Advanced Data Structures Jan-Apr 2012

Lecture 15 — March 15, 2012

Lecturer: Venkatesh Raman *Scribe: Esha Ghosh and Sudeshna Kolay*

1 Overview

Like in the last lecture we will look at another data structure that supports dynamic operations on a forest of trees in worst case $O(\log n)$ time. Our final aim is to perform these operations efficiently in general graphs.

In general, data structures supporting dynamic operations can be classified into the following:

- **Fully dynamic data structures** which support both insertion and deletion of edges on the given graph.
- **Partially dynamic data structures** which support one of the two operations on edges.
- **Incremental data structures** which support insertion of edges.
- **Decremental data structures** which support deletion of edges.

2 Euler tree data structure

As before we want to maintain a forest of trees and represent this using Euler trees. This data structure supports the following operations:

1. *insert(,)*, which is a function that takes in as parameters 2 vertices and adds an edge between them in the graph. In the Euler tree data structure it is no longer necessary that one of the vertices has to be the root of a tree. This is a natural condition since we know that in fact the tree structure is still preserved on adding an edge between vertices of two distinct trees.
2. *delete(,)*, which deletes an existing edge between two vertices in a tree.
3. *path(,)*, which determines whether there is a path between two input vertices, ie whether the two vertices belong to the same tree in the forest.

For each tree in the forest we first duplicate each edge of the tree and find a Eulerian tour starting from the root of the tree. If the vertex set of a tree is of size n, we can depict the Eulerian tour of the tree as a sequence of length $2n - 1$ which starts and ends at the root of the tree. We store this sequence of vertices (with copies for repititions in the sequence) in a balanced binary search tree with the distance from the start of the sequence as the key. This is what we call Eulerian tree.

For each tree in the forest, we will maintain an array for the vertices. Each element of this array stores an array of pointers to all the occurrences of the corresponding vertex in the Eulerian tree.

Note that even if multiple pointers have to be maintained for a vertex all of them point to vertices in the same Eulerian tree.

This data structure supports the following operations in worst case $O(\log n)$ time:

1. *path(x,y)*. We find the root of the Eulerian tree containing x and that containing y. If they have the same root we know they belong to the same tree and hence have a path between them. We know that in binary search trees we can do all this in $O(\log n)$ time.

2. *findroot(u)* To find the root of a tree in the given forest, given any node in the tree, we find out the leftmost leaf of the corresponding Eulerian tree. Since the Eulerian tree has the distance from start of the Eulerian tour as the key, the root is the minimum by definition.

3. *makeroot(x)*. This operation makes x the root of its tree in the forest. This means the Eulerian sequence of the tree T_x containing x has to be altered such that the sequence starts and ends at x.

   Suppose $T_x = (r, u, \ldots v, x, \ldots x, \ldots x, \ldots w, r)$ be the sequence for T_x.

   Then, $T'_x = (x, \ldots x, \ldots x, \ldots w, r, u, \ldots v, x)$ is a rearrangement of T_x where x has been made the root.

   We will call the corresponding Eulerian trees by the same names as the Eulerian sequences. To get the Eulerian tree T'_x we first delete the last occurrence of x, which is in fact the rightmost leaf of T_x. Then we split T_x at x. Let A and B be the left and right subtrees after split. We first concatenate x with B to get a tree say C. Then we concatenate C with A and finally the resulting tree with a copy of x. This corresponds to the Eulerian tree T'_x. Since only a constant number of splits and concatenates are performed for this operation this can be done in $O(\log n)$ time.

4. *insert(x,y)*. We can check if x and y belong to two distinct trees in $O(\log n)$ time using *path(x,y)*. If not then we can insert the edge (x, y). We will look at what it means to add the edge (x, y) in terms of the Eulerian sequences for the containing trees of the two vertices. Suppose $T_1 = (r_1, \ldots y, \ldots y, \ldots y, \ldots y, \ldots r_1)$ be the sequence for the tree containing y.

