CS 5413
Chapter 23

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

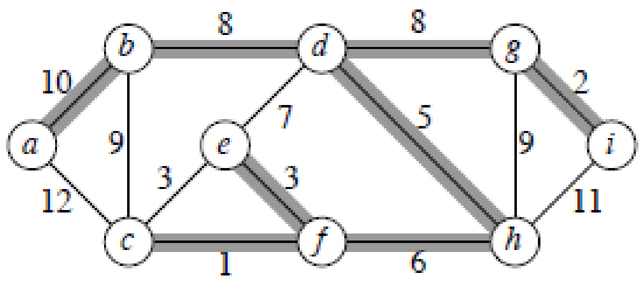
  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

Model as a graph:

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

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[edges in MST are shaded]* :



In this example, there is more than one MST. Replace edge (e, f) by (c, e). Get a different spanning tree with the same weight.

Growing a minimum spanning tree

Some properties of an MST:

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

Building up the solution

- We will build a set A of edges.
- Initially, A has no edges.
- As we add edges to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.

- Add only edges that maintain the 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. So we will add only safe edges.

Generic MST algorithm

```
GENERIC-MST$(G, w)$
$A \leftarrow \emptyset$
while $A$ is not a spanning tree
    do find an edge $(u, v)$ that is safe for $A$
        $A \leftarrow A \cup \{(u, v)\}$
return $A$
```

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since we add only safe edges, A remains a subset of some MST.

Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Finding a safe edge

How do we find safe edges?

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

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? (Which would be (c, f) in this case.)

Some definitions: Let $S \subset V$ and $A \subseteq E$.

- A *cut* $(S, V - S)$ is a partition of vertices into disjoint sets 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 edge crossing it.

Theorem

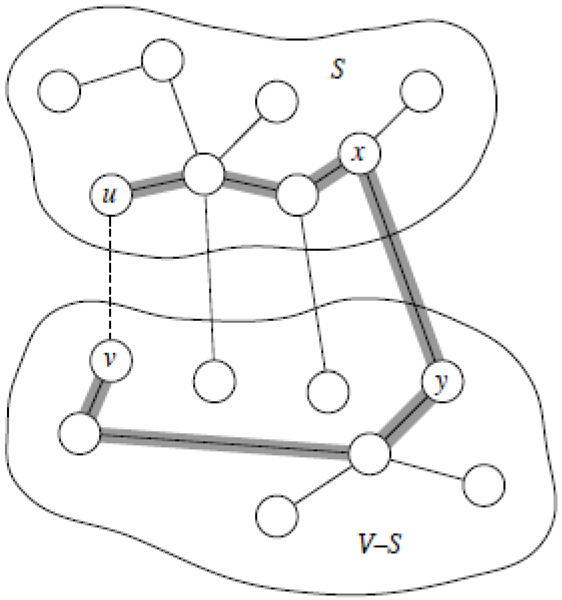
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). We'll construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut $(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \leq w(x, y)$.



[Except for the dashed edge (u, v), all edges shown are in T. A is some subset of the edges of T, but A cannot contain any edges that cross the cut $(S, V - S)$, since this cut respects A. Shaded edges are the path p.]

Since the cut respects 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 $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) , \end{aligned}$$

since $w(u, v) \le w(x, y)$. Since T' is a spanning tree, $w(T') \le w(T)$, and T is an MST, then T' must be an MST.

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' is an MST, (u, v) is safe for A. ■ (theorem)

