6.851: Advanced Data Structures Spring 2012

Prof. Erik Demaine

Lecture 20 — May 3, 2012

1 Overview

Last lecture we covered dynamic trees, also known as link-cut trees. Link-cut trees are able to represent a dynamic forest of rooted trees in $O(\log n)$ amortized time per operation.

In this lecture we will see how to maintain connectivity information for general graphs. We will start by examining a simpler, although not strictly better, alternative to link-cut trees known as Euler-tour trees. Then, for the special case of *Decremental Connectivity*, in which edges may only be deleted, we will achieve constant runtime. We will then use Euler-tour trees to achieve dynamic connectivity in general graphs in $O(\log^2 n)$ time. Finally we will survey some of what is and is not known for dynamic graphs. The next class will be about lower bounds.

2 Dynamic Connectivity

Our goal is to maintain an *undirected* graph subject to:

- `insert`/`delete` of edges or vertices (with no edges).
- `connected`(u, v): whether u and v are connected, on general undirected graphs that are subject to vertex and edge insertion and deletion.
- Another possible query is to determine whether the entire graph is connected; though this may seem easier, the current lower and upper bounds are the same as for pairwise connectivity.

We will focus on the former type of query.

The different types of updates that we will consider are:

- **Fully Dynamic**. Supports insertion and deletion of vertices and edges.
- **Incremental**. Supports only insertions of vertices and edges.
- **Decremental**. Supports only deletions of vertices and edges.

2.1 Results in Dynamic Connectivity

There are several results for dynamic connectivity in specific types of graphs.

- Trees. For trees, we can support $O(\log n)$ time for queries and updates using Euler-Tour trees or Link-Cut trees. If the trees are only decremental, then we can support queries in constant time, as we will see in this lecture.
- Planar graphs. Eppstein et al. have proven that we can attain $O(\log n)$ queries and updates [2].

Here are some results for general dynamic graphs.

- In 2000 Thorup showed how to obtain $O(\log n(\log\log n)^3)$ updates and $O(\log n/\log\log\log n)$ queries [3].
- Holm, de Lichtenberg, and Thorup obtained $O(\log^2 n)$ updates and $O(\log n/\log\log n)$ queries [4].
- For the incremental data structure, the best result we have is $\Theta(\alpha(m,n))$ using union-find [14].
- For the decremental data structure, we have a $O(m\log n + \text{polylog}n + \ \#\text{queries})$ solution, which essentially amounts to an $O(\log n)$ solution for dense graphs [15].
- **OPEN**: It remains an open problem whether $O(\log n)$ queries and updates are attainable for general graphs.

All of the aforementioned runtimes are amortized. Less is known about worst-case bounds:

- Eppstein et al. showed that an $O(\sqrt{n})$ worst-case update and $O(1)$ query time is achievable [6].
- **OPEN**: It remains an open problem whether we can achieve $O(\text{poly}\log n)$ worst-case updates and queries.

Patrascu and Demaine proved two lower bounds on dynamic connectivity, namely that an $O(x\log n)$ update requires an $\Omega(\log n/\log x)$ query time, and an $O(x\log n)$ query requires an $\Omega(\log n\log x)$ update time for $x > 1$ [5]. We will be focusing more on lower bounds next lecture. Note that [3, 4, 6] are all optimal in a sense, by the trade-off bounds shown in [5].

We end with an **OPEN** question: are $o(\log n)$ update and $\text{poly}\log n$ query achievable?

3 Euler-Tour Trees

Euler-Tour trees are due to Henzinger and King [1] and are an alternative to link-cut trees for representing dynamic trees. Euler-tour trees are simpler and easier to analyze than link-cut trees, but do not naturally store aggregate information about paths in the tree. Euler-tour trees are well suited for storing aggregate information on subtrees, which is a feature we will use in the next section.

The idea behind Euler-Tour trees is to store the Euler tour of the tree. In an arbitrary graph, an Euler tour is a path that traverses each edge exactly once. For trees we say that each edge is bidirectional, so the Euler tour of a tree is the path through the tree that begins at the root and ends at the root, traversing each edge exactly twice — once to enter the subtree, and once to exit it. The Euler tour of a tree is essentially the depth-first traversal of a tree that returns to the root at the end. The correspondence between a tree and its Euler tour is shown in Figure 1.

