Dynamic Graph Algorithms Using Dynamic Forest Techniques

Lawrence Wu
llwu@mit.edu

December 10, 2014

Abstract

A dynamic graph problem is a problem of maintaining a graph subject to edge insertions and/or deletions, and queries about some property of the graph. An incremental dynamic graph algorithm supports just edge insertions, a decremental dynamic graph algorithm supports just edge deletions, and a fully dynamic graph algorithm supports both edge insertions and deletions. In this paper, we survey some of the currently best results in dynamic graph algorithms, first describing the dynamic forest data structures used, then describing the high-level techniques used and presenting an amortized analysis. Then, we discuss some open conjectures about whether the results described in this paper could be generalized.

1 Introduction

In section 2, we briefly describe the *Euler tour trees* (abbreviated *ET-trees*) of Henzinger and King (1995) [3] and the top trees of Alstrup et al (1997) [1], which will be used in later sections. In section 3, we present algorithms for the fully dynamic connectivity and decrementally dynamic minimum spanning tree problems (and also briefly describe algorithms for the fully dynamic minimum spanning tree and fully dynamic 2-connectivity problems) with polylogarithmic amortized performance due to Holm et al (2001) [5]. Finally, Section 4 concludes with a discussion of open problems in the area.

2 Dynamic Forest Representations

The following data structures are presented mainly as black boxes, with some intuition given about implementation. For more detailed treatments of the subject, we refer the reader to [3] for Euler tour trees and [1] for top trees.

2.1 Euler tour trees

In the directed representation of a tree where each undirected edge (v, w) of the tree is represented by the directed edges (v, w) and (w, v), a Euler tour of the tree is a path on the tree that visits each edge exactly once (so the Euler tour of a size n tree has length $2n - 2$). Here we describe the method of representing a forest by representing a *Euler tour* of each tree in the forest as a balanced binary search tree, with pointers from each vertex in the forest to its first appearance in the Euler tour. Using this method, the following operations can be supported in $O(\log n)$ time using $O(n)$ space:

- **connected(v,w)** : returns whether v and w are in the same tree.
- **link(v,w)** : if v and w are in different trees, inserts an edge with endpoints v and w connecting the trees.
- **cut(v,w)** : deletes the edge v and w, splitting the tree that (v, w) was in into two trees.
- **size(v)** : returns the number of vertices in the tree containing v.
- **min-key(v)** : if each vertex is assigned a numerical key, then this returns the minimal key among the vertices in the tree containing v.

Since linking two trees corresponds to concatenating their Euler tours and cutting a tree corresponds to splitting its Euler tour, **link** and **cut** can be implemented using the **merge** and **split** operations of the balanced binary tree. Since two vertices are in the same tree iff they are in the same Euler tour, **connected** can be implemented using the **findroot** operation of the balanced binary search tree. **size** and **min-key** can be implemented by augmenting each node of the balanced binary trees with the *size* and *min-key* of the subtree rooted at that node, so that our operations simply need to return the augmentation values at the root.

Finally, Henzinger and King (1999) [4] note that if we use a $\Theta(\log n)$-ary B-tree instead of a balanced binary tree, we can, at the cost of link and cut taking $O(\log^2 n / \log\log n)$ time, make the other operations run in $O(\log n / \log\log n)$ time, since the height of the B-tree is $O(\log n / \log\log n)$ and the branching factor is $O(\log n)$.

2.2 Top Trees

Node that each Euler tour tree augmentation stores information about a contiguous section of a Euler Tour, so the Euler tour tree representation method stores aggregate data about subtrees of trees in the forest. The *top trees* discussed in this section stores information about paths, which will turn out to be useful in the dynamic 2-connectivity problem. A top tree $\mathcal{T}$ is defined on an underlying tree T and a set ∂T of at most two vertices of T called *external boundary vertices*. The boundary vertices of any connected subtree C of T are

defined to be the vertices of C which are either boundary vertices of T or are connected to a vertex outside of C by an edge. C is a cluster if and only if it has at most 2 boundary vertices. The top tree is a binary tree representing the underlying tree's subdivision into clusters, such that:

- The nodes of $\mathcal{T}$ are the clusters of $(T, \partial T)$.
- The leaves of $\mathcal{T}$ are the edges of T.
- Sibling nodes of $\mathcal{T}$ are clusters intersecting in a single vertex, and their parent node is the union of the clusters.
- The root of $\mathcal{T}$ is T.

For each cluster, we associate a path $\pi(C)$, which is the path between the boundary vertices. So at each node C we can store any information about $\pi(C)$ as long as it can be maintained in $O(1)$ time under the following $O(1)$ time core operations (we will see that these are the only operations that we need to change the tree with):

- **Merge(A, B)** : Merges the top trees rooted at clusters A and B into a top tree rooted at the cluster $A \cup B$, then returns the new root cluster.
- **Split(C)** : Deletes the root cluster C, turning $\mathcal{T}$ into two top trees.

An example of an augmentation that we can maintain is the maximal weight edge in $\pi(C)$, since we need to do no maintenance work for **Split(C)**, and for **Merge(A, B)** we need only augment the new root cluster with the greater of the augmentations of A and B.

Alstrup et al (1997) [1] and Frederickson (1985) [2] proved that for a dynamic forest we can maintain top trees of height $O(\log n)$ and total size linear in the size of the dynamic forest supporting the link, cut, and expose (defined below) operations in with a sequence of $O(\log n)$ Merges and Splits, and that the sequence itself can be identified in $O(\log n)$ time. The expose operation is defined as follows:

- **Expose(v,w)** : If v and w are in the same tree, makes v and w external boundary vertices of the top tree containing them and returns the new root cluster, else returns **nil**.

