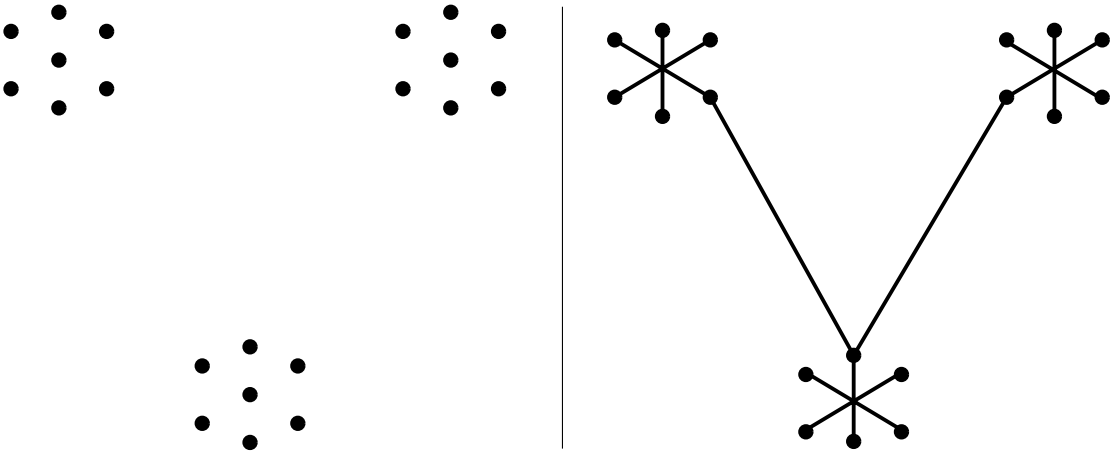

INPUT OUTPUT

15.3 Minimum Spanning Tree

Input description: A graph $G = (V, E)$ with weighted edges.

Problem description: The minimum weight subset of edges $E' \subset E$ that form a tree on V.

Discussion: The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are interested in minimum spanning trees, because the MST of a set of locations defines the wiring scheme that connects the sites using as little wire as possible. MST is the mother of all network design problems.

Minimum spanning trees prove important for several reasons:

- They can be computed quickly and easily, and create a sparse subgraph that reflects a lot about the original graph.
- They provide a way to identify clusters in sets of points. Deleting the long edges from an MST leaves connected components that define natural clusters in the data set, as shown in the output figure above.
- They can be used to give approximate solutions to hard problems such as Steiner tree and traveling salesman.
- As an educational tool, MST algorithms provide graphic evidence that greedy algorithms can give provably optimal solutions.

Three classical algorithms efficiently construct MSTs. Detailed implementations of two of them (Prim's and Kruskal's) are given with correctness arguments in Section 6.1 (page 192). The third somehow manages to be less well known despite being invented first and (arguably) being both easier to implement and more efficient.

The contenders are

- *Kruskal's algorithm* – Each vertex starts as a separate tree and these trees merges together by repeatedly adding the lowest cost edge that spans two distinct subtrees (i.e. , does not create a cycle).

```
Kruskal(G)
    Sort the edges in order of increasing weight
    count = 0
    while (count < n − 1) do
        get next edge (v, w)
        if (component (v) ≠ component(w))
            add to T
            component(v) = component(w)
```

  The "which component?" tests can be efficiently implemented using the union-find data structure (Section 12.5 (page 385)) to yield an $O(m \lg m)$ algorithm.

- *Prim's algorithm* – Starts with an arbitrary vertex v and "grows" a tree from it, repeatedly finding the lowest-cost edge that links some new vertex into this tree. During execution, we label each vertex as either in the tree, in the *fringe* (meaning there exists an edge from a tree vertex), or *unseen* (meaning the vertex is still more than one edge away from the tree).

```
Prim(G)
    Select an arbitrary vertex to start
    While (there are fringe vertices)
        select minimum-weight edge between tree and fringe
        add the selected edge and vertex to the tree
        update the cost to all affected fringe vertices
```

  This creates a spanning tree for any connected graph, since no cycle can be introduced via edges between tree and fringe vertices. That it is in fact a tree of minimum weight can be proven by contradiction. With simple data structures, Prim's algorithm can be implemented in $O(n^2)$ time.

