n)$ time:
  - » Insert (Q, u, key): Insert u with the key value key in Q
  - » u = Extract_Min(Q): Extract the item with minimum key value in Q
  - » Decrease_Key(Q, u, new_key): Decrease the value of u's key value to new_key

How to Find Light Edges Quickly?

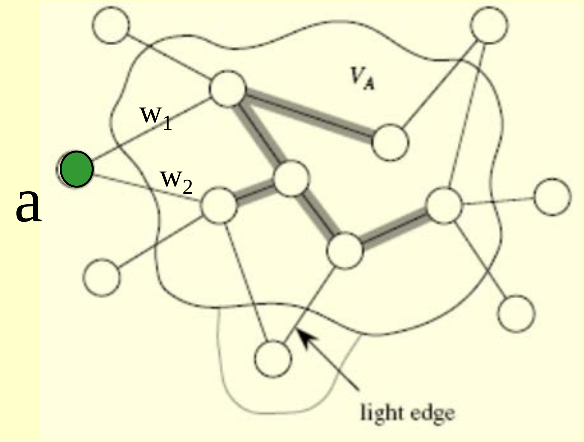
Use a priority queue Q:



- Contains vertices not yet included in the tree, i.e., $(V - V_A)$
  - » $V_A = \{a\}$, $Q = \{b, c, d, e, f, g, h, i\}$
- We associate a key with each vertex v:

key[v] = minimum weight of any edge (u, v) connecting v to V_A

Key[a]=min(w_1,w_2)

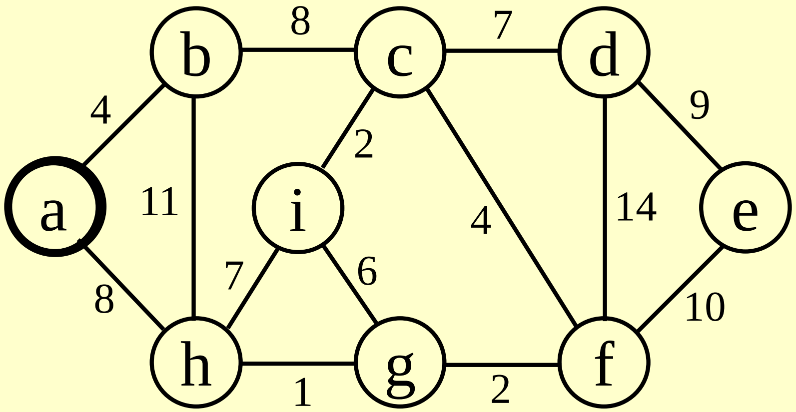


How to Find Light Edges Quickly? (cont.)

- After adding a new node to V_A we update the weights of all the nodes adjacent to it

  e.g., after adding a to the tree, k[b]=4 and k[h]=8

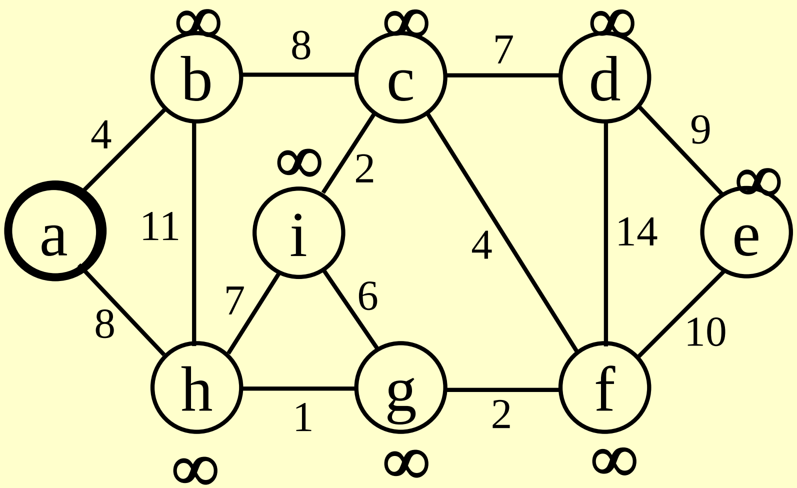
- Key of v is ∞ if v is not adjacent to any vertices in V_A



MST-Prim(G, w, r)

```
Q ← V[G]
for each vertex u ∈ Q                         // initialization: O(V) time
      do key[u] ← ∞
key[r] ← 0                                    // start at the root
π[r] ← NIL                                    // set parent of r to be NIL
while Q ≠ ∅                                   // until all vertices in MST
      do u ← Extract-Min(Q)                   // vertex with lightest edge
            for each v ∈ adj[u]
                  do if v ∈ Q and w(u,v) < key[v]
                        then π[v] ← u
                             key[v] ← w(u,v)  // new lighter edge out of v
                             decrease_Key(Q, v, key[v])
```