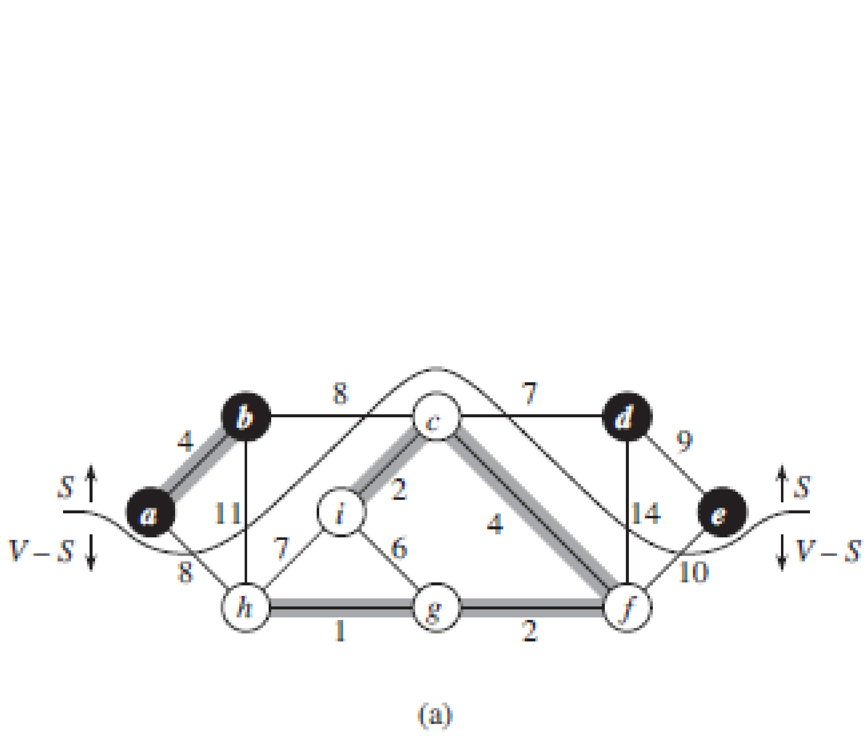


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** Black vertices are in the set S, and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut. **(b)** The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

So, 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. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V| - 1$ safe edges, we're down to just one component.

Corollary

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

Proof Set $S = V_C$ in the theorem. ■ (corollary)

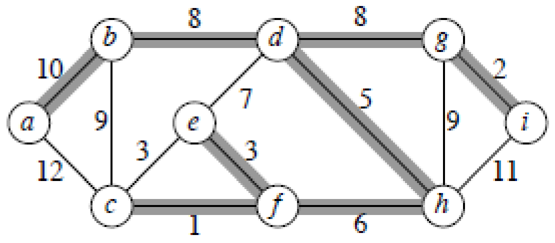
This naturally leads to the algorithm called Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbf{R}$.

