Today we will be covering methods for finding minimum spanning trees.

Definition: A tree is a connected graph with no cycles.

Definition: Suppose we have a connected, undirected graph G. A spanning tree of G is a subset of the edges that connects all the vertices and has no cycles.

Definition: A minimum spanning tree is a spanning tree that has the lowest possible weight. That is, the sum of the weights of the edges must be as low as possible.

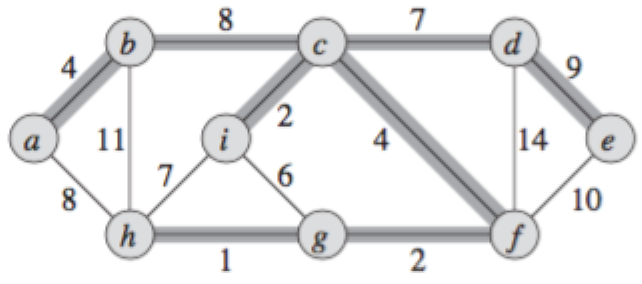


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

1 General approach for finding minimum spanning trees

Our two algorithms (Kruskal's and Prim's) both use a greedy strategy, where on each iteration we add one of the graph's edges to the minimum spanning tree. We do this until we have $n - 1$ edges. (Note that there may be other ways of constructing minimum spanning trees that do not take advantage of this approach.)

On each iteration, we claim that our current set of edges A is a subset of a minimum spanning tree. The goal is to prove that the new edge is a **safe edge**, meaning we can add it to the edge set and still have it be a subset of some minimum spanning tree.

Note that unless A already has $n - 1$ edges, we can always find a safe edge for A, because A is a subset of a minimum spanning tree, and we can theoretically use one of the other edges in the tree as a safe edge (we just might not know what those edges are yet).

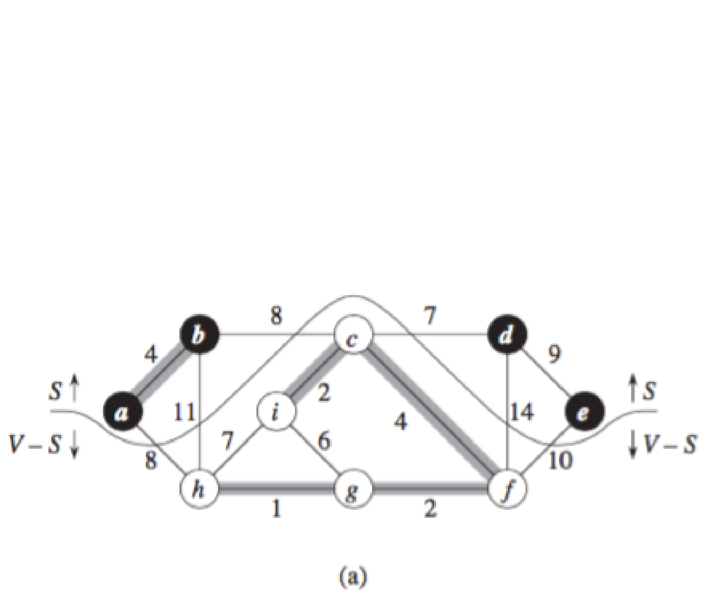
1.1 Definitions

Definition: A **cut** $(S, V - S)$ of an undirected graph is a partition of the set of vertices into the sets S and $V - S$.

Definition: A cut **respects** a set of edges A if no edge in A crosses the cut. That is, none of the edges have one vertex in S and the other vertex in $V - S$.

Definition: An edge is a **light edge** satisfying a property if it has the smallest weight out of all edges that satisfy that property.

Specifically, an edge is a light edge crossing a cut if it has the smallest weight out of all edges that cross the cut.