Example

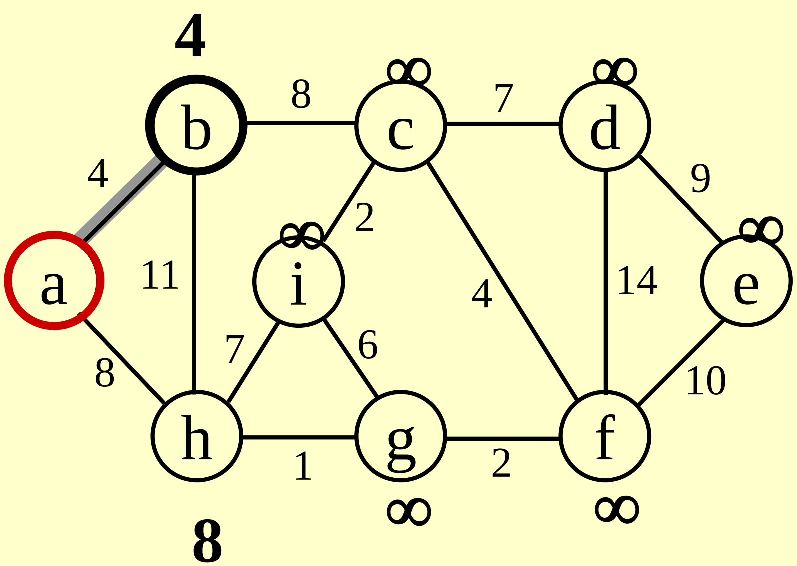


$0\ \infty\ \infty\ \infty\ \infty\ \infty\ \infty\ \infty\ \infty$

Q = {a, b, c, d, e, f, g, h, i}

$V_A = \varnothing$

Extract-MIN(Q) $\Rightarrow$ a



key [b] = 4 π [b] = a

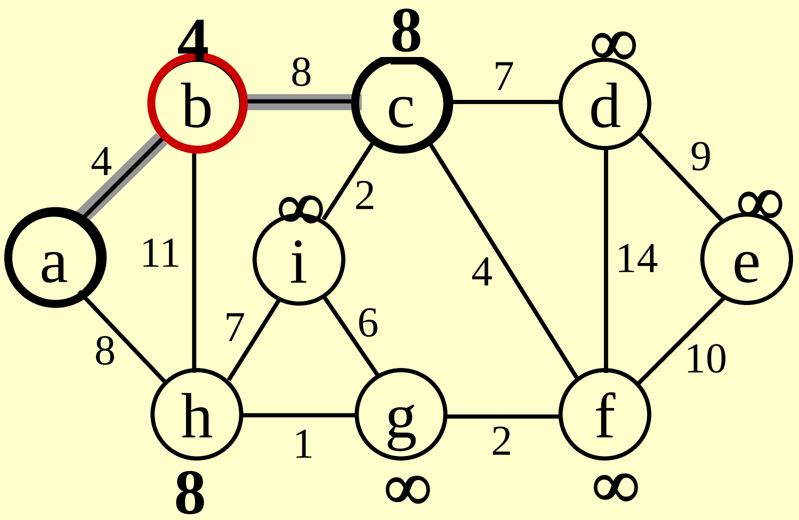
key [h] = 8 π [h] = a

4 ∞ ∞ ∞ ∞ ∞ **8** ∞

Q = {b, c, d, e, f, g, h, i} V_A = {a}

Extract-MIN(Q) $\Rightarrow$ b

Example



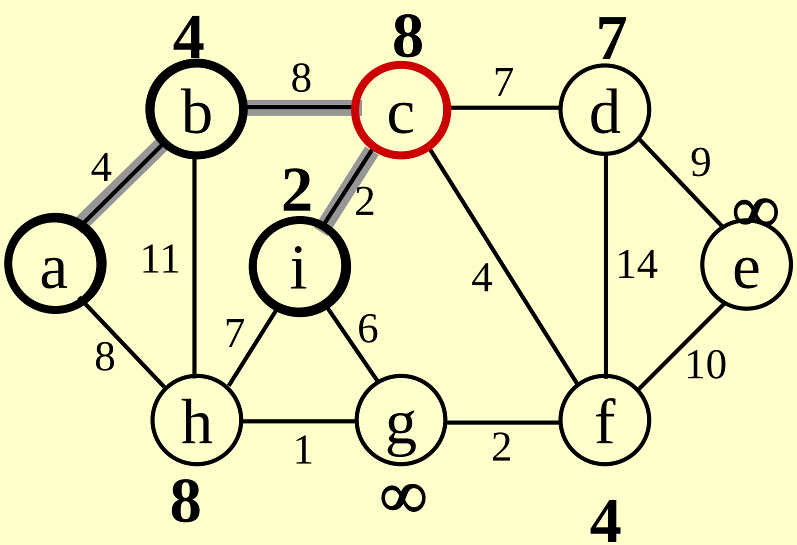
key [c] = 8   π [c] = b

key [h] = 8   π [h] = a - unchanged