- 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($V$, $E$, $w$)
$A \leftarrow \emptyset$
for each vertex $v \in V$
    do MAKE-SET($v$)
sort $E$ into nondecreasing order by weight $w$
for each $(u, v)$ taken from the sorted list
    do if FIND-SET($u$) $\neq$ FIND-SET($v$)
        then $A \leftarrow A \cup \{(u, v)\}$
             UNION($u$, $v$)
return $A$
```



Run through the above example to see how Kruskal's algorithm works on it:

(c, f)	:	safe
(g, i)	:	safe
(e, f)	:	safe
(c, e)	:	reject
(d, h)	:	safe
(f, h)	:	safe
(e, d)	:	reject
(b, d)	:	safe
(d, g)	:	safe
(b, c)	:	reject
(g, h)	:	reject
(a, b)	:	safe

At this point, we have only one component, so all other edges will be rejected. *[We could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to A.]*

Get the shaded edges shown in the figure.

Suppose we had examined (c, e) *before* (e, f). Then would have found (c, e) safe and would have rejected (e, f).

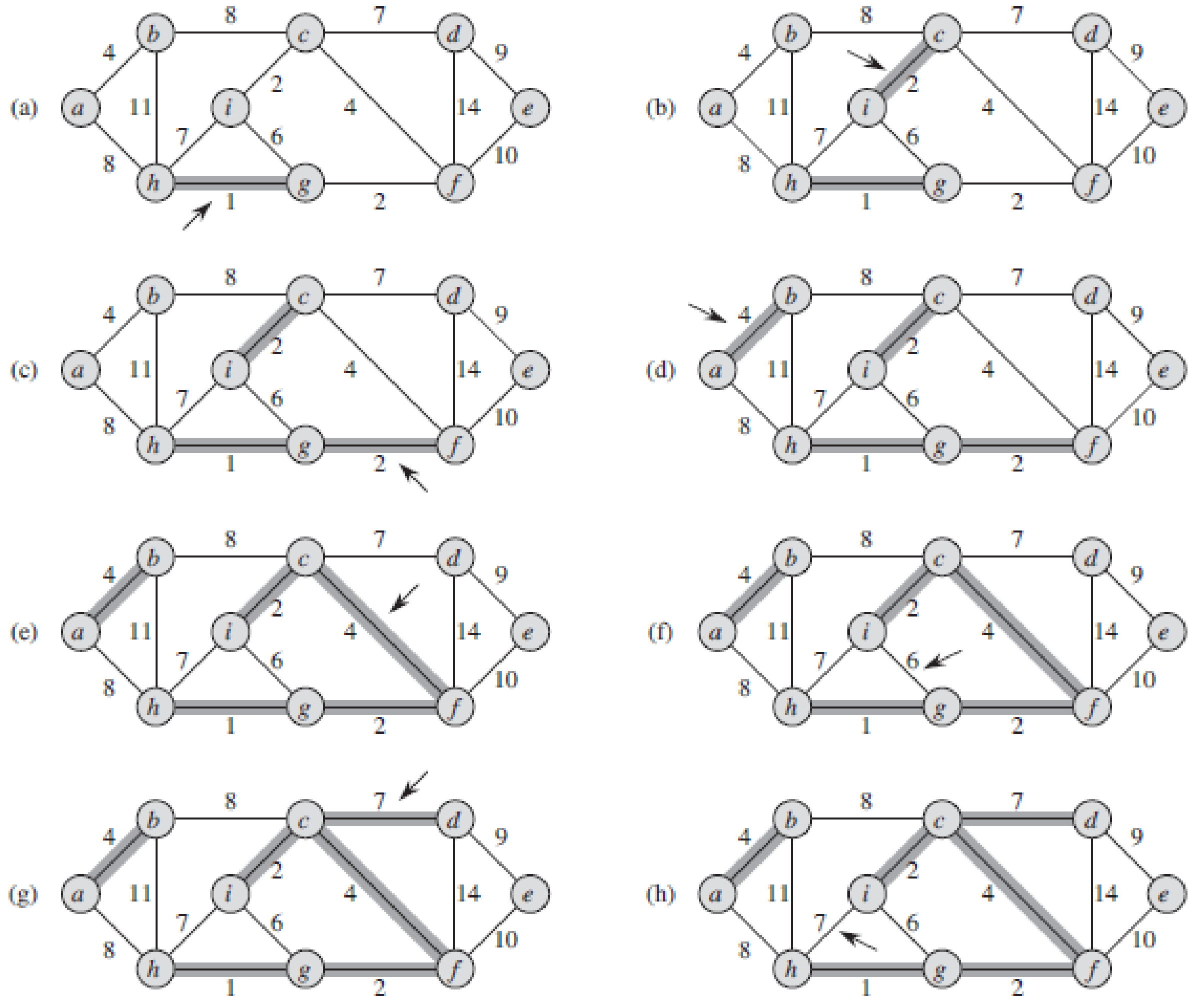


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, line 7 adds the edge (u, v) to A, and line 8 merges the vertices in the two trees.

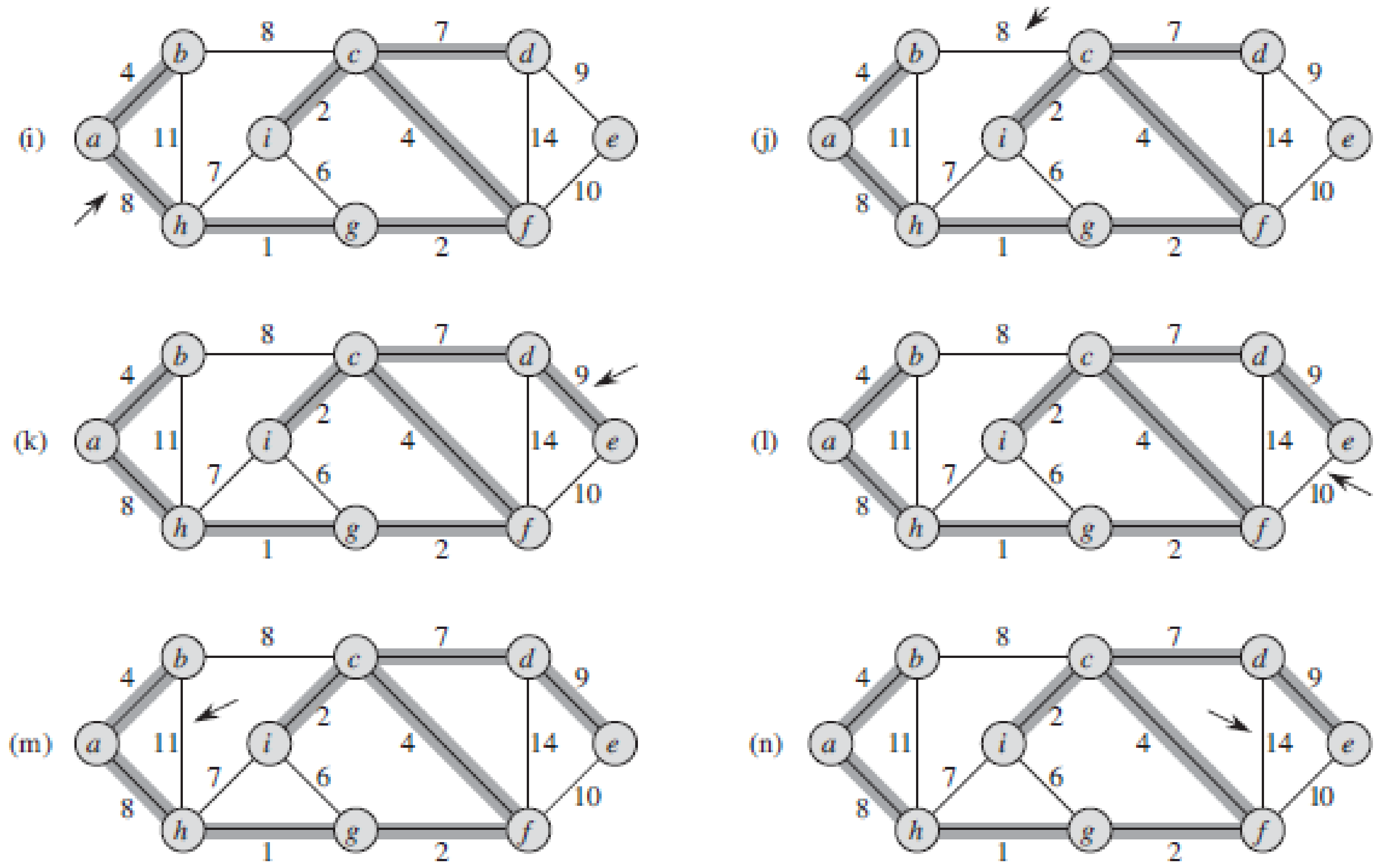


Figure 23.4, continued Further steps in the execution of Kruskal's algorithm.

Analysis

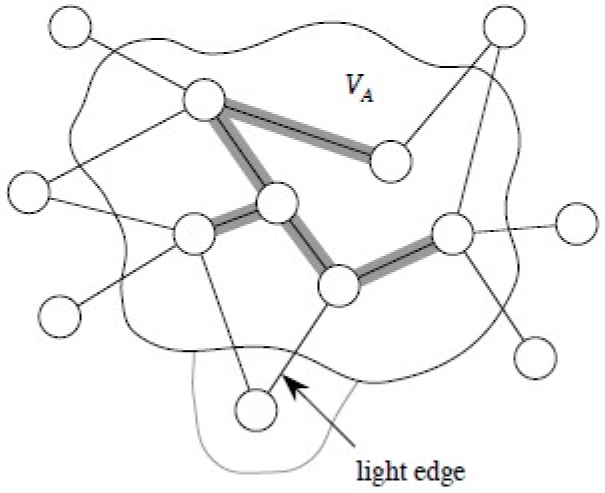
Initialize A:	$O(1)$
First **for** loop:	$\|V\|$ MAKE-SETS
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETS and UNIONS

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 21, that uses union by rank and path compression:

  $O((V + E)\,\alpha(V)) + O(E \lg E)$.
- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Prim's algorithm

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge crossing cut $(V_A, V - V_A)$, where $V_A =$ vertices that A is incident on. Add this edge to A.



[Edges of A are shaded.]

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex in $V - V_A$.
- Key of v is minimum weight of any edge (u, v), where $u \in V_A$.
- Then the vertex returned by EXTRACT-MIN is v such that there exists $u \in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
- Key of v is ∞ if v is not adjacent to any vertices in V_A.

The edges of A will form a rooted tree with root r:

- r is given as an input to the algorithm, but it can be any vertex.
- Each vertex knows its parent in the tree by the attribute $\pi[v]$ = parent of v. $\pi[v] =$ NIL if $v = r$ or v has no parent.
- As algorithm progresses, $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
- At termination, $V_A = V \Rightarrow Q = \emptyset$, so MST is $A = \{(v, \pi[v]) : v \in V - \{r\}\}$.

