Round Robin Algorithm and Leftist Heaps

Jonathan Turner

February 22, 2013

So far, we have studied two algorithms for the minimum spanning tree problem, Prim's algorithm and Kruskal's algorithm. When implemented with Fibonacci heaps, Prim's algorithm runs in $O(m + n \log n)$ time, which is optimal for $m \geq n \log n$. For sparser graphs, there is an opportunity to improve on this. In this section, we study an algorithm that can be implemented to run in $O(m \log \log n)$ time, making it faster than Prim's algorithm for very sparse graphs ($m < (n \log n)/\log \log n$).

The *round robin algorithm* for the minimum spanning tree problem can be viewed as another special case of the general greedy method. It builds a collection of blue trees, by repeatedly merging trees using the following rule, until there is just one tree left.

> *Coloring Rule.* Select a blue tree and minimum cost edge incident to it. Color the edge blue

An edge is *incident* to a tree if one of its endpoints is in the tree and the other is not.

Expressed in algorithmic notation, this becomes

```
procedure minspantree(graph G = (V, E), modifies set blue)
    vertex u, v; list queue; Partition(V);
    blue := {}; queue := [];
    for u ∈ [1..n] ⇒ queue := queue & [u]; rof
    do |queue| > 1 ⇒
        Let {u, v} be a min cost edge incident to the tree that
        contains queue(1)
        blue := blue ∪ {{u, v}}; queue := queue − {find(u), find(v)};
        link(find(u),find(v)); queue:= queue & [find(u)];
    od; // edges not added to blue are implicitly red
end;
```

The partition data structure is used to determine if two vertices are in the same blue, as in Kruskal's algorithm. Whenever we add an edge joining two trees, we link the corresponding sets.

To complete the algorithm, we need an efficient way to determine the least cost edge incident to a tree. The natural thing to do is to define a heap for every tree, where each heap contains the edges incident to vertices in the heap. Since each edge connects two trees, each edge will appear in two heaps. However, this immediately raises two issues.

- When we combine two trees together, we need to replace their heaps, with a new heap containing the edges incident to the new tree. This suggests using a meldable heap, like a Fibonacci heap.

- But this is not quite right, since melding the original heaps may result in a new heap that contains many edges that now join two vertices in the same tree.

One way to address the second issue is to go through the new heap and delete from it, all edges that now join two vertices in the same tree. Unfortunately, removing these edges is an expensive operation, so we need to find a way to avoid it whenever we can.

The concept of *lazy deletion* can be helpful in this situation. This is a general technique that can often be used to avoid much of the work associated with deleting an item from a data structure. The idea is to set a bit, marking the item as deleted, but delaying the actual deletion to some future time when it may be possible to handle it more efficiently. In many cases we can put off the actual deletion until the algorithm terminates, eliminating the need to process the deletion at all. The round-robin algorithm uses a particularly clever form of lazy deletion, but before we can describe it, we need to introduce a new data structure, the *leftist heap*.

The leftist heap data structure defines a collection of heaps over a set of items. Like the Fibonacci heap data structure, it supports a meld operation. In addtion, it supports an efficient form of lazy deletion.

A leftist heaps is defined with respect to a *full binary tree*, which is simply a binary tree in which every non-leaf node has two children. We store a data item at every internal node and the external nodes (aka leaf nodes) are represented implicitly. We define the *rank* of a node x as the length of the shortest path from x to an external node. We say that a tree is *leftist* if $rank(left(x)) \geq rank(right(x))$ for every internal node x. In a leftist heap, each node has an associated key, and the key values are arranged in the usual heap order. These definitions are illustrated in Figure 1.

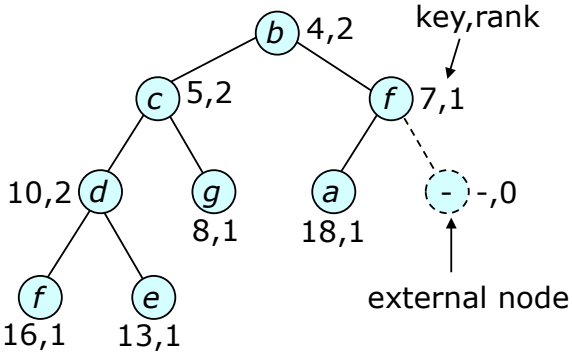

Figure 1: Example of a leftist heap

In a leftist heap, the path from the root to the rightmost leaf is referred to as the *right path*. The leftist property ensures that the right path is a shortest path from the root to a leaf, and has length at most $\lg n$. This property enables an efficient implementation of the meld operation.