Combining these estimates we have the following theorem:

THEOREM 5.2. *If we use self-adjusting binary trees to represent solid paths, a sequence of m tree operations including n maketree operations requires* $O(m \log n)$ time.

5.4. Remarks. We can add other operations to our repertoire of tree operations without sacrificing the bound of $O(\log n)$ amortized time per operation. Perhaps the most interesting is evert (v), which turns the tree containing vertex v "inside out" by making v the root. (See Fig. 5.7.) When this operation is included the data structure is powerful enough to handle problems requiring linking and cutting of free (unrooted) trees; we root each tree at an arbitrary vertex and reroot as necessary using evert. We can associate costs with the edges of the trees rather than with the vertices. If worst-case rather than amortized running time is important, we can modify the data structure so that each tree operation takes $O(\log n)$ time in the worst case; the resulting structure uses biased search trees [1] in place of self-adjusting trees and is more complicated than the one presented here. Details of all these results may be found in Sleator and Tarjan [4].

References

[1] S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased search trees,* SIAM J. Comput., submitted.

[2] Z. GALIL AND A. NAAMAD, *An* $O(E V \log^2 V)$ *algorithm for the maximal flow problem,* J. Comput. System Sci., 21 (1980), pp. 203–217.

[3] Y. SHILOACH, *An* $O(n \cdot I \log^2 I)$ *maximum-flow algorithm,* Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, 1978.

[4] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees,* J. Comput. System Sci., 24 (1983), to appear; also in Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.

CHAPTER 6

Minimum Spanning Trees

6.1. The greedy method. In the last half of this book we shall us structures developed in the first half to solve four classical problems i optimization. A *network* is a graph, either undirected or directed, eacl edges has an associated real number. The object of a network optimizatic is to find a subgraph of a given network that has certain specified properti minimizes (or maximizes) some function of the edge numbers. In this c shall study one of the simplest problems of network optimization, the *spanning tree problem:* given a connected undirected graph each of whos a real-valued *cost,* find a spanning tree of the graph whose total ed minimum. We shall denote the cost of an edge $e = \{v, w\}$ by *cost* (e) o extra brackets, by *cost* (v, w).

To solve this problem we use a simple incremental technique called t *method:* we build up a minimum spanning tree edge by edge, including a small edges and excluding appropriate large ones, until at last we have a spanning tree. We shall formulate the greedy method in a way general include all known efficient algorithms for the problem; this leaves us roo the details of the method to make it as efficient as possible.

To make the greedy method concrete, we shall describe it as an edg process. Initially all edges of the graph are uncolored. We color one edge either blue (accepted) or red (rejected). To color edges we use two maintain the following *color invariant:*

> There is a minimum spanning tree containing all of the blue edges ar of the red ones.

The color invariant implies that when all the edges are colored, the blue on minimum spanning tree.

To formulate the coloring rules we need the notion of a cut. A *cut* in a g $[V, E]$ is a partition of the vertex set V into two parts, X and $\bar{X} = V - X$. *crosses* the cut if it is incident to X (one end is in X and the other in $\bar{X}$). sometimes regard a cut as consisting of the edges crossing it, but it is imp remember that technically a cut is a vertex partition.

The greedy method uses two coloring rules:

Blue rule. Select a cut that no blue edges cross. Among the uncolor crossing the cut, select one of minimum cost and color it blue.

Red rule. Select a simple cycle containing no red edges. Among the u edges on the cycle, select one of maximum cost and color it red.

The method is nondeterministic: we are free to apply either rule at any arbitrary order, until all edges are colored.

THEOREM 6.1. *The greedy method colors all the edges of any connected graph and maintains the color invariant.*

Proof. First we prove that the greedy method maintains the color invariant. Initially no edges are colored, and any minimum spanning tree satisfies the invariant; there is at least one minimum spanning tree since the graph is connected. Suppose the invariant is true before an application of the blue rule. Let e be the edge colored blue by the rule and let T be a minimum spanning tree satisfying the invariant before e is colored. If e is in T, then T satisfies the invariant after e is colored. If e is not in T, consider the cut X, $\overline{X}$ to which the blue rule is applied to color e. There is a path in T joining the ends of e, and at least one edge on this path, say e', crosses the cut. By the invariant no edge of T is red, and the blue rule implies that e' is uncolored and $cost\ (e') \geq cost\ (e)$. Adding e to T and deleting e' produces a minimum spanning tree T' that satisfies the invariant after e is colored.

The following similar argument shows that the red rule maintains the invariant. Let e be an edge colored red by the rule and let T be a minimum spanning tree satisfying the invariant before e is colored. If e is not in T, then T satisfies the invariant after e is colored. Suppose e is in T. Deleting e from T divides T into two trees that partition the vertices of G; e has one end in each tree. The cycle to which the red rule was applied to color e must have at least one other edge, say e', with an end in each tree. Since e' is not in T, the invariant and the red rule imply that e' is uncolored and $cost\ (e') \leq cost\ (e)$. Adding e' to T and deleting e produces a new minimum spanning tree T' that satisfies the invariant after e is colored. We conclude that the greedy method maintains the color invariant.

