Advanced Data Structures	Jan-Apr 2012
Lecture 28 May 4, 2012	
Lecturer: Anil Shukla	*Scribe: Anil Shukla*

1 Overview

The problem of maintaining a forest where edge insertions and deletions are allowed is called the dynamic trees problem. Efficient solution to this problem have numerous applications, particularly in dynamic graphs. Several data structures which supports dynamic trees problem in logarithmic time have been proposed. In these lecture we will see two such data structures, the first is **ST-tress** proposed by Sleator and Tarjan in [1] [2] and the second one is **top trees**, proposed by Alstrup et al. in [4]. This lecture will be based on Renato F. Werneck's PhD dissertation [3].

2 Introduction

2.1 Dynamic Trees Problem

We are given n vertex forest of rooted tress with costs on edges. We are allowed to modify the structure of the forest by two basic operations: (1) $link(v, w, c)$, which adds an edge between a root v and a vertex w in a different tree (component) with cost c; (2) $cut(v)$, which removes the edge between v and its parent. We are interested in a data structure to maintain a forest supporting in $O(\log n)$ time queries and updates related to vertices and edges indiviually, and to entire trees or paths. We call this the dynamic trees problem.

3 Sleator and Tarjan's ST Trees [1] [2]

The first data structures to support dynamic tree operations in $O(\log n)$ time were Sleator and Tarjan's ST trees [1] [2]. Primarily ST-trees are used to represent rooted trees with all edges directed towards the root. ST-trees proposed in [2] associates a cost with each vertex in the forest and handle the costs by the following operations:

- $findcost(v)$: returns the cost of vertex v.
- $findmin(v)$: returns the minimum-cost vertex on the path from v to the root of its tree.
- $addcost(v, x)$: adds x to each vertex on the path from v to the root of its tree.

Apart from the usual update operation $link(v, w, c)$ and $cut(v)$, it also supports the following operations:

- $findroot(v)$: returns the root of the tree containing v.

- *parent*(v): returns the parent of v.

3.1 Representation

ST-trees are based on path-decomposition technique. Each rooted trees in the forest is represented as follows. The tree is first partitioned into vertex-disjoint paths. Edges within a path are called solid edge (and so is the path). The remaining edges (that are not in any partition) are called dashed edges. Dashed edges links solid paths. Each solid path is represented as a solid subtree, which is a binary search tree in which the bottom most vertex of the original solid path is represented as the leftmost vertex. Finally, the solid subtrees are "glued" together (defined below), creating a shadow (or virtual) tree.

Shadow (virtual) tree: Consider a solid path P. Let v be the topmost vertex of P and let $p(v)$ be its parent which belongs to some other solid path Q. Let r_p be the root of the solid subtree representing P then in shadow tree we have a dotted edge between r_p and $p(v)$. We call r_p as the middle child of $p(v)$. See figure 1. There are no pointers from a node to its middle children.

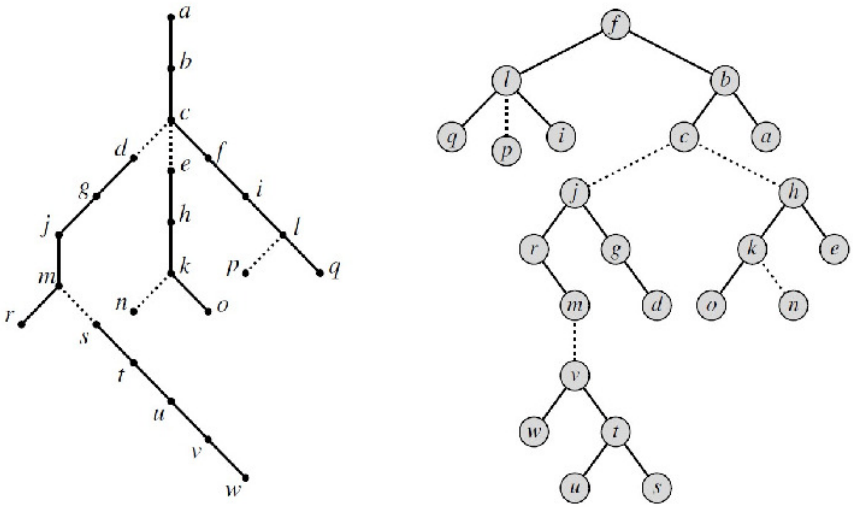

Figure 1: Example of an ST-tree (adapted from [3]). On the left, the original rooted tree with root `a', partitioned into solid paths; on the right, a shadow tree representing the tree on its left. In shadow tree, middle children are connected to their parents by dotted lines. Costs are not shown in the figure.

