# Implementing Dynamic Trees

Jonathan Turner

April 14, 2013

In this section, we'll complete our discussion of the dynamic trees data structure. In particular, we'll look at how path sets can be implemented efficiently using binary search trees, and then we'll complete the analysis of the performance of the complete data structure. Before we get started, let's review the big picture, illustrated in Figure 1.

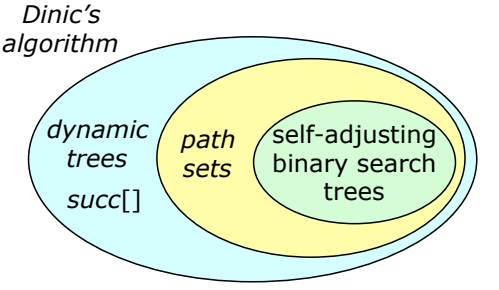


Figure 1: Levels of abstraction for dynamic trees

The motivating application for the data structure is Dinic's algorithm, which uses dynamic trees to store information about portions of the residual graph that may be used to construct new augmenting paths. The data structure divides trees into paths, that are linked together using the successor array *succ*[]. The *expose* operation modifies the path sets, so that the nodes on a specific node-to-root path can be processed efficiently, without affecting other nodes. In this section, we'll examine the inner-most layer of the diagram, where self-adjusting binary search trees are used to implement the paths.

To represent a path as a binary search tree, we simply equate the head-to-tail order of the nodes in the path with the left-to-right order of the nodes in the binary search tree. This is illustrated in Figure 2 which shows a path and two different search trees that can be used to represent it. If $u$ comes

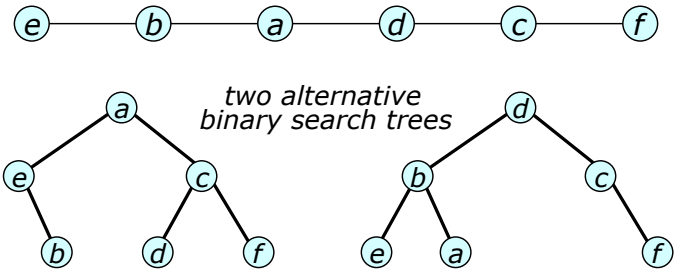


Figure 2: Representing paths as search trees

before $v$ in the path, then $u$ must appear to the left of $v$ in the binary search tree. Any search tree that satisfies this condition for all pairs of nodes in the path is a legitimate representation of the path. So for example, if we do a rotation operation on a search tree, the new search tree that results is still a valid representation of the path, since the rotation operation preserves the left-to-right ordering of the nodes in the search tree. Notice that no keys are required here. While we can think of the search tree nodes as having keys equal to their positions in the path, there is no need to represent these keys explicitly.

Of course, while we may not need the usual binary search tree keys, the path sets data strcture does define costs for every node, and we need a way to efficiently determine the minimum cost of any node in a path. To make this easier, we can associate a *mincost* value with each subtree of a binary search tree. This is illustrated on the left side of Figure 3. For each node $x$,

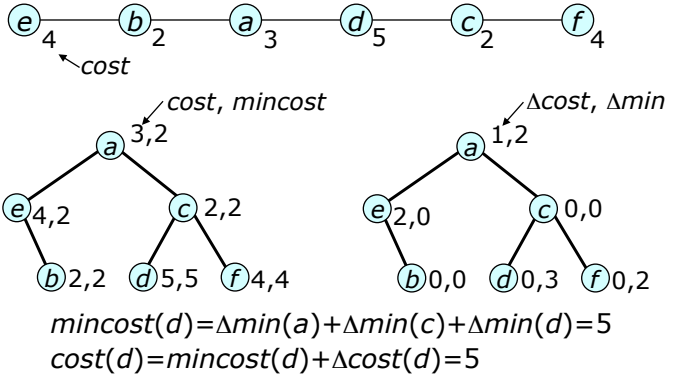


Figure 3: Representsing node costs in search trees

$mincost(x)$ is the minum cost among all the nodes in the subtree with root $x$,

so $mincost(e) = 2$, because the subtree with root $e$ includes the node $b$ that has cost 2. Using the *mincost* values, it's trivial to determine the minimum cost for a path. Moreover, it's straightforward to find the last node on the path that has the minimum cost. If the right subtree of the root has the same *mincost* value as the root, we'll find the last min cost node in the right subtree. If not, but the root has minimum cost, then the root is the last min cost node in the path. Otherwise, the last min cost node is in the left subtree. These observations lead directly to an efficient search procedure.

So this is great, but it's not a complete solution to the problem of implementing path sets efficiently. To complete the implementation, we also need a way to efficiently change the costs of all nodes in a path. If we store the *cost* and *mincost* values explicitly, as fields in the search tree nodes, we'll need to update those fields every time we do an *addpathcost* operation. To make the *addpathcost* efficient, we'll represent the cost information using a *differential representation*. First, we define

$$\Delta min(x) = \begin{cases} mincost(x) & \text{if } x \text{ is a tree root} \\ mincost(x) - \Delta min(p(x)) & \text{if } x \text{ is not a tree root} \end{cases}$$

This definition is illustrated on the right side of Figure 3. Notice that $mincost(x)$ is the sum of the $\Delta min$ values on the path from $x$ to the root (in the example $mincost(c) = \Delta min(a) + \Delta min(c) + \Delta min(d) = 2 + 0 + 3 = 5$). Consequently, as we follow a path down from the root, we can easily determine the *mincost* value of the current node; so, we can still use it to quickly find the last min cost node in the path.

To determine the actual cost of a tree node, we maintain a second field with each node.

$$\Delta cost(x) = cost(x) - mincost(x)$$

So, we can determine the actual cost of $x$ by adding $\Delta cost(x)$ to $mincost(x)$. The advantage of this representation is that we can now do an *addpathcost* operation on a path by simply adding the desired increment to the $\Delta min$ field of the root. This effectively changes the *mincost* values for all nodes and hence their costs, as well.

The differential cost representation does complicate some of the routine operations on search trees. In particular, whenever we perform the rotation operation, $rotate(x)$, the *mincost* values of $x$ and its original parent change. This requires an adjustment to the $\Delta min$ and $\Delta cost$ values of these nodes and to the $\Delta min$ values of their children.

It's important to remember that the node costs here are not binary search tree keys. So, in particular, the costs need not satisfy the usual left-to-right

ordering property for search tree keys. In this application, we are using the search tree ordering only to determine the order of nodes in the path. This can be done without any explicit keys.