Now suppose that the method stops early; that is, there is some uncolored edge e but no rule applies. By the invariant, the blue edges form a forest, consisting of a set of trees that we shall call *blue trees*. If both ends of e are in the same blue tree, the red rule applies to the cycle formed by e and the path of blue edges joining the ends of e. If the ends of e are in different blue trees, the blue rule applies to the cut one of whose parts is the vertex set of a blue tree containing one end of e. Thus an uncolored edge guarantees the applicability of some rule, and the greedy method colors all the edges. □

The greedy method applies to a wide variety of problems besides the minimum spanning tree problem. The proper general setting of the method is matroid theory [19]. In the special case of finding minimum spanning trees, especially efficient implementations are possible. In §6.2 we shall examine three well-known minimum spanning tree algorithms that are versions of the greedy method. In §6.3 we shall develop a relatively new algorithm that is the fastest yet known for sparse graphs.

6.2. Three classical algorithms. The minimum spanning tree problem seems to be the first network optimization problem ever studied; its history dates at least to 1926. Graham and Hell [14] have written an excellent historical survey of the problem. Borůvka [4] produced the first fully realized minimum spanning tree algorithm. The same algorithm was rediscovered by Choquet [7], Lukaszewicz et al. [20] and Sollin [1].

To understand Borůvka's algorithm, we need to study the behavior of the greedy

method. Recall our definition in the proof of Theorem 6.1 that a *blue tree* the forest defined by the set of blue edges. Initially there are n blue consisting of one vertex and no edges. Coloring an edge blue combines tw into one. The greedy method combines blue trees two at a time until final blue tree, a minimum spanning tree, remains.

Borůvka's algorithm begins with the initial set of n blue trees and r following step until there is only one blue tree (see Fig. 6.1).

COLORING STEP (Borůvka). For every blue tree T, select a minimum incident to T. Color all selected edges blue.

Since every edge has two ends, an edge can be selected twice in execution of the coloring step, once for each end. Such an edge is only col Borůvka's algorithm is guaranteed to work correctly only if all the edg distinct, in which case the edges can be ordered so that the blue rule will one at a time. If the edge costs are nondistinct, we can break ties by numbers to the edges (or vertices) and ordering edges lexicographically b number (or cost and end numbers). Borůvka's algorithm, though old, related to the new fast algorithm we shall study in §6.3. There are sever implement the algorithm, which we shall not discuss here except to not method is well suited for parallel computation.

The second and most recently discovered of the classical algorithms

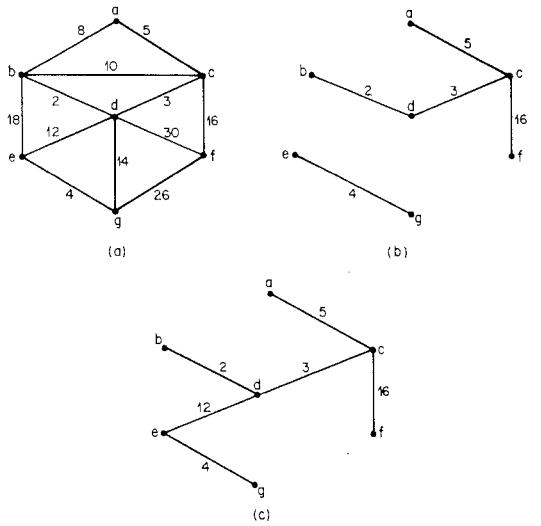

FIG. 6.1. *Execution of Borůvka's algorithm.* (a) *Input graph.* (b) *First step. Vertex a s* $\{a, c\}$, *b and d select* $\{b, d\}$, *c selects* $\{c, d\}$, *e and g select* $\{e, g\}$ *and f selects* $\{c, f\}$. (c) *Second blue trees select* $\{d, e\}$.