In [2], Sleator and Tarjan proposed a simple implementation of ST-trees, where each solid paths are represented as splay trees. It ensures that each access to the splay tree will take logarithmic amortized time. Infact they showed much stronger property that each access to an ST-tree, which invovles splaying on a series of splay trees, will also take $O(\log n)$ amortized time. ST-trees supports path-related queries (for example, $findmin(v)$) very efficiently.

Limitations of ST-trees: An important feature of an ST-trees is that a node does not need to access its middle children. Hence it suffices to have pointers from each middle children to its

parent. This simplifies the data structure, as there may be $\Theta(n)$ middle children for any node. However this restriction limits the scope of ST-trees. In particular, several application require information to be aggregated over the entire tree. In these cases, one does need access from a node to its middle children. Hence ST-trees does not support operation which depends on the entire tree. Such operations are called as sub-tree related operations (for example, adding a value to all vertices in the original tree).

One way to support sub-tree related operations using ST-trees is by storing information in each node about all its children. This will take time proportional to the degree of that vertex, which may be very expensive, unless we can guarentee that all vertices in the original forest have bounded degree. Clearly this assumption is not true in general. The usual solution to this problem is to use ternarization.

Ternarization: Whenever the input forest has a high degree (> 3) vertex, this operation replaces it by a chain of degree-three vertices. A common technique is to replace each vertex with degree $d > 3$ by a path with $d - 2$ vertices: each vertex in this path are connected to exactly one of the original neighbour except the first and the last vertices which are each connected to two of the original neighbours. See figure 2.

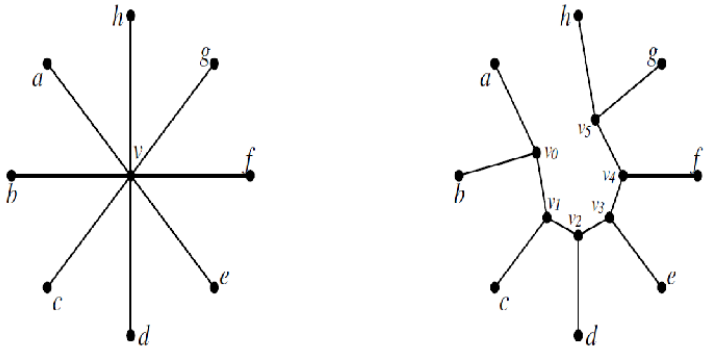

Figure 2: Example of ternarization (adapted from [3]). Each vertex v of degree greater than 3 are replaced by a chain of degree 3 vertices.

The drawback of using ternarization is that we must have to remember which vertices and edges are from the original tree and which are added as the special elements by this procedure.

We now consider a new data structure, **Top tree**, proposed by Alstrup et al. [4] which supports both path-related queries and sub-tree related queries efficiently. We will also see that top tree interfaces (small set of operations defined by the users) are also the easiest to use.

4 Top Trees [3] [4]

We have seen that ST-trees are based on path decomposition technique, i.e, the original tree is partitioned into disjoint paths, and each is represented as a binary tree. On the other hand top trees are based on Tree contraction technique. Instead of representing the tree itself, they represent a *contraction* (defined below) of the tree, which progressively transforms the original tree into smaller trees that summarizes the original information about the tree.

Top trees represent a collection of free trees (i.e., trees that are unrooted and undirected). Also top trees assume that there is a circular order of edges around every vertex of the original tree. See figure 3.