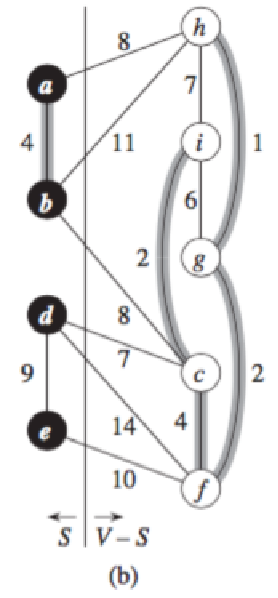


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** Black vertices are in the set S, and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. 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.

1.2 Rule for recognizing safe edges

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

Corollary: Consider the connected components in the forest formed by the vertex set and the edges A. Any light edge connecting two components in the forest is a safe edge for A.

The corollary follows because we can create a cut $(S, V - S)$, where S is just one component, and $V - S$ is the rest of the graph. This cut respects A, and (u, v) is a light edge for the cut.

You may visualize our minimum spanning tree algorithms as follows. We slowly build up our spanning tree one edge at a time. At the start of our algorithm, our set A has no edges, and the connected components in the graph $G_A = (V, A)$ are just disconnected vertices. At any point in the algorithm, all of the components in this graph are trees (since if they had cycles, they could not be subsets of a spanning tree). Any safe edge for A must merge two trees in G_A (because otherwise it would create a cycle).

Our algorithms must iterate $n-1$ times, since there must be exactly $n-1$ edges in a spanning tree.

1.2.1 Proof of theorem

We will use a cut-and-paste argument (pay attention since this sort of argument is used a lot).

Let T be a minimum spanning tree that includes A, and suppose (u, v) is a light edge crossing the cut $(S, V - S)$. If T contains the edge (u, v), we are done, so let's assume T does not contain (u, v). Then we construct another minimum spanning tree T' as follows:

Recall that the minimum spanning tree T must connect all the vertices in the graph, so there must be a path p from u to v that lies entirely in T, and does not include (u, v). We can combine this path with the edge (u, v) to form a cycle.

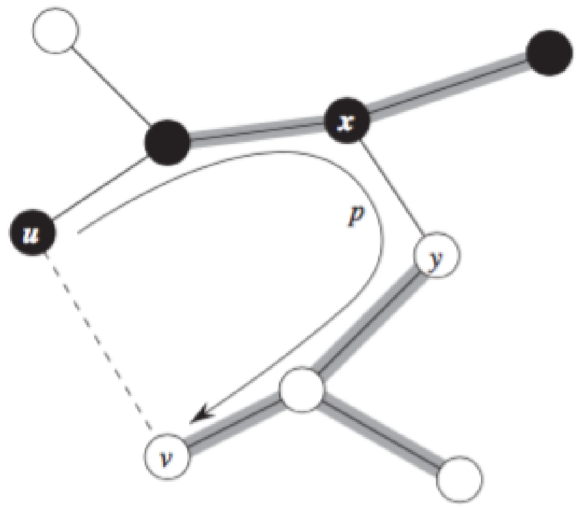


Figure 23.3 The proof of Theorem 23.1. Black vertices are in S, and white vertices are in $V - S$. The edges in the minimum spanning tree T are shown, but the edges in the graph G are not. The edges in A are shaded, and (u, v) is a light edge crossing the cut $(S, V - S)$. The edge (x, y) is an edge on the unique simple path p from u to v in T. To form a minimum spanning tree T' that contains (u, v), remove the edge (x, y) from T and add the edge (u, v).

Since u and v are on opposite sides of the cut, at least one of the edges on the path p must cross the cut. We call this edge (x, y). (Observe that (x, y) is not in A, since the cut respects A.)

Now we can replace (x, y) with (u, v) to form a new tree, $T' = T - \{(x, y)\} \cup \{(u, v)\}$. We show that T' is a minimum spanning tree.

Observe that T' is a spanning tree, since there are still $n-1$ edges, and all of the vertices are still connected to each other, since any path that would have used the edge (x, y) can use the path through (u, v) instead. Furthermore, T' has smaller total weight than T, because (u, v) is a light edge crossing the cut, and (x, y) also crosses the cut. So $w(u, v) \leq w(x, y)$, and $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. Since T is a minimum spanning tree, T' must also be a minimum spanning tree.

Finally, we show that (u, v) is a safe edge for A. Since $A \subseteq T$ and $(x, y) \notin A$, we have $A \subseteq T'$. So $A \cup \{(u, v)\} \subseteq T'$, and since T' is a minimum spanning tree, (u, v) is safe for A.

2 Prim's algorithm

2.0.1 Idea

At any point, the graph G_A is composed of a single tree and a bunch of isolated vertices. Every step adds the smallest possible edge that connects A to an isolated vertex. (Note that this is always a safe edge for A.)

To get the smallest possible edge, we can use a min-priority queue, like in Dijkstra's algorithm. The queue contains all the vertices that aren't in the tree, and each vertex's key is the minimum weight of any edge connecting that vertex to a vertex in the tree. (If there is no such edge, the key is ∞.)

Then to grow the tree, we extract the minimum vertex u from the queue (adding the edge between u and its parent, to be defined later). Then after putting that edge in the tree, we must decrease the keys of u's neighbors, because now there are additional edges connecting those neighbors to vertices in the tree. If we decrease the key of a neighbor, then the best way to get to that neighbor is through u, so we set the neighbor's parent to u.

2.0.2 Algorithm