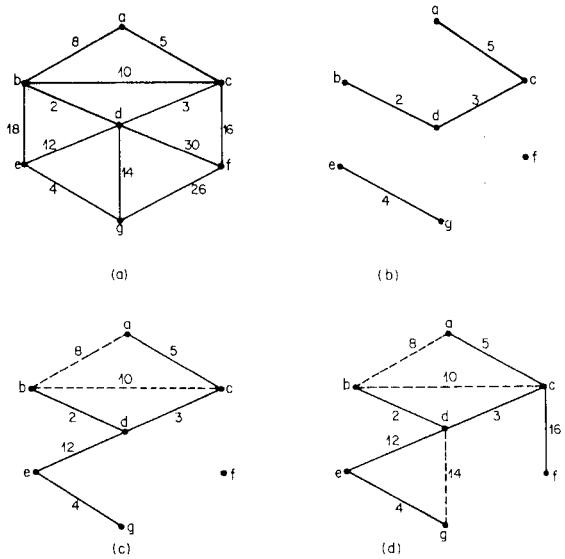

FIG. 6.2. *Execution of Kruskal's algorithm. Red edges are dashed.* (a) *Input graph.* (b) *After the first four steps. Edges* $\{b, d\}$, $\{c, d\}$, $\{e, g\}$, *and* $\{a, c\}$ *are colored blue.* (c) *After the next three steps. Edges* $\{a, b\}$ *and* $\{b, c\}$ *are colored red;* $\{d, e\}$ *is colored blue.* (d) *After two more steps. Edge* $\{d, g\}$ *is colored red and* $\{c, f\}$ *is colored blue.*

Kruskal [18]. Kruskal's algorithm consists of applying the following step to the edges in nondecreasing order by cost (see Fig. 6.2):

COLORING STEP (Kruskal). If the current edge e has both ends in the same blue tree, color it red; otherwise color it blue.

To implement Kruskal's algorithm, we must solve two problems: ordering the edges by cost and representing the blue trees so that we can test whether two vertices are in the same blue tree. The simplest way to solve the former problem is to sort the edges by cost in a preprocessing step. The latter problem is a version of the disjoint set union problem discussed in Chapter 2, and we can use the operations of makeset, find and link to maintain the vertex sets of the blue trees.

The following program implements Kruskal's algorithm using these ideas. The program accepts as input sets of the vertices and edges in the graph and returns a set of the edges in a minimum spanning tree.

```
set function minspantree (set vertices, edges);
  set blue;
  blue := { };
  edges := sort edges by cost;
  for v ∈ vertices → makeset (v) rof;
```



```
  for {v, w} ∈ edges: find (v) ≠ find (w) →
    link (find (v), find (w)); blue := blue ∪ {{v, w}}
  rof;
  return blue
end minspantree;
```

The sorting step in the implementation requires $O(m \log n)$ tim minimum spanning tree given a sorted edge list requires $O(m, \alpha(m, n))$ results of Chapter 2. Thus the total running time is $O(m \log n)$. This best in situations where the edges are given in sorted order or can be so instance when the costs are small integers and radix sorting can be u cases the running time is $O(m\alpha(m, n))$.

The third classical minimum spanning tree algorithm was discover [16] and rediscovered by Prim [22] and Dijkstra [9]; it is commonl Prim's algorithm. The algorithm uses an arbitrary starting vertex s an repeating the following step $n - 1$ times (see Fig. 6.3):

COLORING STEP (Prim). Let T be the blue tree containing s. Select a cost edge incident to T and color it blue.

Prim's algorithm is a kind of "shortest-first" search. (We shall study a of shortest-first search in Chapter 7.) Since the algorithm builds only on

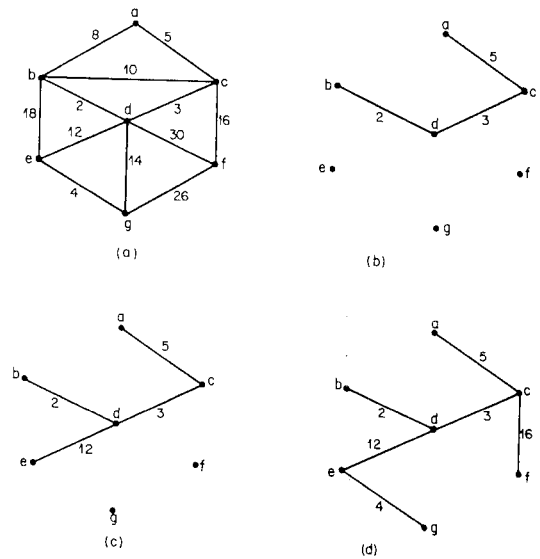

FIG. 6.3. *Execution of Prim's algorithm.* (a) *Input graph. Vertex a is the start vertex.* (b steps. *Edges* $\{a, c\}$, $\{c, d\}$, *and* $\{d, b\}$ *become blue.* (c) *After four steps. Edge* $\{d, e\}$ *becomes bl six steps. Edges* $\{e, g\}$ *and* $\{c, f\}$ *become blue.*

blue tree, we can implement the method without using a data structure for disjoint set union. We need only a single heap.