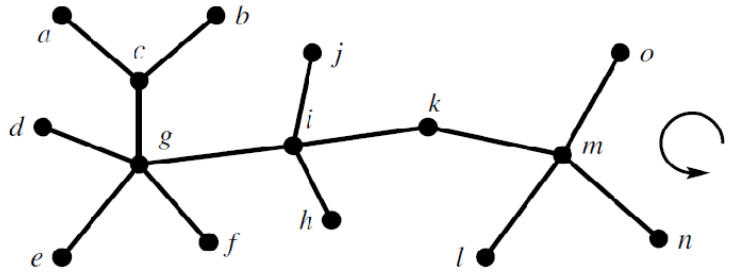

Figure 3: A free tree (adapted from [3]). Around each vertex edges are arranged in counterclockwise order.

Contraction of a tree (forest) is based on two operations: The rake and compress operation.

Rake operation: A degree-one vertex v with neighbour x is raked if the edge (v, x) and its successor (w, x) around x are replaced by a single edge, with end points w and x. We can also say this as: (1) the edge (v, x) is raked onto (w, x). (2) edge (v, x) is raked around x. (3) edge (w, x) receives (v, x) and (4) edge (w, x) is the target of a rake. Rake operations eliminates vertices of degree one.

Compress operation: A degree two vertex v is compressed if the two edges incident to it, (u, v) and (v, w), are replaced by a single edge (u, w). We also say that either edge is compressed at v. Clearly this operations removes degree-two vertices. See figure 4.

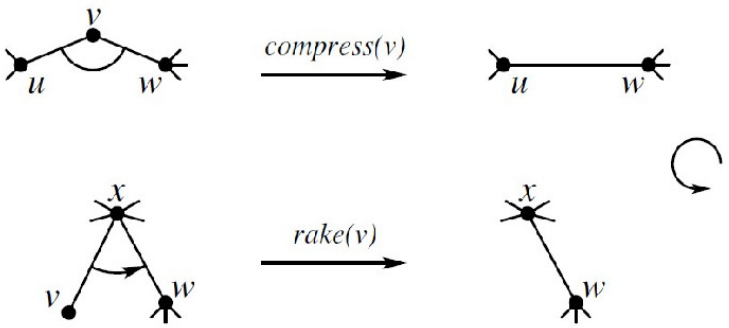

Figure 4: Top trees: Basic operations (adapted from [3]). (Above) compress removes degree two vertex v by replacing edge (u,v) and (v,w) by a single edge (u,w). (Below) rake removes degree one vertex by replacing edge (v,x) and (w,x) by a single edge (x,w).

Cluster: Consider each edge of the original tree as a cluster (called it base cluster). When two clusters are combined by either rake or compress, the result will be a new cluster, the parent cluster.

Observe that every cluster (base as well as the new one after rake or compress) have exactly two end points, and can therefore be consider as an edge of the contracted tree.

A top tree is a binary tree that embodies a contraction of a free tree into a single cluster via a sequence of rake and compress operations. Each Leaf of the top tree is a base cluster representing an original edge, and each internal node of the top tree is either a rake or a compress cluster. A node in the top tree aggregates information pertaining to all descendants; in particular, the entire original tree is represented at the root of the top tree. It is clear that if the top tree is representing a free forest in this way, then the number of roots of the top tree is equal to the number of non-trivial trees in the original forest. A tree with a single vertex is represented by an empty top tree.

To support $O(\log n)$ update and query time we are after the sequence of rake and compress operations that produces balanced top trees. Since when an edge is deleted or inserted in the original forest the leaf (base cluster) of the top tree is affected and we need to propagate the changes to the root of the top tree. One such sequence can be obtained as follows: work in rounds, and in each round perform a maximal set of independent moves (rake and compress), that is, each cluster participates in at most one move and no valid moves is left undone. Figure 5 shows a contraction of the free tree shown in figure 3 into a single cluster that obeys the above rules. Figure 5 also shows the corresponding top tree.