```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi$ = NIL
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

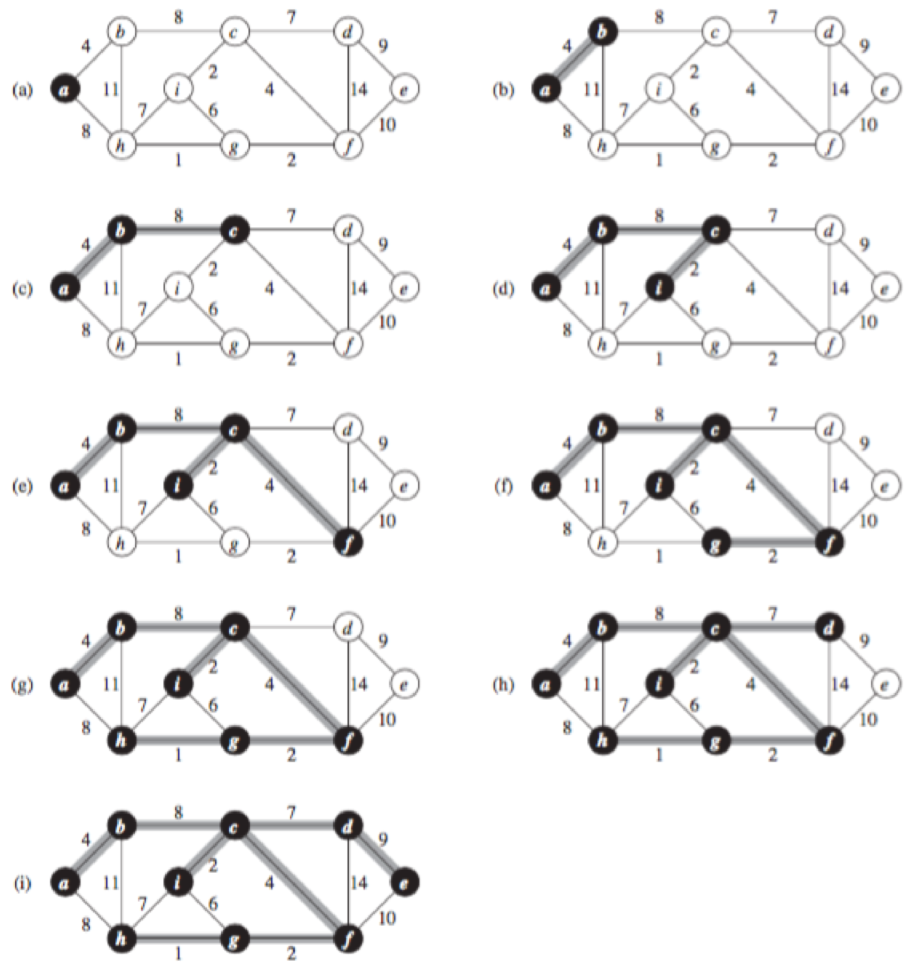


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added 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.

2.0.3 Runtime

The runtime depends on how we implement the priority queue. If we use an ordinary heap (like the one described in class), we use one call to `BuildMinHeap`, which takes $O(n)$ time. There are n calls to `ExtractMin`, each of which takes $O(\log n)$ time, so those calls take $O(n \log n)$ time total. Furthermore, there are m calls to `DecreaseKey` (across all iterations of the for loop, so those calls take $O(m \log n)$. Because the original graph is connected, $m \geq n - 1$, so the runtime is $O(n \log n + m \log n) = O(m \log n)$.

If we use Fibonacci heaps, `Insert` takes $O(1)$ time, `ExtractMin` takes $O(\log n)$ amortized time and `DecreaseKey` takes $O(1)$ amortized time. So in total this takes $O(m + n \log n)$.

3 Disjoint set data structures

Another minimum spanning tree algorithm, Kruskal's algorithm, uses disjoint-set data structures. A disjoint-set data structure maintains a collection $\{S_1, \dots, S_k\}$ of disjoint sets. Each set is identified by a "representative," which is some member of the set.

Disjoint-set data structures must support the following operations:

- `MakeSet(x)`: Create a new set containing x and only x. (x should not be in any other set, since the sets are disjoint.)
- `Union(x, y)`: Create a new set that is the union of the sets containing x and y. Destroy the old sets.
- `FindSet(x)`: Return a pointer to the representative of the set containing x.

3.1 Example

One application of disjoint-set data structures is finding the connected components of an undirected graph.

```
CONNECTED-COMPONENTS($G$)
    for each vertex $v \in G.V$
        MAKE-SET($v$)
    for each edge $(u, v) \in G.E$
        if FIND-SET($u$) $\neq$ FIND-SET($v$)
            UNION($u, v$)
```

```
SAME-COMPONENT($u, v$)
    if FIND-SET($u$) == FIND-SET($v$)
        return TRUE
    else return FALSE
```

3.2 Simple implementation

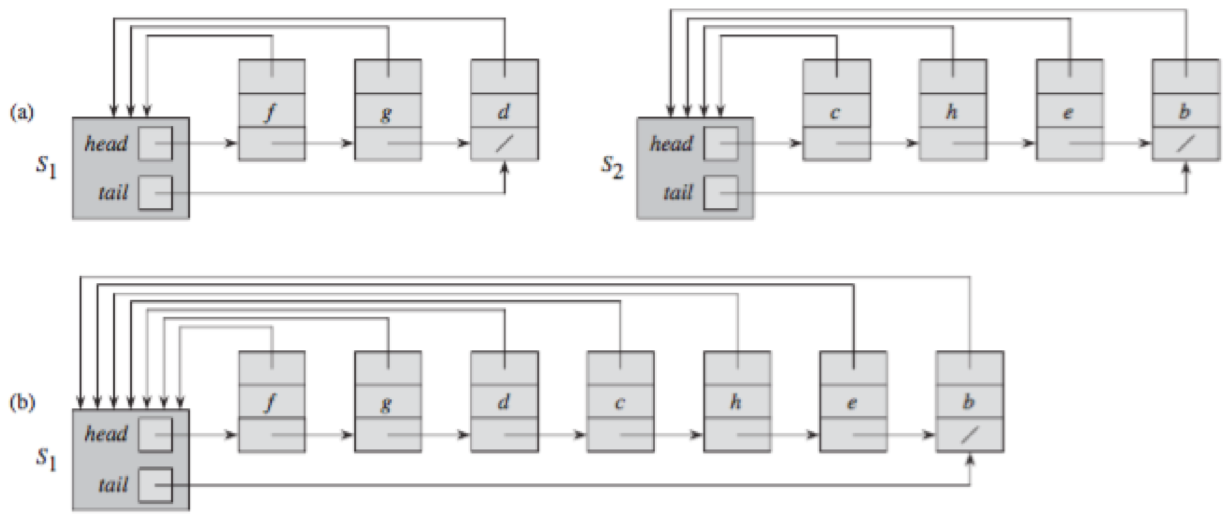


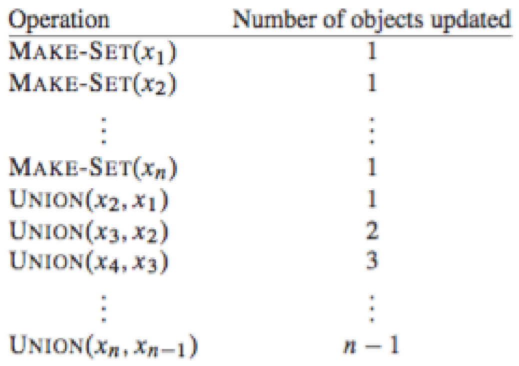
Figure 21.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

Represent each set by its own linked list. Each object in a linked list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. The representative of the set is the object that "head" points to.

`MakeSet(x)` creates a new linked list whose only object is x, which runs in $O(1)$ time. `FindSet(x)` also takes $O(1)$ time, since you just have to follow the pointer from x back to the set object and then return the member in the object that "head" points to.

`Union(x, y)` may take considerably longer. We perform `Union(x, y)` by appending y's list onto the end of x's list, and the representative of x's list becomes the representative of the resulting set. We use the "tail" pointer for x's list to quickly find where to append y's list. The slow step is that for each of the objects originally in y's list, we have to update their set pointers to point to x's set object. This takes time linear in the length of y's list.

We can't amortize away the slowness either – consider the sequence of $2n - 1$ operations where we create n sets and then unionize all of them in the "wrong order":



Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n - 1$

Figure 21.3 A sequence of $2n - 1$ operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

This sequence takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation.

Note that we can shorten the runtime by always appending the shorter list onto the longer list. Then you can show that a sequence of m `MakeSet`, `Union`, and `FindSet` operations, n of which are `MakeSet` operations, takes $O(m + n \log n)$ time. We won't cover this right now but the proof is in the textbook.

The best known runtime for the union-find operations is amortized $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. Note that $\alpha(n) \leq 4$ for all n less than the number of atoms in the universe.

4 Kruskal's algorithm

4.0.1 Algorithm

At any point, the connected component forest is composed of several trees. Choose the lightest edge that connects two of the trees, and add it to the tree.

It is easiest to do this if we sort all of the edges by weight in advance and consider them in order. Then use a disjoint-set data structure to test whether an edge connects two components, and join the components if so.