- *Boruvka's algorithm* – Rests on the observation that the lowest-weight edge incident on each vertex must be in the minimum spanning tree. The union of these edges will result in a spanning forest of at most $n/2$ trees. Now for each of these trees T, select the edge (x, y) of lowest weight such that $x \in T$ and

$y \notin T$. Each of these edges must again be in an MST, and the union again results in a spanning forest with at most half as many trees as before:

```
Boruvka(G)
    Initialize spanning forest F to n single-vertex trees
        While (F has more than one tree)
            for each T in F, find the smallest edge from T to G − T
            add all selected edges to F, thus merging pairs of trees
```

The number of trees are at least halved in each round, so we get the MST after at most $\lg n$ iterations, each of which takes linear time. This gives an $O(m \log n)$ algorithm without using any fancy data structures.

MST is only one of several spanning tree problems that arise in practice. The following questions will help you sort your way through them:

- *Are the weights of all edges of your graph identical?* – Every spanning tree on n points contains exactly $n-1$ edges. Thus, if your graph is unweighted, *any* spanning tree will be a minimum spanning tree. Either breadth-first or depth-first search can be used to find a rooted spanning tree in linear time. DFS trees tend to be long and thin, while BFS trees better reflect the distance structure of the graph, as discussed in Section 5 (page 145).

- *Should I use Prim's or Kruskal's algorithm?* – As implemented in Section 6.1 (page 192), Prim's algorithm runs in $O(n^2)$, while Kruskal's algorithm takes $O(m \log m)$ time. Thus Prim's algorithm is faster on dense graphs, while Kruskal's is faster on sparse graphs.

  That said, Prim's algorithm can be implemented in $O(m + n \lg n)$ time using more advanced data structures, and a Prim's implementation using pairing heaps would be the fastest practical choice for both sparse and dense graphs.

- *What if my input is points in the plane, instead of a graph?* – Geometric instances, comprising n points in d-dimensions, can be solved by constructing the complete distance graph in $O(n^2)$ and then finding the MST of this complete graph. However, for points in two dimensions, it is more efficient to solve the geometric version of the problem directly. To find the minimum spanning tree of n points, first construct the Delaunay triangulation of these points (see Sections 17.3 and 17.4). In two dimensions, this gives a graph with $O(n)$ edges that contains all the edges of the minimum spanning tree of the point set. Running Kruskal's algorithm on this sparse graph finishes the job in $O(n \lg n)$ time.

- *How do I find a spanning tree that avoids vertices of high degree?* – Another common goal of spanning tree problems is to minimize the maximum degree,

typically to minimize the fan out in an interconnection network. Unfortunately, finding a spanning tree of maximum degree 2 is NP-complete, since this is identical to the Hamiltonian path problem. However, efficient algorithms are known that construct spanning trees whose maximum degree is at most one more than required. This is likely to suffice in practice. See the references below.

Implementations: All the graph data structure implementations of Section 12.4 (page 381) *should* include implementations of Prim's and/or Kruskal's algorithms. This includes the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For some reason it does not seem to include the Java graph libraries oriented around social networks, but Prim and Kruskal are present in the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*).

Timing experiments on MST algorithms produce contradicting results, suggesting the stakes are really too low to matter. Pascal implementations of Prim's, Kruskal's, and the Cheriton-Tarjan algorithm are provided in [MS91], along with extensive empirical analysis proving that Prim's algorithm with the appropriate priority queue is fastest on most graphs. The programs in [MS91] are available by anonymous FTP from *cs.unm.edu* in directory */pub/moret_shapiro*. Kruskal's algorithm proved the fastest of four different MST algorithms in the Stanford GraphBase (see Section 19.1.8 (page 660)).