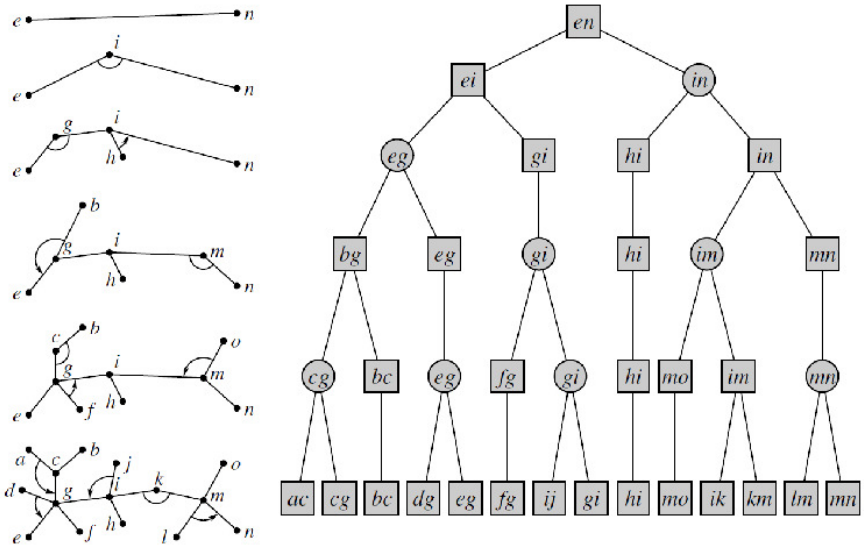

Figure 5: (Left) a contraction of a free tree (figure 3) to be read bottom-up. (On right) the corresponding top tree. (adapted from [3]). Circles represents rake nodes in the top tree; square represents base, dummy or compressed nodes, depending on whether the number of children is zero, one or two.

The example top tree in figure 5 has four types of nodes. A Base node, shown as a square with no children, A dummy node, shown as square with one child, represents an edge that was not involved in any move in the previous round, A compress node, shown as a square with two children, represents the compress of its two children (in top tree they can appear in any order). Finally, a rake node, shown as a square with two children, represents the rake of its left child onto the right

child. However dummy nodes in principle could be eliminated.

A cluster always has two end points. Therefore a cluster which in l^{th} level (from bottom, i.e, leaves are at level zero) in the top tree, can be thought off as an edge in the l^th round of contraction of the free tree. It can be naturally mapped to both a path and a subtree of the original free tree. The path is the one between its end points; the subtree is the one induced by all base clusters that descend from it. Call the vertices that belong to the subtree of a cluster C but are not the end points of C as internal vertices of C.

4.1 Operations Supported by Top Trees

The following are the three external operations that top tree suppports, which user can call directly:

- $C \leftarrow link(e)$: adds an edge $e = (v, w)$ to the forest and returns the base cluster representing the new edge e. If v and w are in the same component of the forest then the function does not add e and just returns a NULL.
- $cut(C)$: removes the edge represented by base cluster C from the forest.
- $C \leftarrow expose(v, w)$: takes one or two vertices as input and ensures that they are the endpoints of the root cluster of their trees. If v and w are in the same component then the function returns its root cluster; otherwise it returns a NULL.

The 3^{rd} operation is important because users are not allowed to traverse freely on top tree, they have only a direct access to its root. This simplifies implementation of various applications. Because of operation 3 this strategy is as powerfull as any other. For example assume that vertices v and w are in the same component and a user is interested in the path between v and w. Then she first call $expose(v, w)$ and look at the cluster returned. This function modifies the top tree in such a way that v and w becomes the end points of the cluster representing their tree (this can be ensured just by gauranteeing that vertex v and w do not disappear by any rake and compress operation during the tree contraction process) and retured this cluster. Suppose user is interested only in the tree containing vertex v. Then she first call $expose(v, \cdot)$ and look at the cluster returned, because again this function guaranteed that vertex v is one of the end points of the cluster representing its tree and it had retuned this cluster (the other end point of the cluster returned is any arbitrary vertex of the tree containig v).

To execute these operations, the clusters in the affected top trees must be rearranged. The exact information to be updated depends on the application the top tree is solving. It is the user who defines what information each cluster should have and how they should be updated. Therefore Alstrup et al. in [4] proposes four internal operations whose implementation detail user must have to define.

Internal Operation: These are the operations whose implementation details are to be defined by the users based on the application they want the top tree to solve. External operations uses internal operations. 4 internal operations defined in [4] is as follows:

1. $C \leftarrow create(e)$: makes a base cluster representing edge e;
2. $C \leftarrow join(A, B)$: performs a rake or compress of two adjacent clusters;

3. $(A, B) \leftarrow split(C)$: disassembles a rake or compress cluster and returns its children;

4. $destroy(C)$: eliminates a base cluster.