```
PRIM($V, E, w, r$)
$Q \leftarrow \emptyset$
for each $u \in V$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
       INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)    $\triangleright$ $key[r] \leftarrow 0$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                 then $\pi[v] \leftarrow u$
                      DECREASE-KEY($Q, v, w(u, v)$)
```

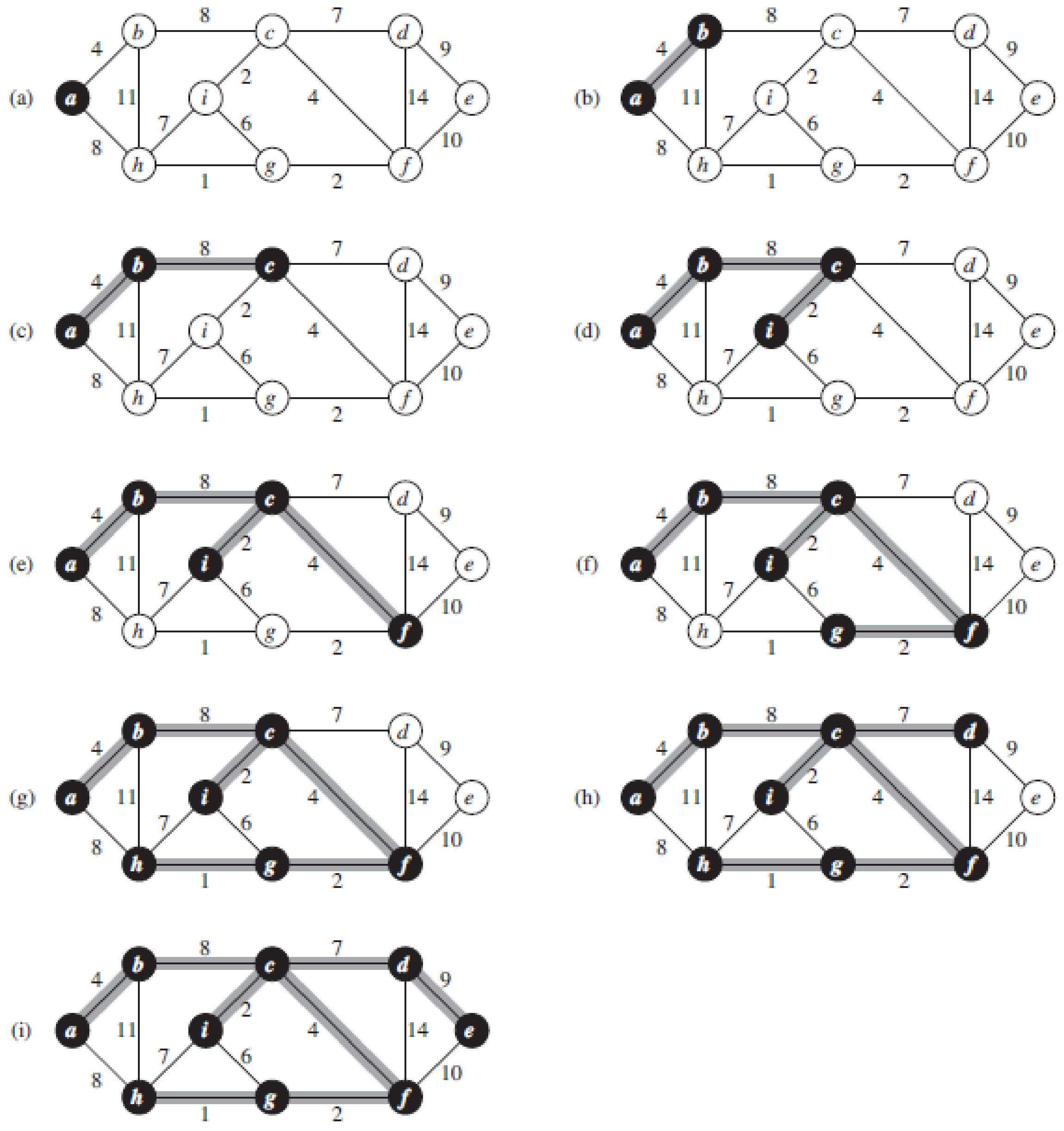


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Analysis

Depends on how the priority queue is implemented:

- 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:	$\lvert V\rvert$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
	$\leq \lvert E\rvert$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
Total:	$O(E \lg V)$

- Suppose 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 $O(V \lg V + E)$.

  In fact, there is a way to do DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, in Chapter 20.