Combinatorica [PS03] provides Mathematica implementations of Kruskal's MST algorithm and quickly counting the number of spanning trees of a graph. See Section 19.1.9 (page 661).

My (biased) preference for C language implementations of all basic graph algorithms, including minimum spanning trees, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: The MST problem dates back to Boruvka's algorithm in 1926. Prim's [Pri57] and Kruskal's [Kru56] algorithms did not appear until the mid-1950's. Prim's algorithm was then rediscovered by Dijkstra [Dij59]. See [GH85] for more on the interesting history of MST algorithms. Wu and Chao [WC04b] have written a monograph on MSTs and related problems.

The fastest implementations of Prim's and Kruskal's algorithms use Fibonacci heaps [FT87]. However, pairing heaps have been proposed to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

A simple combination of Boruvka's algorithm with Prim's algorithm yields an $O(m \lg \lg n)$ algorithm. Run Borukva's algorithm for $\lg \lg n$ iterations, yielding a forest of at most $n/\lg n$ trees. Now create a graph G' with one vertex for each tree in this forest, with the weight of the edge between trees T_i and T_j set to the lightest edge (x, y), where $x \in T_i$ and $y \in T_j$. The MST of G' coupled with the edges selected by Boruvka's algorithm yields the MST of G. Prim's algorithm (implemented with Fibonacci heaps) will take $O(n + m)$ time on this $n/\lg n$ vertex, m edge graph.

The best theoretical bounds on finding MSTs tell a complicated story. Karger, Klein, and Tarjan [KKT95] give a linear-time randomized algorithm for MSTs, based again on Borukva's algorithm. Chazelle [Cha00] gave a deterministic $O(n\alpha(m,n))$ algorithm, where $\alpha(m,n)$ is the inverse Ackerman function. Pettie and Ramachandran [PR02] give an provably optimal algorithm whose exact running time is (paradoxically) unknown, but lies between $\Omega(n+m)$ and $O(n\alpha(m,n))$.

A *spanner* $S(G)$ of a given graph G is a subgraph that offers an effective compromise between two competing network objectives. To be precise, they have total weight close to the MST of G, while guaranteeing that the shortest path between vertices x and y in $S(G)$ approaches the shortest path in the full graph G. The monograph of Narasimhan and Smid [NS07] provides a complete, up-to-date survey on spanner networks.

The $O(n \log n)$ algorithm for Euclidean MSTs is due to Shamos, and discussed in computational geometry texts such as [dBvKOS00, PS85].

Fürer and Raghavachari [FR94] give an algorithm that constructs a spanning tree whose maximum degree is almost minimized—indeed is at most one more than the lowest-degree spanning tree. The situation is analogous to Vizing's theorem for edge coloring, which also gives an approximation algorithm to within additive factor one. A recent generalization [SL07] gives a polynomial-time algorithm for finding a spanning tree of maximum degree $\leq k+1$ whose cost is no more than that of the optimal minimum spanning tree of maximum degree $\leq k$.

Minimum spanning tree algorithms have an interpretation in terms of *matroids*, which are systems of subsets closed under inclusion. The maximum weighted independent set in matroids can be found using a greedy algorithm. The connection between greedy algorithms and matroids was established by Edmonds [Edm71]. Expositions on the theory of matroids include [Law76, PS98].

Dynamic graph algorithms seek to maintain an graph invariant (such as the MST) efficiently under edge insertion or deletion operations. Holm et al. [HdlT01] gives an efficient, deterministic algorithm to maintain MSTs (and several other invariants) in amortized polylogarithmic time per update.

Algorithms for generating spanning trees in order from minimum to maximum weight are presented in [Gab77]. The complete set of spanning trees of an unweighted graph can be generated in constant amortized time. See Ruskey [Rus03] for an overview of algorithms for generating, ranking, and unranking spanning trees.

Related Problems: Steiner tree (see page 555), traveling salesman (see page 533).