   $T_2 = (r_2, v, \ldots x, \ldots x, \ldots x, \ldots r_2)$ be the sequence for the tree containing x.

   Let T'_2 be the Eulerian tree when x has been made the root of its containing tree, using the operation *makeroot(x)*.

   $T'_1 = (r_1, \ldots y, \ldots y, \ldots y, \ldots y, T'_2, y, \ldots r_1)$ is a Eulerian sequence of the tree resulting from adding the edge (x, y) and considering it to be rooted at r_1.

   We split T_1 at the last occurrence of y to get left and right subtrees A and B. We concatenate A with y. Attach T'_2 as a subtree of y to get a tree T. Finally we concatenate T with a copy of y followed by B. This gives us the Eulerian tree T'_1.

   Again only a constant number of splits and concatenations are done and the total time for this operation is $O(\log n)$.

5. *delete(x,parent(x))*. Again we analyse what to do with the Eulerian sequences. To get the resulting Eulerian tree we need to split at the first occurrence of x, and then at the last occurrence of x. Then we concatenate the left and right subtrees obtained respectively from the two splits. We also concatenate together all other resulting subtrees. This gives us the

required Eulerian trees containing x and its parent. Again, the total time for this operation is $O(\log n)$.

Notice that a consecutive subsequence of any Eulerian sequence is represented by a subtree in the Eulerian tree. Therefore, if the number of distinct vertices in the subsequence is m then the number of vertices in the Eulerian subtree is $2m - 1$. Using this relation we can maintain aggregate values at internal nodes of the eulerian trees and relate them to actual aggregate values on the trees of the forest. Therefore we can count the number of nodes in a tree containing x, by maintaining the count at the Eulerian subtrees and finally relating the number to the actual number of distinct vertices. More simply, we can find the minimum weighted node in the tree containing x by maintaining the information at the internal nodes of the Eulerian tree.

3 Fully Dynamic Algorithm for Graph Connectivity

We present a simple $O(\log^2 n)$ amortized time deterministic fully dynamic connectivity algorithm described in [1].

High-Level Idea:
The dynamic algorithms maintains a spanning forest F for the input graph G using an Euler-Tour tree. using this simple structure the following operations are easy:

- **Insert**(x, y): Given that (x, y) was not an edge in G, check if x and y are connected in F; if not add (x, y) to F. Otherwise if x and y belong to the same tree in F then just update the adjacency matrix of G.
- **Path**(x, y): Check if x and y belong to the same tree in F. Note that it is not possible that x, y where connected in G, but not in F.
- **Delete**(x, y): This is easy when (x, y) is a non-tree edge in F. We just need to update the information in the adjacency matrix of G.

The main challenge is when (x, y) is a tree edge. We cannot tell whether deleting (x, y) will disconnect x and y in G. The deletion splits some tree in F, but its correponding components in G might be connected. To deal with deletes, we will modify our initial idea by hierarchically dividing G into $\log n$ subgraphs and then storing a maximum spanning forest for each subgraph.

3.1 Algorithm

To implement this idea, we partition the edges into $\log n$ levels. We start by assigning a *level* to each edge. The level of an edge is an integer between 0 and $\log n$ inclusive that can only increase over time. We define G_i to be the subgraph of G composed of edges at level $\geq i$. Note that $G_0 = G$. Let F_i be the spanning forest of G_i.

During the execution of this algorithm we will maintain the following two invariants:

- Invariant 1: $F_0 \supseteq F_1 \supseteq F_2 \supseteq \ldots \supseteq F_{\log n}$. In other words, F_i is the maximum(w.r.t level) spanning forest of G_i.

- Invariant 2: Number of vertices in any tree in F_i is $\leq \lfloor n/2^i \rfloor$.

The algorithms associates with each edge e a lebel $l(e) \leq l_{max} = \lfloor \log n \rfloor$. We will also maintain the adjacency matrix for each G_i.

Initially all edges have level 0. Hence both invariants are satisfied. Notice that since levels of edges are never decreased, so we can have at most l_{max} increases per weight. Before describing how to delete an edge in detail, let us quickly revisit the other functions.

