Advanced Data Structures Jan-Apr 2012

Lecture 14 — March 13, 2012

Lecturer: Venkatesh Raman *Scribe: Sudeshna Kolay*

1 Overview

In this lecture we will look at a data structure for dynamic graphs, that will efficiently support desired operations like deleting or adding edges.

2 Main Section

When we say dynamic graphs we want to maintain a forest of vertex disjoint rooted trees at all times. Since each tree is rooted, we can define the depth of each node in the current forest. In each tree every node except the root has a unique parent and all edges are directed towards the root. Let the number of vertices in the graph be n.

On such graphs we want to efficiently support operations like the following:

1. *findRoot(x)*, which finds the root of the tree to which the argument x of the function belongs
2. *link*(x, y), which, when x is the root of a tree and x and y belong to distinct trees, adds the directed edge (x, y)
3. *cut(x)*, which deletes the edge (x,*parent(x)*) from the tree to which x currently belongs
4. *makeTree(x)* makes a tree out of the singleton node x.

Note that given a singleton x, *makeTree(x)* takes constant time. We will maintain an array of nodes of the graph, with each node pointing to its copy in the forest. This enables us to access a node x in constant time.

2.1 First attempt

Suppose for each node x in the array we also store the *parent(x)*. Then both *link*(x, y) and *cut(x)* operations take constant time:

1. *link*(x, y) assumes that x is a root. So it will simply reset the *parent(x)* to y.
2. *cut(x)* will reset *parent(x)* to x itself.

However, *findRoot(x)* could take as much as $O(n)$ time.

2.1.1 Link-cut Trees

Link-cut trees were first introduced by Sleator and Tarjan [1] [2]. This data structure supports the above operations in $O(\log n)$ amortised time.

Like tango trees (Lecture 4), link-cut trees also decompose each tree of the forest into a collection of vertex disjoint paths, called preferred paths. Each of these preferred paths are stored as auxiliary trees.

Definition : The i^{th} child of node x is a *preferred child* of x if the latest access in the subtree rooted at x is in the subtree rooted at the i^{th} child. If there was no access in the subtree of x or the access was x itself then we set the preferred child of x to NIL. The edge between x and its preferred child is called a *preferred edge.*

Definition : A *preferred path* from a node x is the path traced out by the preferred children at each node.

Thus, we can decompose each tree of the forest by constructing preferred paths from the nodes in increasing height order. The auxiliary trees in which we represent the preferred paths are splay trees, where the key for each node is its depth in the rooted tree it belongs to.

Definition : A *pathParent* of a preferred path is the parent of the node of lowest depth in the preferred path.

The pathParent of a preferred path is stored in the root of the corresponding auxilliary tree. If a root is contained in the preferred path the pathParent is NIL.

Now we look at a function *access()* which is going to be the building block for all the desired operations. Notice that once the node x is accessed then the path from x to root becomes the preferred path. If x was already contained in the preferred path containing the root then the part of the path below x has to be cut out. Otherwise the preferred path from the root has to be changed to the path from root to x.

access(x):

```
    Splay at $x$,ie $x$ becomes root. All nodes with higher depth is in the right subtree, with
      lower depth in the left subtree
    Remove preferred child of $x$:
    pathParent(right($x$)) = $x$
    parent(right($x$)) = right($x$)
    right($x$) = NIL
    while ($x$ is not a root) :
      Splay at pathParent ($x$).
      pathParent(right(pathParent($x$))) = pathParent($x$)
      parent(right(pathParent($x$))) = right(pathParent($x$))
      right(pathParent($x$)) = x.
      parent($x$) = pathParent($x$)
      pathParent($x$) = NIL
```

Finally we can splay at x so that it is the root at the end of access.

In terms of access the rest of the operations can be defined:

1. *findRoot(x)*:

   access(x); This brings x to the same auxilliary splay tree as its root.
   find leftmost leaf of the auxilliary splay tree. This is the root.
   Splay at the root.

2. *cut(x)*: We need to disconnect x from its parent. The preferred path from x could be decomposed as singleton x and the preferred path below x, with x as its pathParent.

   access(x); in the end x is the root with an empty right subtree
   The left subtree of x has to be disconnected:
   pathParent(left(x)) = pathParent(x) parent(left(x)) = left(x)
   pathParent(x) = NIL
   parent(x) = x

3. *link*(x, y):

   access(x); As x was a root access makes it a singleton node
   access(y); In the end y is the root with an empty right subtree
   right(y) = x
   parent(x) = y
   pathParent(x) = NIL

2.1.2 Analysis for $O(\log^2 n)$ amortised time

In all the operations we have to find the time taken for *access()*. Other than that all other operations take $O(\log n)$ time. Total time by *access()* is the time taken by each splaying times the number of splays required in the while loop. The time taken for a splay is $O(\log n)$ amortised. The number of splays required is the number of times a node changes to a preferred child, or the edge to its parent changed into a preferred edge.

Definition : A node x is *heavy* if $size(x) > \frac{1}{2} size(parent(x))$ in the forest representation.

Definition : An edge (x,y) is heavy if x is a heavy child of y. Otherwise the edge is light.

The number of light edges in a path from the root has to be at most $\log n$, as at each light edge we reduce the subtree by more than half. So the number of light edges that become preferred edges is bounded by $\log n$.

The number of heavy edges that become preferred edges is at most the sum of the number of heavy edges that become unpreferred and the total number of edges in the tree of the forest. This is because in the end all the edges of the tree in the forest could become heavy and preferred. Also if a heavy edge becomes preferred twice then it has to be unpreferred in between. So the total number of heavy edges that became preferred is bounded by $m \log n + (n-1)$, where m is the total number of accesses. Hence, the amortised number of heavy edges changing to preferred edges per *access* is $O(\log n)$.

From the above the amortised time of $O(\log^2 n)$ per operation follows.

2.1.3 Analysis for $O(\log n)$ amortised time

Now we pay more attention to the cost of splaying when a node changes to a preferred child. Cost of splaying at $x < 3(\log \mathit{size}(y) - \log \mathit{size}(x)) + 1$ Here y is the root of the auxilliary tree where x belongs. Let z be the pathParent(y).

Then $\mathit{size}(x) < \mathit{size}(y) < \mathit{size}(z)$. (Lecture 3)

Now we add the equations for the cost of splaying at each node that changes into a preferred child till we reach r, the root in the forest representation. This is a telescoping sum. Let p be the number of nodes changed to preferred child.

Total cost of splaying for an access $< 3(\log \mathit{size}(r) - \log \mathit{size}(x)) + O(p)$

$\mathit{size}(r) \leq \log n$.

Therefore, the total cost of splaying per access works out to be $O(\log n)$.

References

[1] D. D. Sleator, R. E. Tarjan, *A Data Structure for Dynamic Trees*, Journal. Comput. Syst. Sci., 28(3):362-391, 1983.

[2] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial Applied Mathematics, 1984.