COLORING STEP (round robin). Select a blue tree. Find a minimum-cost edge incident to this tree and color it blue.

Kruskal's algorithm builds blue trees in an irregular fashion dictated by the edge costs, Prim's algorithm builds only one nontrivial blue tree, and Borůvka's algorithm builds blue trees uniformly throughout the graph. By judiciously implementing a Borůvka-like algorithm using the appropriate data structures, we can obtain a method that is faster on sparse graphs than is any of the classical algorithms. Yao [29] was the first to propose such a method. His algorithm runs in $O(m \log \log n)$ time but needs a linear-time selection algorithm [2], [24] and is thus not very practical. We shall describe a similar but more practical $O(m \log \log n)$-time algorithm proposed by Cheriton and Tarjan [6].

To implement the general blue rule algorithm, we need two data structures for each blue tree. First, we need a way to represent the set of vertices in each tree. Second, we need a heap of the edges with at least one end in the tree that are candidates for becoming blue; the cost of an edge is its key in the heap. To represent the vertex sets, we use the data structure of Chapter 2, as in our implementation of Kruskal's algorithm. To represent the edge heaps, we use the leftist heaps of §3.3 with lazy melding and lazy deletion; we declare an edge "deleted" if its ends are in the same blue tree. We never explicitly mark edges deleted; instead, they are deleted implicitly when blue trees are combined.

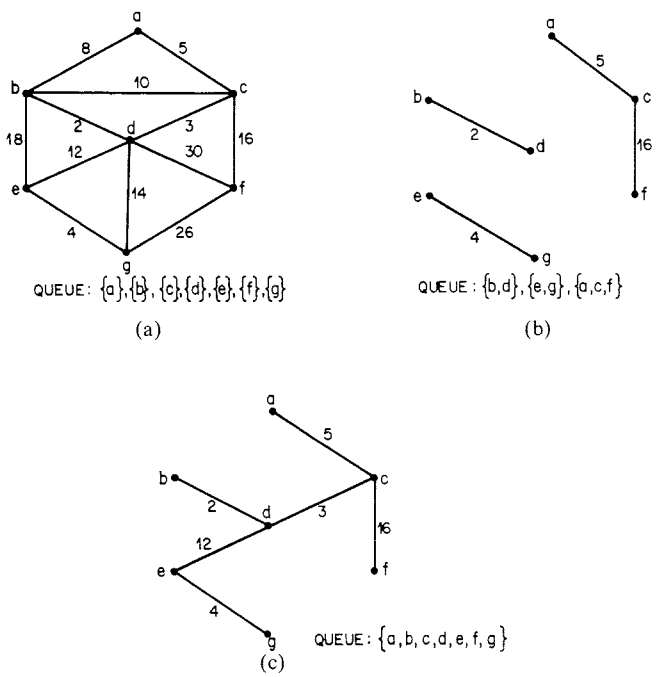

FIG. 6.5. *Execution of the round-robin algorithm.* (a) *Input graph. The queue contains the vertex sets of the blue trees.* (b) *After the initial pass through the queue. Edges* $\{a, c\}$, $\{b, d\}$, $\{e, g\}$, *and* $\{c, f\}$ *become blue.* (c) *After another pass. Edges* $\{c, d\}$ *and* $\{d, e\}$ *become blue.*

We also need a way to select blue trees for application of the blue rule. For this purpose we use a queue containing all the blue trees. To carry out a coloring step, we remove the first tree, say T_1, from the front of the queue and perform a findmin on its heap. We color blue the edge, say e, returned by the findmin. We remove from the queue the other tree, say T_2, incident to e. We update the vertex sets and edge heaps to reflect the combining of T_1 and T_2 and add the new blue tree to the rear of the queue. (See Fig. 6.5.) We call this method the *round robin algorithm.*

The following program implements this method. Input to the program is the set of vertices in the graph and, for each vertex v, the set *edges* (v) of incident edges. To represent each blue tree, the program uses its canonical vertex as defined by the disjoint set union algorithm (see §2.1). The queue for selecting blue trees consists of a list of canonical vertices. For each canonical vertex v, $h(v)$ is the edge heap of the blue tree containing v. The predicate deleted (e) returns **true** if the ends of e are in the same blue tree. The program returns a set of the edges in a minimum spanning tree.

```
set function minspantree (set vertices);
   set blue;
   map h;
   list queue;
   vertex v, w;
   blue := { };
   queue := [ ]
   for v ∈ vertices → makeset (v); h(v) := makeheap (edges (v));
                      queue := queue & [v] rof;
   do |queue| > 1 →
      {v, w} := findmin (h(queue (1)));
      blue := blue ∪ {{v, w}};
      queue := queue − {find (v), find (w)};
      h(link (find (v), find (w))) := meld (h(find (v)), h(find (w)));
      queue := queue & [find (v)]
   od;
   return blue
end minspantree;

predicate deleted (edge {v, w});
   return find (v) = find (w)
end deleted;

real function key (edge e);
   return cost (e)
end key;
```

Notes. Every edge $\{v, w\}$ is initially in two heaps, $h(v)$ and $h(w)$. When such an edge is colored blue, the corresponding link operation automatically deletes both copies from the heap associated with the new blue tree. Since arbitrary elements must be deleted from the queue (by the assignment "*queue* := *queue* − {find (v), find (w)}"), it is best to implement the queue as a doubly linked list or as an array of

vertices, each with an index giving its position in the array. (See §1.3.) Then the time for a deletion is $O(1)$. □

Let us analyze the running time of this algorithm. The running time is $O(m)$ not counting set and heap operations. The time for all the makeset and link operations is $O(n)$. If we assume that the time per find is $O(1)$, then the heap operations dominate the running time. We shall justify this assumption later.