Insert(x, y): Give that level$(x, y) = 0$ and insert in G_0.

Path(x, y): Check in G_0.

Delete(e): If e is a tree-edge then we need to find a *replacement edge* reconnecting F at the hoghest possible level. Since F is a maximum spanning forest, we know that the replacament edge can come from level $l(e)$ or lower. So now we need to describe the function **Replacement** $((x, y), i)$ which finds a replacement edge of the highest level $\leq i$, if any.
Replacement $((x, y), l)$: let T_x and T_y be the trees in F_l containing x and y respectively. WLOG, let $|T_x| \leq |T_y|$. Now, note that before deleting the edge (x, y), $T = \{T_x \cup (x, y) \cup T_y\}$ was a tree on level l with twice as many vertices as T_x.

We will look for this replacement edge by doing the following:

Replacement $((x, y), i)$**:**

---

1. By Invariant 2, we know that $|T| \leq \lfloor n/2^i \rfloor$, so $|T_x| \leq \lfloor n/2^{i+1} \rfloor$. This means that we can afford to push all edges of T_x of level i to level $i+1$, maintaining our invariants, so as to make T_x a tree in F_{i+1}.
2. For each non-tree edge f at level i incident to T_x:
   (a) If f both endpoints of f are in T_x, make $level(f) = i + 1$ and insert f in G_{i+1}.
   (b) If f connects T_x and T_y, insert it as a replacement edge in all levels i and below and STOP.
   (c) If no level i edges are left, call *Replacement* $((x, y), i - 1)$.
   (d) If $i = 0$, return (No replacement edge for (x, y)) and STOP.

3.2 Analysis

Implementation:

- ET-trees are maintained for each F_i.
- Along with that, maintain a list of non-tree edges of every vertex.
- We will also maintain the adjacency matrix for each G_i.

Amortized cost for each operation:

Insert: When and edge is inserted at level 0, the direct cost is $O(\log n)$.

Delete: Deleting a non-tree edge takes $O(\log n)$ time. Each tree-edge deletion results in

1. $O(\log n)$ direct insertions (of the replacement edge to the levels below), each costing $O(\log n)$, so totally $O(\log^2 n)$ and
2. A number of insertions of edges to higher levels (including in recursive calls), each of the insertions costing $O(\log n)$.

As each edge can move to at most $O(\log n)$ levels, total number of insertions of type (2) above over all operations is $O(m \log n)$, each costing $O(\log n)$, and so a total cost of $O(m \log^2 n)$ over all operations. (Here m is the maximum number of edges in the graph at any point of time.) Therefore total time for t deletions is $= O(t \log^2 n + m \log^2 n)$. Thus amortized time for each deletion is $O(\log^2 n)$, when $t \gg m$.

Improving the query time

We can apply the $\Theta(\log n)$-ary ET trees mentioned in section 2 to the spanning forest F_0 and maintain BST's for all other levels. The cost of a split/concatenate on the k-ary trees is: $O(\log n / \log k)$. Hence the total cost of a query is $O((\log n/\log k) * k) + O(\log^2 n)$. Choosing $k = \Theta(\log n)$ gives the cost as $O(\log^2 n/\log\log n) + O(\log^2 n)$, which is subsumed by the $O(\log^2 n)$ factor from above.

Note that choosing the parameter k as $\Theta(\log n)$ is optimal as if we increase it further, this additive cost would not be subsumed by the $O(\log^2 n)$ factor.

3.3 Decremental Algorithm for Minimum Spanning Forest (MSF)

MSF Algorithm can be obtained from the previous algorithm by some simple changes.

- The initial spanning forest F_0 should be a minimum weight spanning forest instead of maximum spanning forest.
- In the function *Replace()* push the minimum weight edges in increasing order of weight, i.e, push the lighest edge first, then the second lightest and so on.

References

[1] Holm, Jacob and de Lichtenberg, Kristian and Thorup, Mikkel, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, J. ACM, 48(4):723-760, 2001.