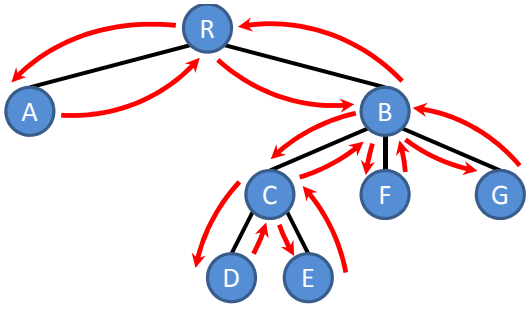

Figure 1: An example tree and its Euler tour. The order in which nodes in the tree are visited is indicated by the curved red arrows. The Euler tour of the tree shown here corresponds to the sequence: *R A R B C D C E C B F B G B R*.

In an Euler-Tour tree, we store the Euler tour of a represented tree in a balanced binary search tree (BST). For some represented tree, each visit to a node in that tree's Euler tour corresponds to a node in the BST. Each node in the represented tree holds pointers to the nodes in the BST representing the first and last times it was visited. As an example, the pointers between the root of the tree in Figure 1 and its Euler-tour node visitations is shown in Figure 2.

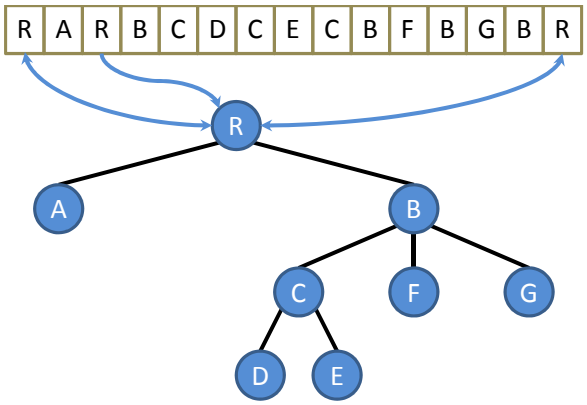

Figure 2: We represent a tree's Euler tour as a sequence of visitations starting and ending at the root. Pointers from that sequence point to the node, and pointers from the node point to the first and last visitations. In this figure only the pointers relating to the root are shown.

An Euler-Tour tree supports the following operations:

`FindRoot`(v) Find the root of the tree containing node v. In the Euler tour of a tree, the root is visited first and last. Therefore we simply return the minimum or maximum element in the BST.

`Cut`(v) Cut the subtree rooted at v from the rest of the tree. Note that the Euler tour of v's subtree is a contiguous subsequence of visits that starts and ends with v, contained in the sequence of visits for the whole tree. To cut the subtree rooted at v, we may simply split the BST before its first and after its last visit to v. This splitting gives us the Eulertour of the tree before reaching v, the tour of v's subtree, and the tour of the tree after leaving v. Concatenating the first and last pieces together, and possibly deleting one redundant visitation between the end of the first piece and beginning of the last, gives us our answer.

`Link`(u, v) Insert u's subtree as a child of v. In the resulting Euler tour, we need to traverse u's subtree immediately after and immediately before visits to v. Therefore we will split v's traversal before the last visit to v, and then concatenate onto the left piece a singleton visit to v, followed by the Euler tour of u's subtree, followed by the right piece.

`Connectivity`(v, w) Two nodes are connected if they are in the same rooted tree, so it suffices to check that `FindRoot`(v) = `FindRoot`(w).

`Subtree Aggregate`(v) Compute the min, max, sum, etc. of the nodes in the subtree rooted at v. This is accomplished through a range query in the BST between the first and last occurrence of v.

Each of these operations performs a constant number of search, split, and merge operations on the Euler-tour tree. Each of these operations takes $O(\log n)$ per operation on a balanced BST data structure. Consequently, the total running time for Euler-Tour trees is $O(\log n)$ per operation. If we use a B-tree with fanout of $\Theta(\log n)$ instead of a balanced BST in our Euler-tour trees, we can achieve $O(\log n / \log\log n)$ searches (from the depth of the tree) and $O(\log^2 n / \log\log n)$ updates (from the depth times the branching factor).

3.1 Decremental Connectivity in Trees

We are going to achieve constant amortized time per operation if we are only working with decremental trees. We assume that all edges eventually get deleted for our amortized analysis to work. We make a few observations:

1. We can easily get $O(\log n)$ per operation by using link-cut or Euler-tour trees.
2. To reduce this, we use leaf trimming: we cut below the maximally-deep nodes that have at least $\log n$ descendants. The top tree will have $O(n/\log n)$ branching nodes; compressing paths results in a tree of size $O(n/\log n)$. We can use a link-cut or Euler-tour tree on this structure, giving a total time of $O(n)$ for queries on the top tree.
3. The bottom trees have only $O(\log n)$ edges, so we can store the edge set as a bit vector (only one machine word). We also preprocess a lookup table storing the path to the root from an any vertex as a bit vector on edges. This allows us to determine whether there is still a path to the root in constant time (compute the bitwise AND between the path to the root and the current edge set).