8 ∞ ∞ ∞ ∞ 8 ∞

Q = {c, d, e, f, g, h, i} V_A = {a, b}

Extract-MIN(Q) ⇒ c



key [d] = 7   π [d] = c

key [f] = 4   π [f] = c

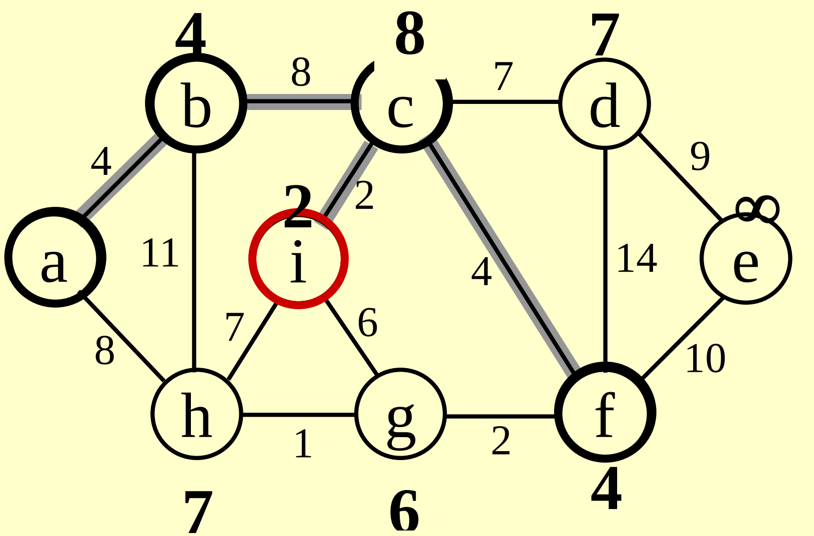
key [i] = 2   π [i] = c

7 ∞ 4 ∞ 8 2

Q = {d, e, f, g, h, i} V_A = {a, b, c}

Extract-MIN(Q) ⇒ i

Example



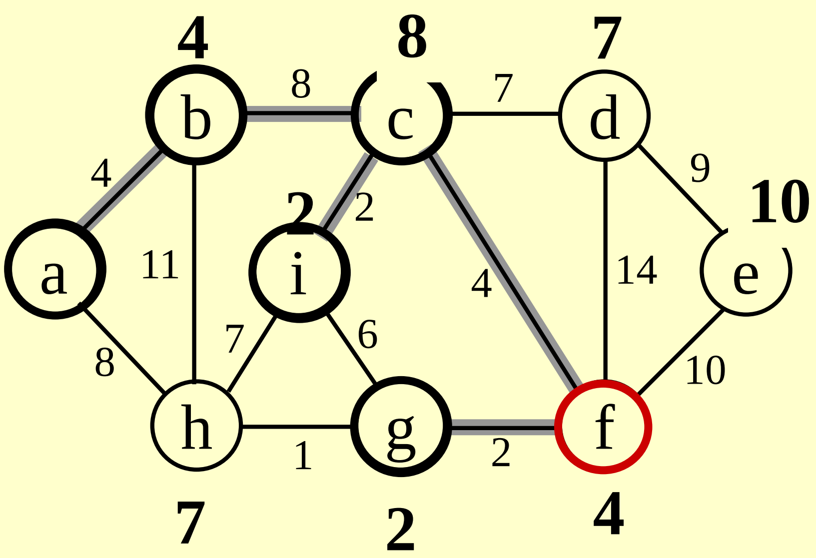
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

Q = {d, e, f, g, h} V_A = {a, b, c, i}

Extract-MIN(Q) $\Rightarrow$ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

