Graph Theory. Chapter 2.

Spanning trees.

The government has decided to scrap the current railway system and start again from scratch. (This may not be true.) We are to decide what railway lines will be needed. We have a list of towns to be included in the network, and the distances between any town on the list and the neighbouring towns on the list. We want to build as little track as possible.

Note that the resulting system will give a weighted spanning tree for the graph whose vertices are the towns on the list. It will be a tree as given any circuit we could remove one of the edges, and it will be a spanning tree as we need to be able to reach every town.

A spanning tree can be constructed as follows. Pick a town on the list, say London, and apply Dijkstra's algorithm. Using the path procedure (as in the last chapter) we get a minimal path from every town to London, and it is easy to prove that the edges on these minimal paths will produce a spanning tree. Unfortunately, this railway system is only optimal if we regard the object of the exercise being to supply the shortest route from any town to London. However, this London based solution to the railway net-work problem cannot be expected to give the shortest possible spanning tree.

We now want to solve the real railway problem. So define the *weight* of a tree to be the sum of the weights of its edges. Then we want to find a spanning tree of minimal weight.

Before getting down to the algorithm, a little tree theory.

Lemma. *In a tree there is a unique path between any two vertices.*

Proof. Suppose not.

Let T be a tree which contains two different paths joining vertices v and w. Let $v_0, e_0, v_1, e_1, \ldots, e_{m-1}, v_m$ be one of these paths, and let $w_0, f_0, w_1, f_1, \ldots, f_{n-1}, w_n$ be the other, so $v = v_0 = w_0$, and $w = v_m = w_n$ is the other.

To get our contradiction we need to find a circuit in T.

We may assume that the paths have been chosen with m and n as small as possible.

Now consider the walk $v = v_0, e_0, \ldots, e_{m-1}, v_m = w_n, f_{n-1}, \ldots, f_0, w_0 = v$. If this is a circuit we are done; so suppose not. Then $v_i = w_j$ for some i, j with $0 < i+j < m+n$. But then we could have taken i and j in place of m and n, and this contradicts the minimality of m and n, and this completes the proof.

Lemma. *Let T be a tree, and e be an edge in T. Then the graph S obtained by deleting e, but retaining all vertices and all the other edges, has two components, both of which are trees.*

Proof. Let v and w be the ends of e. Since T is a tree, $v \neq w$. Construct two subgraphs T_1 and T_2 of T as follows. The vertices of T_1 are the vertices x with the property that the path from x to v does not pass through w, and the vertices of T_2 are all the other vertices. The edges of T_1 are the edges of T with both ends in T_1, and similarly for T_2. Now T_1 and T_2 have no circuits, as T has none.

The next thing is to prove that no edge in S has one end in T_1 and the other in T_2. Let x be a vertex in T_1. Then the path p in T from x to v does not pass through w; and so all the vertices on this path lie in T_1. Now suppose that an edge f joins x to y in T_2.

This gives rise to a walk from y to v that does not pass through w, namely the path y, f, x followed by the path p. But this is clearly a path, as it does not pass twice through y, and cannot pass twice through any other vertex. So y lies in T_1. Thus no edge in S has one end in T_1 and one in T_2, as required.

Finally we show that T_1 and T_2 are connected. If x and y are vertices in T_1 then there are paths from x to v and from v to y that do not pass through w, and hence do not include the edge e. Combining these gives a walk in S from x to y, as required. Similarly if x and y are in T_2 then the path from x to w does not pass through v (for otherwise this would give a path from x to v that does not pass through w); and similarly for y; so again there is a path in S from x to y. Now in both cases (when x and y are both in T_1 and when they are both in T_2) there is a walk from x to y in S, and in the first case this walk is in T_1, since we have seen that a walk in S that starts in T_1 can never leave T_1; and similarly for the second case.

Thus S consists of two components, T_1 and T_2; and these are both trees.