For simplicity we will assume that dummy nodes are not present (otherwise we must have to generalize join and split to allow cluster with single child which is duable). In [3] it has been shown that all these operations can be implemented in worst case $O(\log n)$ time. Assuming this we will now show how powerful and simple a top tree is in solving both path-related and sub-tree related queries in $O(\log n)$. However we will present the applications that uses only the first two internal operations.

Application 1: Maintain sum of the costs of all edges in a tree.
Store a single variable in each cluster. The *create* operation initializes this value as the cost of the corresponding edge. *join* operation store the sum of the values of the children in the new cluster. In the similar manner we can maintain the minimum or maximum edge cost of the entire tree just by changing *join* operation accordingly. Both *destroy* and *split* do nothing.

Application 2: Find the Length of a Given Path in the Tree.
We store a single value in each cluster, corresponding to the length of paths between its two end points. The *create* operation initializes this value as the length of the corresponding edge. When the two cluster $A = (u, v)$ and $B = (v, w)$ are compressed then *join* operation stores the sum of their values in the parent cluster. If cluster $A = (u, v)$ is raked around v onto cluster $B = (v, w)$, *join* copies the value stored in B to the parent cluster. Clearly these update rules ensures that the root of the top tree will contains the length of the path between its two end points. Thus to compute the length of the path between two arbitrary vertices v and w, users needs to first call $expose(v, w)$ and just reads the value of the cluster returned from $expose(v, w)$.

Application 3: Maintaining Diameter (i.e, the largest distance between two vertices in the tree).
Clearly this application needs information about paths and subtrees at the same time. Alstrup et al. [4] were the first to show that the diameter of a tree can be updated after a link or a cut in $O(\log n)$ time using top trees. For maintaining diameter of the tree, we maintain in each cluster $C = (v, w)$ the following values:

- $diam(C)$: diameter of the sub-tree represented by the cluster C;
- $length(C)$: length of the path between v and w;
- $max_v(C)$: maximum distance from v to a vertex in the sub-tree represented by C;
- $max_w(C)$: maximum distance from w to a vertex in the sub-tree represented bt C.

As we had already seen in application 2 how to maintain $length(C)$, we will concentrate on the other three values. $create(e)$ operation just creates a cluster $C = (v, w)$ and initializes all its 4 values with the length of the edge (v, w). When two adjacent clusters A and B with a common vertex v are combined into a cluster C, *join* operation sets,

$$diam(C) = max\{diam(A), diam(B), max_v(A) + max_v(B)\}$$

Let w be the end point of cluster C that belongs to B but not A. *join* update $max_w(C)$ as follows:

$$max_w(C) = max\{max_w(B), length(B) + max_v(A)\}$$

If C is a compressed cluster, it must have an end point u that belongs to A but not B then the value $max_u(C)$ must also be updated similarly. If the move was *rake*, v be the other end point of C which is updated as: $max_v(C) = max\{max_v(A), max_v(B)\}$.

5 Contraction-Based Top Trees

Renato F. Werneck in his PhD dissertation [3] presented a very simple implementation of the top tree interface that supports *link*, *cut* and *expose* in logarithmic time in the worst case. In this section we will see an overview of his result.

Height of the Top Tree is Bounded by $O(\log n)$:
In [3] top tree is implemented using tree contraction technique which works in round. In each round the set of moves (that is, *rake* and *compress*) performed in independent and maximal: each cluster can participate in atmost one move, and no legal move is left undone. We will first proof that this contraction rule results in a top tree with height at most $O(\log n)$.

Lemma 1 If a contraction procedure is guarenteed to eliminate a fraction $\alpha > 0$ of all nodes in each round, then the number of rounds in the contraction is at most $\log_{1/(1-\alpha)} n$.