To simplify the discussion we shall speak of the blue trees themselves rather than the canonical vertices as being on the queue. For $i = 1, 2, \cdots, n$ we define T_i to be the blue tree selected during the ith coloring step, m_i to be the number of edges in the heap associated with T_i when T_i is selected, and k_i to be the number of edges purged from this heap during the ith findmin operation. We divide the execution of the algorithm into *passes* as follows. Pass zero consists of the selection and processing of blue trees initially on the queue. For $j > 0$, pass j consists of the selection and processing of the trees added to but not deleted from the queue during pass $j - 1$.

LEMMA 6.1. *A blue tree on the queue during pass j contains at least 2^j vertices. Thus after at most $\lfloor \lg n \rfloor$ passes there is a single blue tree, and the algorithm stops.*

Proof. Immediate by induction on j, since for $j > 0$ a blue tree on the queue during pass j consists of a combination of two or more blue trees on the queue during pass $j - 1$. □

LEMMA 6.2.

$$\sum_{i=1}^{n-1} m_i \leq (2m + n - 1)\lfloor \lg n \rfloor .$$

Proof. Any two blue trees selected and processed during the same pass are vertex disjoint, since they are on the queue simultaneously. Thus the total size of all heaps processed during a single pass is at most $2m + n - 1$: each actual edge occurs in at most two heaps, and there are at most $n - 1$ dummy edges corresponding to lazy melds. (See §3.3.) The lemma follows from Lemma 6.1. □

Now we can bound the time for the heap operations. The time for all the makeheap operations is $O(m)$. The time for melds is $O(1)$ per meld, totalling $O(n)$ for all $n - 1$ melds. The time for the ith findmin is $O(k_i \max\{1, \log m_i/(k_i + 1)\})$. To estimate the total time for findmin operations, we divide them into two types: the ith findmin is *small* if $k_i \leq m_i/(\lg n)^2 - 1$ and *large* otherwise. The total time for small findmin operations is

$$O\left(\sum_{i=1}^{n-1} \frac{m_i}{(\log n)^2} \log m_i\right) = O\left(\sum_{i=1}^{n-1} \frac{m_i}{\log n}\right) = O(m)$$

by Lemma 6.2. The total time for large findmins is

$$O\left(\sum_{i=1}^{n-1} k_i \log \frac{m_i}{(m_i/(\log n)^2)}\right) = O\left(\sum_{i=1}^{n-1} k_i \log \log n\right) = O(m \log \log n),$$

since $\sum_{i=1}^{n-1} k_i \leq 2m + n - 1$, including the $n - 1$ dummy edges created by the melds.

THEOREM 6.2. *The round robin algorithm, implemented using leftist trees with lazy melding and lazy deletion, runs in $O(m \log \log n)$ time.*

Proof. The analysis above gives a time bound of $O(m \log \log n)$ with the finds counted at $O(1)$ time per find. This means that there are $O(m \log \log n)$ finds, which take $O((m \log \log n)\ \alpha(m \log \log n,\ n)) = O(m \log \log n)$ time (see §2.2). This gives the theorem. □

On sparse graphs, the round robin algorithm is asymptotically faster than any of the classical algorithms. On dense graphs it is slower by a factor of $O(\log \log n)$ than Prim's algorithm, although in practice the $O(m \log \log n)$ time bound is likely to be overly pessimistic. As a matter of theoretical interest, we can improve the running time of the round robin algorithm to $O(m \log \log_{(2+m/n)} n)$ time by "condensing" the graph at appropriate intervals, discarding from the edge heaps every edge with both ends in the same blue tree and all but a minimum cost edge between each pair of blue trees [6], [27]. The round robin algorithm with condensing is asymptotically as fast as any known minimum spanning tree algorithm, for any graph density.

6.4. Remarks. Not surprisingly, there are many results on special cases and variants of the minimum spanning tree problem. We conclude this chapter by mentioning some of the more interesting of such results. Other results may be found in the survey by Maffioli [21].

An on-line algorithm. Suppose we are presented with the edges of the graph one at a time in arbitrary order. We can build a minimum spanning tree on-line, as follows. We maintain a set of blue trees. To process an edge e, we color it blue. If this forms a cycle of blue edges, we discard a maximum-cost blue edge on the cycle. Using the data structures of Chapters 2 and 5 (the latter augmented to allow everting a tree), we can implement this algorithm to run on $O(m \log n)$ time.

Alternative cost functions. Minimum spanning trees are minimum with respect to any symmetric nondecreasing function of the edge costs [15].

Verification, sensitivity analysis, and updating. Three related problems are solvable in $O(m\alpha(m, n))$ time. Given a spanning tree, we can test whether it is minimum [26]. Given a minimum spanning tree, we can test by how much the cost of each edge can be increased or decreased without affecting the minimality of the tree [28]. Given a minimum spanning tree, we can find for each tree edge e a minimum-cost substitute edge e' such that, if e is deleted from the graph, replacing it in the old minimum spanning tree by e' produces a minimum spanning tree in the new graph [26].

Linear-time special cases. Let S be a class of graphs closed under condensation of an edge and such that every graph in S has $O(n)$ edges, where the constant depends on S but not on the graph. Then there is an $O(n)$-time algorithm to find minimum spanning trees for graphs in S. As examples of this result, we can find a minimum spanning tree in a planar graph in $O(n)$ time [6] and we can in $O(n)$ time update a minimum spanning tree if a new vertex and incident edges are added to a graph whose minimum spanning tree is known [23].