```
MST-KRUSKAL($G, w$)
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET($v$)
sort the edges of $G.E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET($u$) $\neq$ FIND-SET($v$)
        $A = A \cup \{(u, v)\}$
        UNION($u, v$)
return $A$
```

You should convince yourself that one pass through Kruskal's algorithm suffices to add exactly $n-1$ edges to the graph. First of all, every time we add an edge, it must be a safe edge, so the for loop must only add edges belonging to the spanning tree. However, if we went back through the for loop again, we would not be able to add any more edges, because every edge was either added already, or we already decided that adding it would form a cycle. So the for loop must add exactly $n-1$ edges.

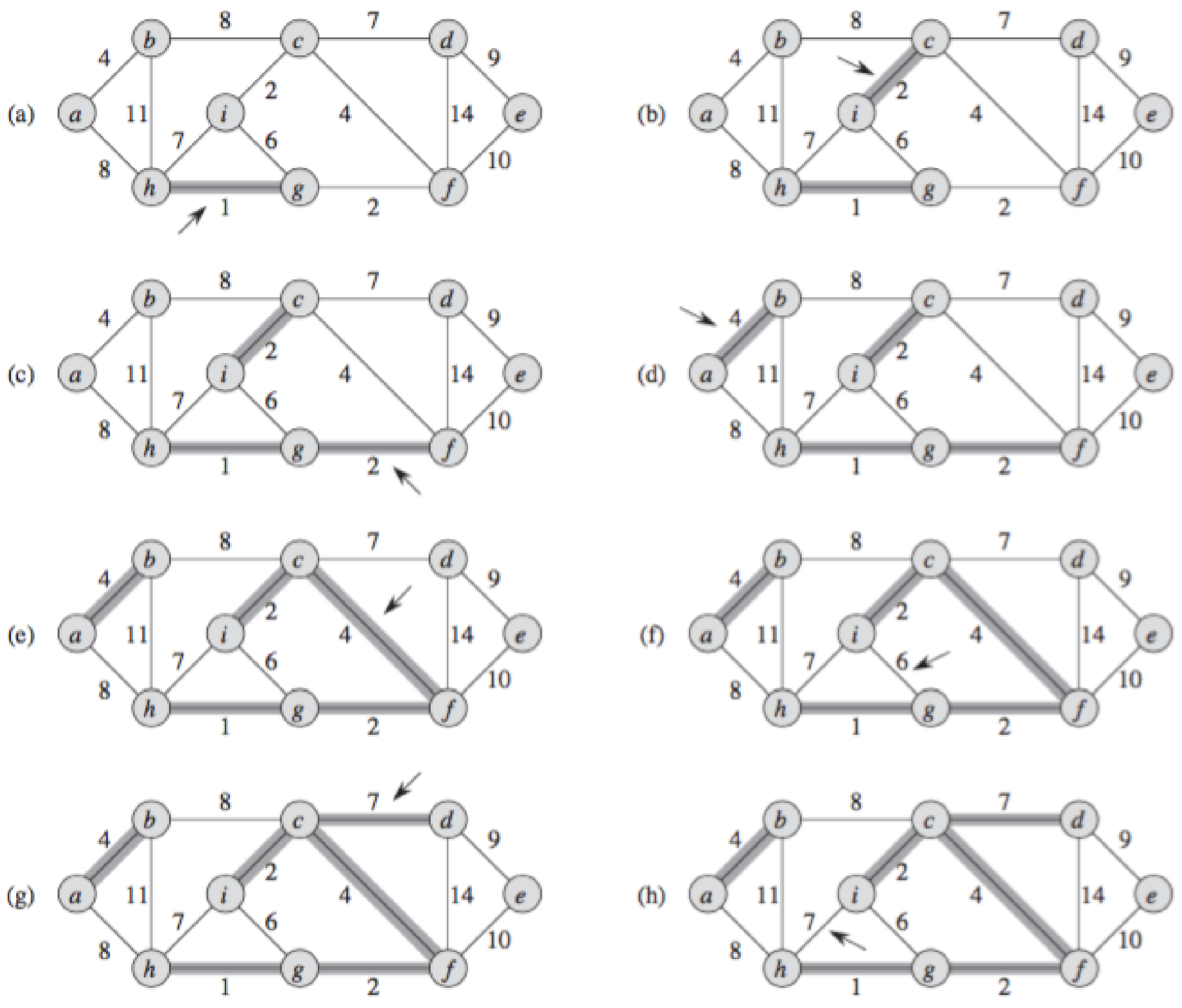


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees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