4. Finally, we need to answer queries about getting between nodes in paths that we compressed in 2. We can split each path into $O(n/\log n)$ chunks of size $\log n$. Each chunk can be stored as a bit vector. Then, we can answer queries within a chunk by masking the bit vector. When we answer queries across chunks, we can treat each chunk in-between as existing if the chunk is the all 1 vector, and not existing otherwise, then use a structure from 1. Thus, we can answer all path queries in $O(\log n)$ time, and like before this gives us $O(n)$ time for all the queries.

3.2 Dynamic Connectivity in $O(\log^2 n)$

We will examine the dynamic connectivity algorithm described in [4], which achieves an $O(\log^2 n)$ amortized bound.

The high-level idea for this data structure is to store a spanning forest (one spanning tree for each connected component) for the input graph G using an Euler-Tour tree. With this idea alone, we see that we can answer connectivity queries by checking if both input nodes are in the same tree; this can be done by performing find root on each and seeing whether they are the same, since the root is a canonical name for a tree. An `Insert`(u, v) can be done by first checking if u and v are in the same tree, and if not adding (u, v) to the spanning forest. The problem with this approach is `Delete`(u, v), and in particular we cannot tell whether deleting (u, v) will disconnect u and v. To deal with deletes, we will modify our initial idea by hierarchically dividing G into $\log n$ subgraphs and then storing a minimum spanning forest for each subgraph.

To implement this idea, 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 decrease over time. (We will use the level of each edge as a charging mechanism.) We define G_i to be the subgraph of G composed of edges at level i or less. Note that $G_{\log n} = G$. Let F_i be the spanning forest of G_i. We will implement our F_i's using Euler-Tour trees built on B-trees.

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

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

Invariant 2 $F_0 \subseteq F_1 \subseteq F_2 \subseteq \ldots \subseteq F_{\log n}$. In other words, $F_i = F_{\log n} \bigcap G_i$, and $F_{\log n}$ is the minimum spanning forest of $G_{\log n}$, where the weight of an edge is its level.

We will also maintain the adjacency matrix for each G_i.

We implement the three operations on this data structure as follows:

`Connected`(u, v) Check if vertices u and v are connected. To do this, we first query $F_{\log n}$ to see if u and v are in the same tree. This can be done by checking $F_{\log n}$ if `Findroot`(u) = `Findroot`(v). This costs $O(\log n/\log\log n)$ using B-tree based Euler-Tour trees.

`Insert`$(e = (u, v))$ Insert edge $e = (u, v)$ into the graph. To do this, we first set the level of edge e to $\log n$ and update the adjacency lists of u and v. If u and v are in separate trees in $F_{\log n}$, add e to $F_{\log n}$. This costs $O(\log n)$.

`Delete`$(e = (u, v))$ Remove edge $e = (u, v)$ from the graph. To do this, we first remove e from the adjacency lists of u and v. If e is not in $F_{\log n}$, we're done. Otherwise:

1. Delete e from F_i for all $i \geq \text{level}(e)$.

   Now we want to look for a replacement edge to reconnect u and v.

   - Note that the replacement edge cannot be at a level less than level(e) by Invariant 2 (recall that each F_i is a minimum spanning forest).
   - We will start searching for a replacement edge at level(e) to preserve the Invariant refinv:subset.

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

2. For $i = \text{level}(e)$ to $\log n$:

   (a) Let T_u be the tree containing u, and let T_v be the tree containing v. WLOG, assume $|T_v| \leq |T_u|$.

   (b) By Invariant 1, we know that $|T_u| + |T_v| \leq 2^i$, so $|T_v| \leq 2^{i-1}$. This means that we can afford to push all edges of T_v down to level $i - 1$.

   (c) For each edge (x, y) at level i with x in T_v:

      i. If y is in T_u, add (x, y) to $F_i, F_{i+1}, \ldots, F_{\log n}$, and stop.

      ii. Otherwise set level$(x, y) \leftarrow i - 1$.

In order for `Delete` to work correctly, we must augment our Euler-Tour trees. First, each Euler-Tour tree must keep track of its subtree sizes in order to find which of $|T_v|$ and $|T_u|$ is smaller in $O(1)$. (This augmentation is standard and easy.) We also need to augment our trees 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 augment the tree to keep track of whether a subtree rooted at some internal node contains any such nodes. With this augmentation we can find the next level-i edge incident to $x \in T_v$ in $O(\log n)$ time using successor and jumping over empty subtrees.