Proof. Let after h rounds there are only two vertices (i.e., an edge) left. Then we have,

$$n(1-\alpha)^h \leq 2$$

Since we started with n vertices and a fraction of at most $(1-\alpha)$ remains after each round. By taking binary logarithm on both sides we have,

$$\log(n(1-\alpha)^h) \leq \log 2$$

$$\log n + h\log(1-\alpha) \leq 1$$

By rearranging the terms we have,

$$\begin{aligned} h &\leq \frac{1-\log n}{\log(1-\alpha)} \\ &= -\frac{\log n - log2}{\log(1-\alpha)} \\ &= -\frac{\log(n/2)}{\log(1-\alpha)} \\ &\leq \log_{1/(1-\alpha)} n \end{aligned}$$

The following 2 lemma proves that more than half of the vertices in any tree have degree one or two which is relevant for the contraction algorithm.

Lemma 2 Let n_i be the number of vertices with degree i on a free tree T with n vertices. If T has at least two vertices, then $\sum_{i=1}^{\infty} n_i(i-2) = -2$ holds.

Proof. From sum of degree theorem we have,

$$\sum_{i=1}^{\infty} in_i = 2(n-1)$$

since total number of edges is $(n-1)$ and in_i is the total degree of all vertices of degree i. We know that $n = \sum_{i=1}^{\infty} n_i$, thus we have,

$$\sum_{i=1}^{\infty} in_i = 2\Big(\sum_{i=1}^{\infty} n_i\Big) - 2$$

Thus we have,

$$\sum_{i=1}^{\infty} n_i(i-2) = -2$$

Lemma 3 In any free tree T, more than half of the vertices have degree one or two.

Proof. In the summation of Lemma 2, each degree-one vertex has a negative contribution of one unit, degree two vertices has no contributions at all, and degree greater than three vertices has a positive contributions. Since the result of the summation is a negative, the Lemma follows.

As every degree one vertex is a rake candidate and degree two vertex is a compress candidate Lemma 3 is relevant to the algorithm. Now without proof we are stating a Lemma from [3] which states that a constant number of vertices disappear after each round.

Lemma 4 [3] If $n \geq 3$, at least $1/6$ of the vertices disappear from one round to the next.

Together, Lemma 1 and 4 ensures that the height of the top tree constructed from the tree contraction procedure that works in maximal independent rounds is at most $\log_{6/5} n$, which is less than $3.802 \log n$.

Finally we are interested in the update problem: given a contraction C of a forest F, compute a contraction C' of F' that differs from F in exactly one edge (which has been added or deleted, due to *link* and *cut* repectively) with atmost $O(\log n)$ modifications. In $expose(v, w)$, F' differs from F in zero edges. We will handle this case separately at last.

Let us call C the original contraction and C' the new contraction. Solving the update problem requires "repairing" the original contraction by undoing some of the existing moves and performing new ones. The repairing rules are very simple: start from the bottom level and for each level just replicate the original moves that can be replicated, then perform the new moves until the over all moves in this round becomes maximal.

The first step of the algorithm, i.e., replicating original moves is done implicitly and can be done constant time. Only the second step, i.e., performing new moves is done explicitly. In [3] it had been shown that (1) the number of new clusters per level is constant after a *link* or *cut* (true for *expose* as well) and (2) the new clusters can be processed in constant time. Assuming the two claims we have:

Update algorithm to construct C' from C after a *link* or *cut* operation takes $O(\log n)$ time: As the number of level (rounds) in top tree in $O(\log n)$ and at each level we have to deal with constant number of new moves that can be processed in constant time. Thus the total running time of the update algorithm is $O(\log n)$. Similarly [3] shows that *expose* can be implemented in $O(\log n)$ time. Since exposing a vertex is to ensure that it does not disappear in the contraction. During the update algorithm one just marked the exposed vertex (or two vertices in case of $expose(v, w)$) and ensure that it can not disappear in the contraction.

References

[1] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *Journal of Computer and System Sciences*, **26(3)**: 362-391, (1983).

[2] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM*, **32(3)**: 652-686, (1985).

[3] Renato F. Werneck, Design and Analysis of Data Structures for Dynamic Trees, PhD Dissertation, *Princeton University*, June, (2006).

[4] S. Alstrup, J. Holm, M. Thorup, and K. de Lichtenberg, Maintaining information in fully dynamic trees with top trees *ACM Transactions on Algorithms*, **1(2)**: 243-264, (2005).