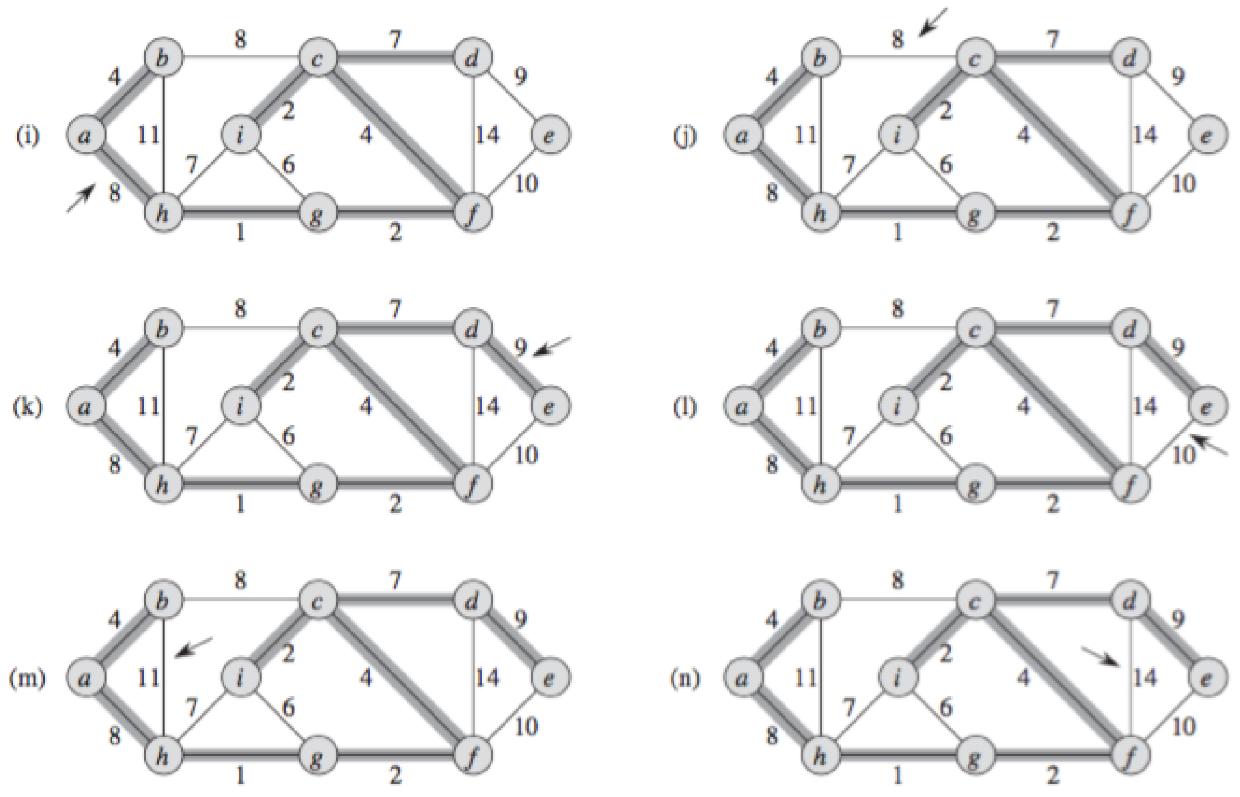


Figure 23.4, continued Further steps in the execution of Kruskal's algorithm.

4.0.2 Runtime

The runtime of Kruskal's algorithm depends on how we implement the disjoint set data structure. Observe that we make n calls to `MakeSet`, $2m$ calls to `FindSet`, and $n-1$ calls to `Union`. With the linked list implementation, these would take $O(n^2)$, since `Union` takes $O(n)$ time and the other operations take $O(1)$ time. In addition, sorting the list takes $O(m \log m) = O(m \log n)$ time.

With the fastest implementation, the union-find operations take $O((n+m)\alpha(n))$, and sorting the list still takes $O(m \log n)$ time. Since $\alpha(n) = O(\log n)$ (and is actually far less than $O(\log n)$), we can represent the runtime of the algorithm by $O(n \log n + m \log n)$. Since G is connected, $m \geq n-1$, so the runtime can be represented as $O(m \log n)$.