Lemma. *Let T be a tree with exactly n vertices. Then T has exactly $n-1$ edges.*

Proof. This is an easy induction on n. If $n = 1$ the result is obvious. So let $n > 1$, and suppose that the result holds for trees with fewer than n vertices. Remove an edge from T, producing a graph S with two components T_1 and T_2, each having fewer than n vertices. Let T_1 have m vertices, so T_2 has $n - m$ vertices. By induction T_1 has exactly $m - 1$ edges and T_2 has exactly $n - m - 1$ edges. So by the induction hypothesis S has $(m - 1) + (n - m - 1) = n - 2$ edges, and so T has $n - 1$ edges.

Lemma. *Let G be a connected graph with n vertices and $n-1$ edges. Then G is a tree.*

Proof. Suppose that G is any connected graph. If G is not a tree then it has a circuit, and removing from G any edge in the circuit the graph remains connected. So we can remove edges from G keeping G connected, until G has no circuits, and hence has become a tree. Now suppose that G has n vertices. By the previous lemma, if we follow the above process, we will have a tree when exactly $n - 1$ edges are left; so if G started with $n - 1$ edges it must have been a tree to start with.

Now we get back to algorithms.

An algorithm is said to be 'greedy' if it always choses what seems (by some simple criterion) to be the best move, without looking ahead. So Dijkstra is a greedy algorithm, as can be seen by thinking about the 'next' procedure called by Dijkstra, which does not look ahead. The minimal weight spanning tree algorithm we are going to study, due to Prim (1957) is also a greedy algorithm. In its simplest form it goes as follows.

```
procedure Prim(G)
/* Find a spanning tree of minimal weight in the connected weighted graph G. */
begin
  v := any vertex of G;
  T := the subtree of G consisting of the vertex v and no edges;
  n := the number of vertices of G;
  for i := 1 to n − 1 do
    e := an edge of G of minimal weight with one end a in T and one end b out of T;
    adjoin b and e to T
```

end for;
return T
end;

The first thing to do is to check that the procedure is correct.

It is clear that T is a tree throughout the procedure. For it is always connected, with one more vertex than edge.

When the procedure is complete we have a tree T with n vertices; so T is then a spanning tree of G.

The more interesting question is whether the spanning tree returned is of minimal weight. Suppose then that the smallest weight for a spanning tree of G is w, where w is less than the weight of a spanning tree T constructed according to Prim's algorithm. Suppose that the sequence of vertices and edges used in the construction of T was $v_0, e_1, v_1, \ldots e_{n-1}, v_{m-1}$. So v_0 is the first vertex, chosen at random, and then e_i, v_i are added to T for $i = 1, 2, \ldots, n-1$, according to the algorithm. Now let S be a spanning tree of weight w. Then S will contain the vertex v_0, but will not contain all the edges e_i, since $S \neq T$. Let j be the least integer such that e_j is not in S. The value of j will depend on the choice of S; assume that S was chosen to maximise j (subject to S being a spanning tree of weight w). Let G be the graph obtained by adding the edge e_j to S. Now G is a connected graph with n vertices and n edges, and as it becomes a tree when e is removed, it must have a circuit that contains e. If any edge is removed from this circuit, the graph obtained is clearly a spanning tree of G. If this circuit contains an edge f of greater weight than e then removing f from G leaves a tree of smaller weight than S, which is impossible. Now the circuit must have another edge g say with one end in $\{v_0, v_1, \ldots, v_{j-1}$ and the other in $\{v_j, v_{j+1}, \ldots, v_{m-1}$. So in Prim's algorithm, when e_j was added, g was a candidate edge. Why did we adjoin e_j rather than g? There are two possible reasons:

(a) Edge g is longer than edge e_j. But this has been ruled out.

(b) The edges have the same length, and we happened to pick e_j. In this case, if we remove edge g from G we have a spanning tree of length w that contains e_i for all $i \leq j$, which contradicts our assumption about maximising j. This contradiction completes the proof.