Spanning Trees

Michael P. Fourman

February 2, 2010

A *tree* is a connected, acyclic graph (V, E). In a tree, $|E| = |V| - 1$ (intuitively, each edge connects one 'non-root' vertex to its 'parent'). The trees we have considered earlier have been *rooted* trees; one vertex is distinguished as the root, and the tree is drawn dangling from this vertex. Choosing a root vertex provides a direction for each edge—conventionally, pointing away from the root. Undirected trees have the properties (a) that, if a new edge is added, the graph is no longer acyclic, and (b) that there is a path between every pair of vertices but, if any edge is deleted, this is no longer true.

A collection of edges that provides a path from the start vertex to all other vertices in the graph, is a *spanning tree,* for the graph. Given a connected, weighted, undirected graph, we want to find a minimal spanning tree—we seek to minimise the sum of the weights of the edges in the tree. We will present two algorithms based on a common idea; both algorithms construct a spanning tree by incrementally adding to a set of edges, and maintain the invariant that this set of edges is a subset of some minimal spanning tree.

We say two vertices of the graph are *linked* by this set if there is a path between them consisting only of edges in the set. Each new edge joins two previously unlinked vertices; this ensures that the collection of edges is acyclic. The algorithms are known by the names of two of their discoverers, Kruskal and Prim.

We will argue argue that, at each step in the construction,,

- *If the original set of edges is contained in some minimal spanning tree, then so is the augmented set.*
- *Two vertices are linked if there is some path between them via previously considered edges.*

In both cases, we start with an empty set of edges, which is certainly contained in a minimal spanning tree. When we have considered all edges, we can conclude that we have a set of edges, linking all nodes, and contained in some minimal spanning tree. It must *be* a minimal spanning tree.

Prim's algorithm

Prim's algorithm grows a tree from an arbitrary root vertex. To extend the tree we select an edge of minimal weight from among those that join a new node to the tree. We maintain a priority queue of edges. When we link a node to the tree, we add to the queue all edges leading from that node. Prim's algorithm is a variation of of our generic graph search; a node is visited when it is linked to the tree.

Suppose the tree A is contained in some minimal spanning tree T. We form a new tree A' by adding an edge (u, v) of minimal weight linking some node u in A to a node v not in A. The edge If (u, v) belongs to T, our new tree, A' is contained in T. Otherwise, T must contain a path linking u, which is in A, and v, which is not. This path must contain some edge (u', v') for which u' is in A, and v' is not. Remove (u', v') from T, and add (u, v), to form a new set of edges, T', which is a spanning tree containing A'. As the weight of (u, v) is minimal, T' is a minimal spanning tree.

An implementation of Prim's algorithm is given by the functor `PRIM`. The auxiliary function `grow` is called $O(|E|)$ times, each with a call to `member`; the priority queue provided by `PQ` is used to process $O(|E|)$ edges. The `else` branch of the recursion is called $O(|V|)$ times, once for each vertex, each with two calls to `insert`. The complexity of the algorithm will depend on the implementation the sets and queue.

If the graph is not connected, Prim's algorithm returns a spanning tree for the component of the initial vertex selected.

Kruskal's algorithm

Kruskal's algorithm grows a forest of trees. Edges of the graph are considered in turn; an edge is added to the forest if it joins two previously unconnected vertices. Edges are considered in order of decreasing weight.

The argument for Kruskal's algorithm is just like that for Prim's. When we extend a forest F to form a forest F', we ensure that any minimal spanning tree T extending F can be modified by minor surgery, to form a minimal spanning tree T' containing F'.

Implementing Kruskal's algorithm efficiently requires a special data-structure, called a *partition,* to keep track of which vertices are linked. This data-structure represents the partitioning of a set into disjoint subsets. It supports two operations: `union`, which forms a new partition in which two subsets have been merged, and `find`, which produces a representative of the subset containing a given element. Two elements, x, and y, are in the same subset of the partition, p, if and only if $\texttt{find}(p)(x) = \texttt{find}(p)(y)$. A signature is

given in the code for Kruskal's algorithm. Implementations of this signature will be discussed later in the course.

The implementation of Kruskal's algorithm, given by the functor `KRUSKAL`, also makes $O(|E|)$ calls to its auxiliary function, `grow`; each call includes two calls to `find`. Again, a priority queue is used to process $O(|E|)$ edges. Since we are building a spanning tree, the `else` branch is executed $|V| - 1$ times, once for each edge in the tree; each time includes one call to `insert` and one to `union`.

For graphs that are not connected Kruskal's algorithm returns a forest of spanning trees— one for each component of the graph.

```
functor PRIM(
structure G :  sig
   type Vertex type Edge type Graph
    val adj : Graph -> Vertex -> Edge list
    val ends: Edge -> Vertex * Vertex
end
   structure VS : SetSig
   structure  T : SetSig
   structure PQ : QueueSig
   sharing type G.Vertex = VS.Item
       and type  PQ.Item = T.Item = G.Edge
) = struct
   type Graph  = G.Graph
   type Vertex = G.Vertex
   type Tree   = T.Set

   fun span (g:Graph) (s:Vertex) : Tree =
   let fun grow edges tree included =
        if PQ.isEmpty edges then tree else
            let val (e, es) = PQ.deq edges
                val (u, v) = G.ends e
             in
                if VS.member included v then
                   grow es tree included
                else grow
                     (PQ.menq (G.adj g v, es))
                     (T.insert (e, tree))
                     (VS.insert (v, included))
            end
    in
       grow (PQ.menq(G.adj g s, PQ.empty))
            T.empty
            (VS.insert(s,VS.empty))
    end
end;
```

```
functor KRUSKAL(
structure G: sig
     type Edge
     type Vertex
     type Graph
     val ends : Edge -> Vertex * Vertex (* graph *)
     val edges: Graph -> Edge list
    end
structure PQ: QueueSig  (* a priority queue of edges *)
structure  F: SetSig    (* a set of edges *)
structure P: sig
     type Part
     eqtype Item
     val empty: Part
     val union: Part -> (Item*Item) -> Part
     val find : Part -> Item -> Item
    end
    sharing type F.Item = PQ.Item = G.Edge
        and type P.Item = G.Vertex
) = struct
   type Graph  = G.Graph
    and Forest = F.Set

   fun span (g: Graph) : Forest =
   let fun grow edgeQ forest p =
           if PQ.isEmpty edgeQ then forest
           else
           let val (e,q) = PQ.deq edgeQ
               val (u,v) = G.ends e
           in if P.find p u = P.find p v
                 then grow q forest p
              else grow q
                       (F.insert(e, forest))
                       (P.union p (u,v))
           end
    in
       grow (PQ.menq (G.edges g, PQ.empty)) F.empty P.empty
    end
end;
```