Although Jarník gave no detailed implementation, both Prim and Dijkstra did. Their idea is as follows: Let T be the blue tree containing s. We say v *borders* T if v is not in T but some edge is incident to both v and T. With each vertex v bordering T we associate a *light blue* edge e that is a minimum-cost edge incident to v and T. (Dijkstra [10] calls these edges "ultraviolet.") The light blue edges are candidates to become blue; the blue and light blue edges together form a tree spanning T and its bordering vertices. To carry out a coloring step, we select a light blue edge of minimum cost and color it blue. This adds a new vertex, say v, to T. We complete the step by examining each edge $\{v, w\}$ incident to v. If w is not in T and has no incident light blue edge, we color $\{v, w\}$ light blue. If w is not in T but w has an incident light blue edge, say e, of greater cost than $\{v, w\}$, we color e red and color $\{v, w\}$ light blue. (See Fig. 6.4.)

Prim's algorithm implemented as described above has a running time of $O(n^2)$, $O(n)$ per coloring step. We can make the method faster on sparse graphs by maintaining a heap (see Chapter 3) of the vertices bordering T and using the appropriate heap operations to carry out each coloring step. The key of a vertex in the heap is the cost of the incident light blue edge.

The program below implements this method. As input, the program needs a list of the vertices in the graph and the start vertex. The program assumes that, for each

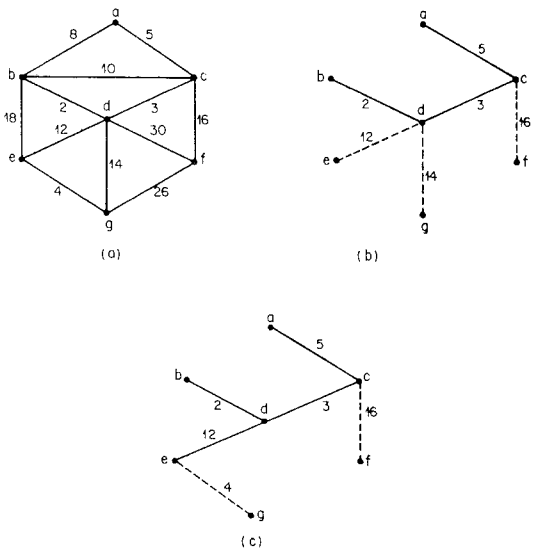

FIG. 6.4. *Efficient implementation of Prim's algorithm. Light blue edges are dashed.* (a) *Input graph. Vertex a is the start vertex.* (b) *After three steps.* (c) *After four steps. Edge* $\{d, e\}$ *becomes blue,* $\{e, g\}$ *becomes light blue, and* $\{d, g\}$ *and* $\{b, e\}$ *become red.*

vertex v, *edges* (v) is the set of edges incident to v. For each vertex v it r incident edge *blue* (v) and a real number *key* (v) defined as follows: If v T nor borders T, *blue* (v) is undefined and *key* (v) is infinity. If v border is the light blue edge incident to v and *key* (v) is the cost of the edge. *blue* (v) is the blue edge whose coloring caused v to be added to T an minus infinity. When the program halts, the edges *blue* (v) for v minimum spanning tree.

The program is designed to use a d-heap (see §3.2) as proposed by Jo In particular it uses the operation siftup (v) to restore the heap after de key of vertex v (by changing its incident light blue edge). The program i able to test vertices for heap membership. The *key* fields can be used for th since if the code below is written appropriately, a vertex v is in the heap if *key* (v) is finite. The heap indices (see § 3.2) also provides this infor

```
procedure minspantree (set vertices, vertex s);
  vertex v; heap h;
  for v ∈ vertices → key (v) := ∞ rof;
  h := makeheap ({ });
  v := s;
  do v ≠ null →
    key (v) := −∞;
    for {v, w} ∈ edges (v): cost (v, w) < key (w) →
      key (w) := cost (v, w);
      blue (w) := {v, w};
      if w ∉ h → insert (w, h) | w ∈ h → siftup (w, h⁻¹ (w), h) fi
    rof;
    v := deletemin (h)
  od
end minspantree;
```

Remark. It is possible to use a real-valued function to define *key*, thus space of one field per vertex. □

The running time of this implementation is dominated by the heap ope which there are $n - 1$ deletemin operations, $n - 1$ insert operations, a $m - n + 1$ siftup operations. By the analysis of §3.2, the total runni $O(n\, d \log_d n + m \log_d n)$. If we choose $d = \lceil 2 + m/n \rceil$, we obtain a tim $O(m \log_{(2+m/n)} n)$. If $m = \Omega(n^{1+\varepsilon})$ for some positive ε, the running time i Thus Prim's algorithm with Johnson's implementation is well suited graphs, and the method is asymptotically worse than Kruskal's only if the presorted.

6.3. The round robin algorithm. All three of the algorithms presented i mainly the blue rule. (It is also possible to find minimum spanning t mainly the red rule, but such methods seem to be less efficient.) The mo blue rule algorithm consists of beginning with the initial blue trees and rep following step $n - 1$ times: