6.851: Advanced Data Structures Spring 2007

Lecture 5 — February 26, 2007

Prof. Erik Demaine *Scribe: Katherine Lai*

1 Overview

In the last lecture we discussed the link-cut tree: a dynamic tree that achieves $O(\lg n)$ amortized time per operation. In this lecture, we leave the BST model and enter the pointer-machine model. In this model, we are allowed to have nodes which each hold $O(1)$ pointers and $O(1)$ fields (e.g. integers). Our machine is also allowed to keep track of $O(1)$ "fingers" to nodes at a time. The operations we are allowed to do are to copy a finger, follow a pointer to a node, and to retrieve or change the contents of a node's fields. So, the BST model allows for a subset of the data structures allowable in the pointer-machine model. We will focus on data structures in this model that solve the dynamic connectivity problem.

2 Euler tour trees

The Euler tour tree data structure is due to Henzinger and King in [1]. An Euler tour of a graph is a path that traverses each edge exactly once. In the context of a tree, we say that each edge is bidirectional, so the Euler tour is the path along the tree that begins at the root and ends at the root, traversing each edge exactly twice — once to enter the subtree at the other endpoint and once to leave it. You can think of an Euler tour as just being a depth first traversal where we return to the root at the end.

We will store the Euler tour of the tree as a balanced binary search tree with one node for each time a node in the represented tree was visited, with each node in the tree keyed by its time of visit. Each node in the represented tree will also hold pointers to the nodes representing the first and last time it was visited.

While the link-cut trees we discussed last lecture are good for maintaining aggregates on paths of a tree (making it a good choice data structure in network flow algorithms), Euler tour trees are better at keeping aggregate information on subtrees. We will use this feature of Euler tour trees toward the end of the lecture notes.

We want Euler tour trees to be able to perform the same three operations as link-cut trees:

- **Findroot(v)**
  In the Euler tour, the root will be visited first (and last), so we simply have to find the min (or max) in the balanced BST.

- **Cut(v)**
  The Euler tour of v's subtree will form one contiguous block in the sequence of visitations for the whole tree, starting and ending with v. The Euler tour will not visit any part of v's

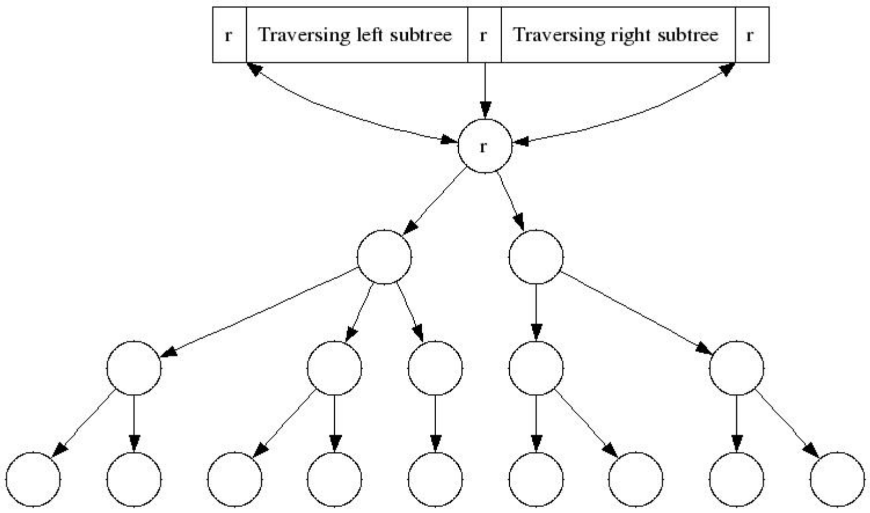

Figure 1: We represent a tree's Euler tour as a sequence of visitations starting and ending at the root. Pointers from that sequence point to the node, and pointers from the node point to the first and last visitations. (Only the pointers relating to the root are shown.) The sequence of visitations is then stored in a balanced binary tree (not shown).

subtree otherwise, so we simply split the Euler tour BST before the first visit to v and after the last visit to v. This gives us the Euler tour tree of v's subtree and two trees that cover the sequence before and and after v's subtree. We concatenate those last two trees, and we're done.

- **Link(v,w)**
  As a child of w, we need v's subtree to be traversed immediately after and immediately before visits to w. This can be accomplished by splitting w's Euler tour tree after the first visit to w, and then concatenating the left piece, v's Euler tree, a new tree of a singleton node of just w, and the right piece. (It is also equivalent to choose to do the first split immediately before the last visit to w and to insert the singleton node on the opposite side of v's Euler tour tree because children are not ordered in the represented tree.)

The above operations only use the basic BST operations that take $O(\lg n)$ using any common balanced BST data structure, so all of them run in $O(\lg n)$ time.

3 Dynamic Graph Problems

We wish to maintain an undirected graph subject to vertex insertion and deletion, and edge insertion and deletion. Deleting a vertex also deletes all incident edges (or alternatively the vertex must have no incident edges to be deleted). Since a data structure that is never observed can perform any operation in zero time, we will also say that the graph can be queried in some way.

3.1 Connectivity

The query type we focus on in this lecture is connectivity. Given a graph G, we wish to be able to handle queries of the form **Connected(v,w)**, which returns true if and only if there is a path from v to w in G. Also, we would like to handle queries of the form **Connected(G)**, which returns whether or not G is connected. We have already seen how to achieve $O(\lg n)$ time per operation when G is a tree — use the link-cut tree data-structure and compare **Findroot(v)** with **Findroot(w)**. There have been several other results for graphs that are not forests:

- $O(\lg n)$ update and query for plane graphs [2].
- $O(\lg n(\lg\lg n)^3)$ update
  $O(\lg n/\lg\lg\lg n)$ query [3]
- $O(\lg^2 n)$ update
  $O(\lg n/\lg\lg n)$ query [4]
- $O(\lg n \cdot x)$ update requires $\Omega(\lg n/\lg x)$ query for $x > 1$ [5].
- **OPEN:** Is $o(\lg n)$ update and $O(\text{poly}(\lg n))$ query achievable?
- **OPEN:** Is $O(\lg n)$ update and query for general graphs achievable?

For the first result, a *plane* graph is similar to a *planar* graph, but the planar embedding is fixed (we are not allowed to change the embedding as we add edges to the graph). Also, all the bounds above are amortized. There has only been one significant worst-case bound proven, done in [6]. They showed that $O(\sqrt{n})$ update and $O(1)$ query is achievable. It is an open problem as to whether we can achieve $O(polylog(n))$ updates and queries in the worst case. Note that the bounds in [3, 4, 6] are in a sense optimal in that they achieve the trade-off bounds shown in [5].

There are some modifications we can play in the types of operations we allow to be done to our graph. For *incremental* dynamic graphs, we are never allowed to delete edges or vertices. For this problem, we know that we can achieve $O(\alpha(n))$ update and query by using the union-find data structure ($\alpha(n)$ being the inverse Ackermann function). It is also known that achieving $\Theta(x)$ update requires $\Theta(\lg n/\lg x)$ for $x > 1$. This has also been achieved. For *decremental* dynamic graphs we allow deletions but no insertions. In [7] it was shown how to achieve $O(m\lg n + n\text{poly}(\log n))$ for all updates, and $O(1)$ query (where m is the number of edges).

4 Dynamic Connectivity

Now we will focus on the explanation of the dynamic connectivity algorithm described in [4]. The high level idea is that we will store the spanning forest using Euler tour trees. We will hierarchically divide the connected components into $\lg n$ levels of spanning forests of subgraphs.

The *level* of an edge is some integer between 0 and $\lg n$, and it only decreases over time. G_i is the subgraph consisting of edges that are at level i or less. Thus $G_{\lg n} = G$. The F_i will also be the spanning forest of G_i. We will keep two invariants during the execution of the algorithm:

Invariant 1: Every connected component of G_i has at most 2^i vertices.

Invariant 2: $F_0 \subseteq F_1 \subseteq \ldots \subseteq F_{\lg n}$. In other words, $F_i = F_{\lg n} \bigcap G_i$, and $F_{\lg n}$ is the minimum spanning forest of $G_{\lg n}$, using edge levels as weights.

We will also keep an adjacency matrix for each G_i. There are three operations we would like our data structure to support:

- **Insert(e=(v,w))**
  We will set the level of e to $\lg n$ and update the adjacency lists of v and w. Also, if v and w are in separate connected components of $F_{\lg n}$ then we add e to $F_{\lg n}$ (we can tell this by calling FindRoot on v and w.
- **Delete(e=(v,w))**
  First we remove e from the adjacency lists of v and w. Then, we do the following:
  - if e is in $F_{\lg n}$
    - delete e from F_i for $i \geq$level(e)
    - look for a replacement edge to reconnect v and w
      - the replacement edge cannot be at a level less than level(e) by invariant 2 (each F_i is a minimum spanning forest).
      - We will start searching for a replacement edge at level(e) to preserve the second invariant.
    - for $i =$level(e),...,$\lg n$:
      - Let T_v be the tree containing v and T_w the tree containing w.
      - Relabel v and w so that $|T_v| \leq |T_w|$.
      - By invariant 1, we know that $|T_v| + |T_w| \leq 2^i \Rightarrow |T_v| \leq 2^{i-1}$ (This means we can afford to push all the edges of T_v down to level $i-1$).
      - for each edge $e' = (x, y)$ with x in T_v and level(e') $= i$
        - if y is in T_w: add (x, y) to $F_i, F_{i+1}, \ldots, F_{\lg n}$ and stop.
        - else set level(e') to $i-1$.
- **Connected(v,w)** Instead of storing the nodes in $F_{\lg n}$ using a balanced binary search tree, we can use a B-tree with a branching factor of $O(\lg n)$, which gives us a query time of $O(\lg n / \lg\lg n)$. Using this tree means that the update time is $O(\lg^2 n / \lg\lg n)$ for our other two operations.

We have to augment our Euler tour trees to do delete so that we can test if $|T_v| \leq |T_w|$ in $O(1)$ time (this part is standard and easy). We also would like to know for each node v in the minimum spanning forest F_i whether or not v's subtree contains any nodes incident to level-i edges. We can find the next level-i edge incident to a node in T_v in $O(\lg n)$ time using successor, jumping over empty subtrees. There is also an $O(\lg n)$ charge to each edge's decrease in level.

The total cost of delete is then $O(\lg^2 n)$ since each inserted edge is charged $O(\lg n)$ times. The remaining part of delete also takes $O(\lg^2 n)$ time since we have to delete the edge from at most $O(\lg n)$ levels, and deleting from each level takes $O(\lg n)$.

4.1 Other Dynamic Graph Problems

4.1.1 Minimum Spanning Forest

Here we would like an MST for each connected component of the graph, each represented as a dynamic tree. An amortized solution to this was described in [4] that achieves $O(\lg^4 n)$ update. A worst case bound of $O(\sqrt{n})$ for general graphs was achieved in [6], and $O(\lg n)$ for plane graphs was achieved in [8]. The problem of determining graph bipartiteness can also be solved by reduction to the minimum spanning forest problem.

4.1.2 k-connectivity (vertex or edge)

A pair of vertices (v, w) is *k-connected* if there are k vertex-disjoint (or edge-disjoint) paths from v to w. A graph is k-connected if each vertex pair in the graph is k-connected. To determine whether the whole graph is k-connected can be solved as a max-flow problem. An $O(\sqrt{n}\text{poly}(\lg n))$ algorithm for $O(\text{poly}(\lg n))$-edge-connectivity was shown in [9]. There have been many results for k-connectivity between a single pair of vertices:

- $O(\text{poly}(\lg n))$ for $k = 2$ [4]
- $O(\lg^2 n)$ for planar, decremental graphs [10]

The above are amortized bounds. There have also been worst case bounds shown in [6]:

- $O(\sqrt{n})$ for 2-edge-connectivity
- $O(n)$ for 2-vertex-connectivity and 3-vertex-connectivity
- $O(n^{2/3})$ for 3-edge-connectivity
- $O(n\alpha(n))$ for $k = 4$
- $O(n \lg n)$ for $O(1)$-edge-connectivity
- **OPEN:** Is $O(\text{poly}(\lg n))$ achievable for $k = O(1)$? Or perhaps $k = \text{poly}(\lg n)$?

4.1.3 Planarity Testing

For this type of query, we would like to ask if inserting some edge $e = (v, w)$ into our graph violates planarity. It was shown in [11] that we can do this in $O(n^{2/3})$. In [12] it is shown how to achieve $O(\lg^2 n)$ worst case for a fixed embedding. La Poutré showed an $O(\alpha(m, n) \cdot m + n)$ algorithm for a total of m operations with an incremental graph in [13].

Planarity is just one type of minor-closed property, i.e. by Kuratowski's theorem we know that a graph is planar if and only if it does not contain a subgraph isomorphic to K_5 or to $K_{3,3}$. In general, it is an open problem to test whether or not a graph has a fixed minor. A minor of G is a graph that can be obtained by a sequence of edge deletions and contractions. A property is minor-closed if G having the property implies that any minor of G must also have the property. It was shown in [14] that G has a certain minor-closed property if and only if it excludes a finite set of minors.

References

[1] Monika Rauch Henzinger, Valerie King: Randomized dynamic graph algorithms with polylogarithmic time per operation. STOC 1995: 519-527

[2] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, Moti Yung: Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph. J. Algorithms 13(1): 33-54 (1992)

[3] Mikkel Thorup: Near-optimal fully-dynamic graph connectivity. STOC 2000: 343-350

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

[5] Mihai Patrascu, Erik D. Demaine: Lower bounds for dynamic connectivity. STOC 2004: 546-553

[6] David Eppstein, Zvi Galil, Giuseppe F. Italiano, Amnon Nissenzweig: Sparsification - a technique for speeding up dynamic graph algorithms. J. ACM 44(5): 669-696 (1997)

[7] Mikkel Thorup: Decremental Dynamic Connectivity. J. Algorithms 33(2): 229-243 (1999)

[8] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, Moti Yung: Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph. J. Algorithms 13(1): 33-54 (1992)

[9] Mikkel Thorup: Fully-dynamic min-cut. STOC 2001: 224-230

[10] Dora Giammarresi, Giuseppe F. Italiano: Decremental 2- and 3-Connectivity on Planar Graphs. Algorithmica 16(3): 263-287 (1996)

[11] Zvi Galil, Giuseppe F. Italiano, Neil Sarnak: Fully Dynamic Planarity Testing with Applications. J. ACM 46(1): 28-91 (1999)

[12] Giuseppe F. Italiano, Johannes A. La Poutré, Monika Rauch: Fully Dynamic Planarity Testing in Planar Embedded Graphs (Extended Abstract). ESA 1993: 212-223

[13] Johannes A. La Poutré: Alpha-algorithms for incremental planarity testing (preliminary version). STOC 1994: 706-715

[14] Neil Robertson, Paul D. Seymour: Graph Minors. XX. Wagner's conjecture. J. Comb. Theory, Ser. B 92(2): 325-357 (2004)