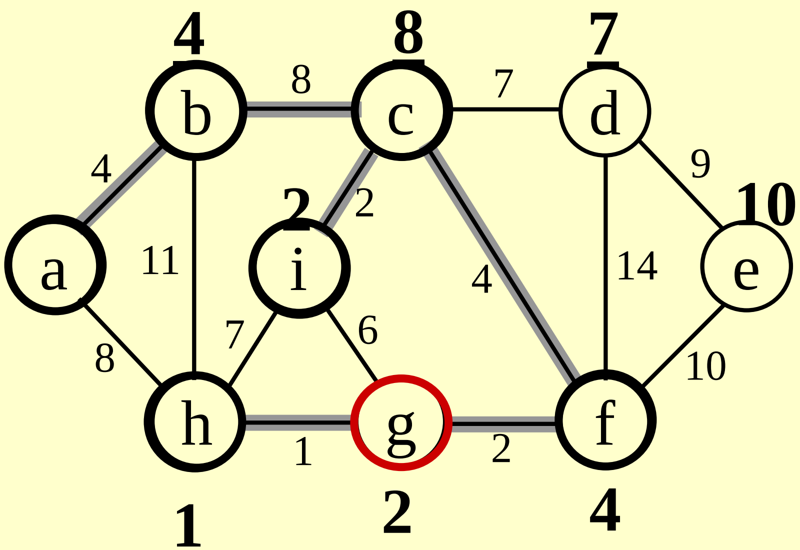
key [e] = 10 π [e] = f

7 10 2 8

Q = {d, e, g, h} V_A = {a, b, c, i, f}

Extract-MIN(Q) $\Rightarrow$ g

Example

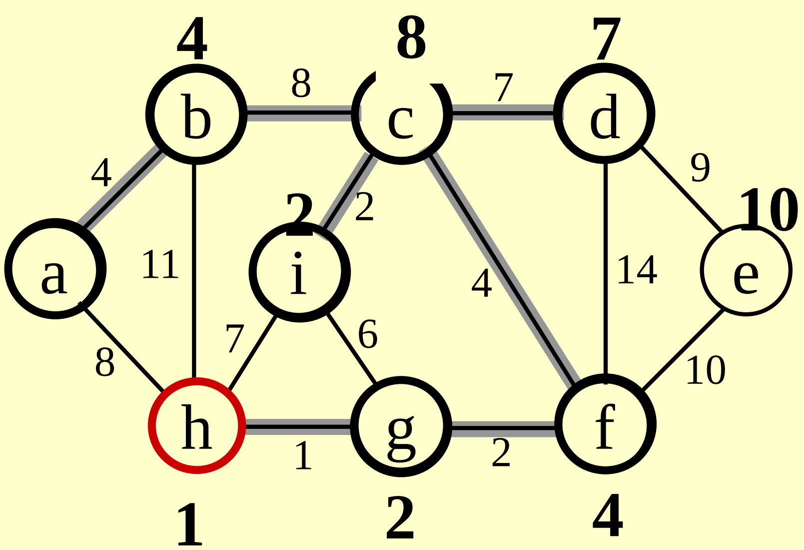


key [h] = 1 π [h] = g

7 10 1

Q = {d, e, h} V_A = {a, b, c, i, f, g}

Extract-MIN(Q) $\Rightarrow$ h

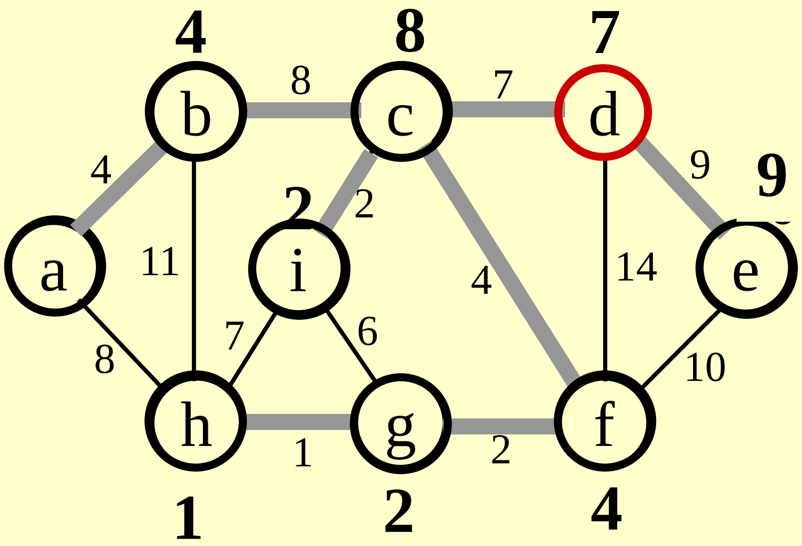


7 10

Q = {d, e} V_A = {a, b, c, i, f, g, h}

Extract-MIN(Q) $\Rightarrow$ d

Example



key [e] = 9 π [e] = f

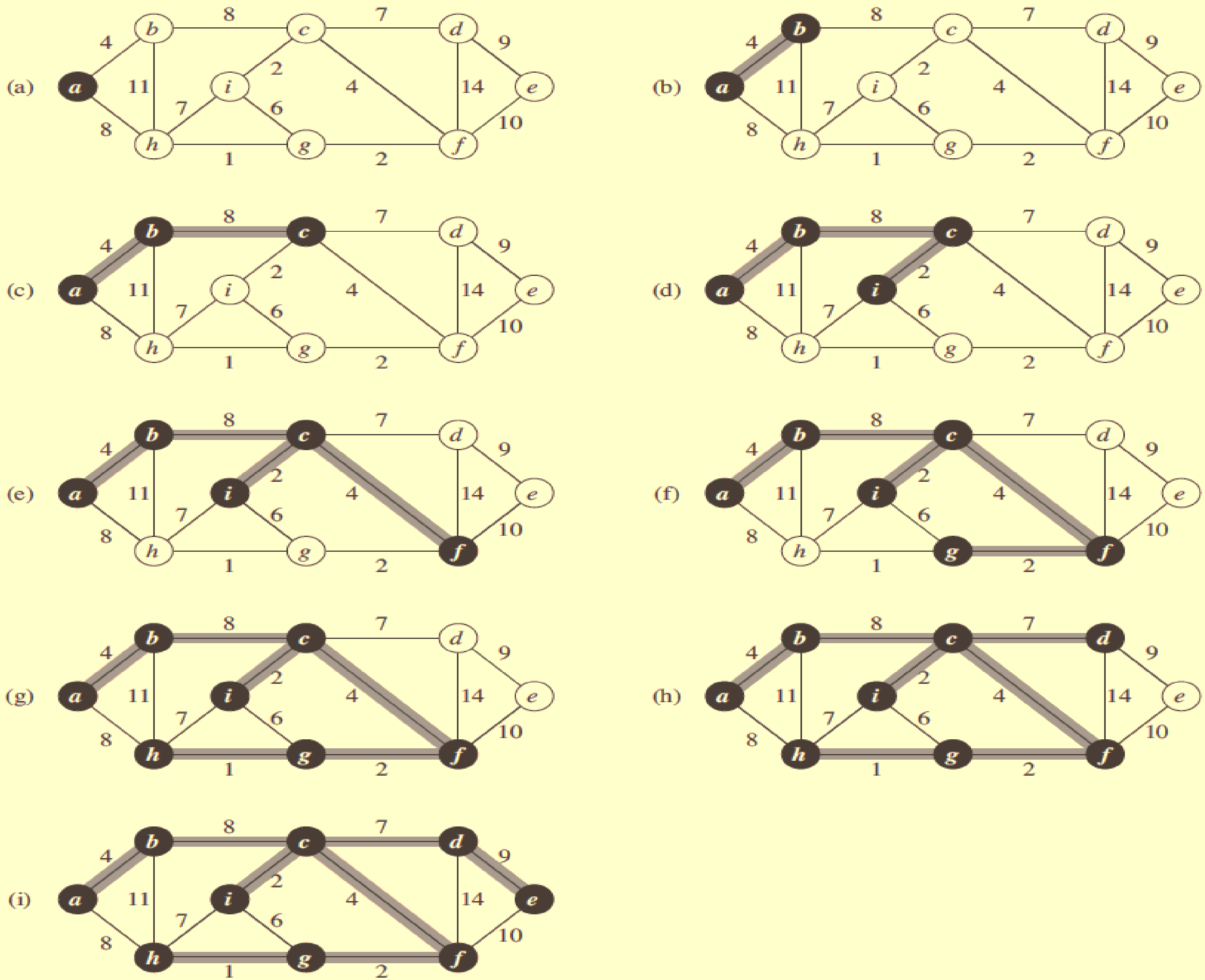
9

Q = {e} V_A = {a, b, c, i, f, g, h, d}

Extract-MIN(Q) $\Rightarrow$ e

Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

Example: Prim's Algorithm



Analysis of Prim

- Extracting the vertex from the queue: $O(\lg n)$
- For each incident edge, decreasing the key of the neighboring vertex: $O(\lg n)$ where $n = |V|$
- The other steps are constant time.
- The overall running time is, where $e = |E|$

$$T(n) = \sum_{u \in V}(\lg n + \deg(u) \lg n) = \sum_{u \in V}(1 + \deg(u)) \lg n$$
$$= \lg n \, (n + 2e) = O((n + e) \lg n)$$

Essentially same as Kruskal's: $O((n+e) \lg n)$ time