Lines 1 and 2(c)i cost $O(\log^2 n)$ total, while 2(c)ii in the loop costs $O(\log n \cdot \#$ of edges decremented) total. Note that an edge's level cannot be lower than 0, so an edge is decremented at most $O(\log n)$ times. Therefore, the amortized cost of `Delete` is $O(\log^2 n)$ as desired.

4 Other Dynamic Graph Problems

We also have results for other kinds of dynamic graph problems.

k-Connectivity

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 [8]. 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 [9]

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 $k = \text{poly}(\lg n)$? This problem is open for whole graph k-connectivity as well.

Minimum spanning forest — $O(\log^4 n)$ updates can be obtained [4], as well as an $O(\sqrt{n})$ worst-case update [6] for general graphs and a $O(\log n)$ update on planar graphs [2]. Bipartiteness, also known as 2-colorability, is reducible to minimum spanning forest.

Planarity testing — in which we ask if inserting some edge (u, v) into our graph violates planarity — can be done in general in $O(n^{2/3})$ [10]. For a fixed embedding, planarity testing can be done in $O(\log^2 n)$ [6]. La Poutré showed an $O(\alpha(m, n) \cdot m + n)$ algorithm for a total of m operations with an incremental graph in [12].

Directed graphs — we now want to know if vertex v is connected to vertex w in a directed graph. Notice that the problem is now no longer transitive. We assume that insertions and deletions are done in bulk, meaning we insert or delete a vertex and all its incident edges at the same time. The current results are:

- $O(n^2)$ amortized updates, O(1) worst-case query [21] [25]
- Sankowski et al. showed the same bound in worst case. This is optimal if we are explicitly storing the transitive closure matrix [24]
- **OPEN**: $o(n^2)$ worst-case for all updates?
- $O(m\sqrt{n}t)$ amortized bulk update, $O(\sqrt{n}/t)$ worst case query for any $t = O(\sqrt{n})$ [23]
- $O(m + n \lg n)$ amortized bulk update, $O(n)$ worst case query [22]
- **OPEN**: Full trade-off: number of updates times number of querries $O(mn)$ or $O(n^2)$?
- On an acyclic graph: $O(n^{1.575}t)$ update, $O(n^{0.575}/t)$ query, with once again $t = O(\sqrt{n})$ [21].

All-pairs shortest paths — weight of shortest $v \to w$ path for all pairs of vertices, still with undirected graphs. Results include:

- $O(n^2(\lg n + \lg^2(1 + m/n)))$ amortized bulk update, $O(1)$ worst case query [20] (improvised on [19])
- **OPEN**: $O(n^2)$ or $o(n^2)$ update, even with undirected graphs?
- $O(n^{2.75})$ worst case update, $O(1)$ query [18]
- For unweighted graphs, $O(m\sqrt{n}$ polylg $n)$ amortizes updates, $O(n^{3/4})$ worst case query [17]
- undirected, unweighted and $(1+\epsilon)$-approx: $O(\sqrt{m}nt)$ amortized update, $O(\sqrt{m}/t)$ worst case query, $t = O(\sqrt{n})$ [16]

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 2005: 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] Mikkel Thorup: Fully-dynamic min-cut. STOC 2001: 224-230

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

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

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

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

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

[14] Robert Tarjan: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22(2): 215-225 (1975)

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

[16] Liam Roditty, Uri Zwick: Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs. FOCS 2004: 499-508

[17] Liam Roditty, Uri Zwick: On Dynamic Shortest Paths Problems. ESA 2004: 580-591

[18] Mikkel Thorup: Worst-case update times for fully-dynamic all-pairs shortest paths. STOC 2005:112-119

[19] Camil Demetrescu, Giuseppe F. Italiano: A new approach to dynamic all pairs shortest paths. STOC 2003:159-166

[20] Mikkel Thorup: Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. SWAT 2004:384-396

[21] Camil Demetrescu, Giuseppe F. Italiano: Fully Dynamic Transitive Closure: Breaking Through the O(n2) Barrier. FOCS 2000:381-389

[22] Liam Roditty, Uri Zwick: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. STOC 2004:184-191

[23] Liam Roditty, Uri Zwick: Improved Dynamic Reachability Algorithms for Directed Graphs. FOCS 2002:679

[24] Piotr Sankowski: Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract). FOCS 2004:509-517

[25] Liam Roditty: A faster and simpler fully dynamic transitive closure. SODA 2003:404-412

6.851 Advanced Data Structures
Spring 2012