It follows from this theorem that we can find the root cluster containing a given vertex v, and therefore implement $Connected(v, w)$ in $O(\log n)$ time by maintaining a pointer from each vertex in the dynamic forest to the smallest cluster that it appears in. [5] It also follows that we can query the maximal weight edge in the path between two vertices v and w by calling **Expose(v,w)**, then returning the augmentation value mentioned earlier from the root cluster containing v and w.

3 The Algorithms of Holm et al.

3.1 Fully Dynamic Graph Connectivity

Holm et al's algorithm maintains a spanning forest F of the graph G. As mentioned by Kapron, King, and Mountjoy in their paper [6], this method is used in most dynamic connectivity algorithms, as edge insertions are easy to handle, while edge deletions become a problem of finding a replacement edge to reconnect the trees that were split. Holm et al's algorithm amortizes this work by assigning a level $\ell(e) \in \{0, 1, \cdots, \lfloor \log_2 n \rfloor\}$ to each edge e. Then G_i is defined to be the subgraph of G containing all edges of level at least i. We maintain the forests $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_{\lfloor \log_2 n \rfloor}$ such such that F_i is a spanning forest of G_i for each i. This takes $O(n \log n)$ space since there are $O(\log n)$ and we store a size $O(n)$ forest for each level. We also maintain the adjacency lists for each vertex, which takes $O(n + m)$ space since each edge appears in two adjacency lists. So the space usage of the algorithm is $O(m+n \log n)$. The algorithm relies on maintaining the following invariants:

1. F_i is a maximum spanning forest of G_i for each i, that is, for each edge (v, w), v and w are connected in $F_{\ell(v,w)}$
2. The maximal number of vertices in a tree in F_i is $\lfloor n/2^i \rfloor$.

The algorithm will never decrease the level of an edge, so that each edge increases in level at most $\lfloor \log_2 n \rfloor$ times, so that we can charge the costs of any operations which increase the level of an edge to the insertion of that edge. Now the insert, connected, and delete operations are implemented as follows:

- **insert(v,w)** : Set $\ell(v, w) = 0$. Add v and w to each other's adjacency lists (cost $O(\log n)$). Call **link(v,w)** on F_0. Invariant 1 is preserved by **link(v,w)**, and invariant 2 is preserved since we only changed F_0, which can have up to n vertices. The actual cost of this operation is $O(\log^2 n / \log \log n)$.
- **connected(v,w)** : Simply call **connected(v,w)** on F_0. The cost of this operation is $O(\log n / \log \log n)$.
- **delete(v,w)** : If v and w are not connected, simply remove v and w from each other's adjancency lists (cost $O(\log n)$). Otherwise, call **cut**(v,w) from each level $\leq \ell(v, w)$, which has total actual cost $O(\log^3 n / \log \log n)$ and then search for a replacement edge. To find a replacement edge, we do, for level $i = \ell(v, w)$ down to $i = 0$:

  Let T_v be the tree containing v and T_w be the tree containing w. Assume, without loss of generality, that $|T_v| \leq |T_w|$, since we are able to query the size of a tree using Euler tour tree representation. Then by Invariant 2, $|T_v| + |T_w| \leq \lfloor n/2^i \rfloor$, so $|T_v| \leq \lfloor n/2^{i+1} \rfloor$, so it would not violate Invariant 2 if the edges of T_v had their levels increased to $i + 1$. Thus, the following procedure does not violate Invariant 2: for each y incident to a vertex

$x \in T_v$ with $\ell(x,y) = i$ (we can augment the nodes of our Euler trees with the incident edges at the level of the tree using $O(m)$ total extra space, which does not change the $O(m+n\log n)$ space bound, so that each y can be considered in $O(\log n)$ time), if y is in T_u, add (x,y) to $F_0, \cdots, F_{\lfloor \log_2 n \rfloor}$ (total actual cost $O(\log^3 n/\log\log n)$) and return, otherwise increase the level of (x,y) to $i+1$, and charge the $O(\log n)$ cost for considering the edge to its insertion.

Note that since we look for replacement edges in decreasing order of level, Invariant 1 is maintained by **delete**. Since the level of an edges increases $O(\log n)$ times, we charge a total of $O(\log^2 n)$ deletion cost to insertion, so insertion has an amortized cost of $O(\log^2 n)$ while deletion has an amortized cost of $O(\log^3 n/\log\log n)$. The paper by Holm et al gives a $O(\log^2 n)$ time bound, although at the time of writing we were not able to verify it, since it relies on $O(\log n)$ time links/cuts (see page 732 of Holm et al [5]) which are not guaranteed by $\Theta(\log n)$-ary B-tree-based Euler trees. As stated before, the cost of a query is $O(\log n/\log\log n)$, so all operations run in amortized polylogarithmic time.

3.2 Decrementally Dynamic Minimum Spanning Tree

In the previous subsection, we already described how to maintain a spanning forest of G. So to maintain a *minimum* spanning forest of G, we will see that it suffices use the same deletion algorithm as above, but considering the potential replacement edges in each level in increasing order of weight (this takes no extra time using Euler tour tree augmentation). To show that we maintain a minimum spanning forest this way, we show that the following invariant is maintained:

1. Every cycle C has a nontree lowest heaviest edge, that is, an edge e with $w(e) = \max_{f\in C} w(f)$ and $\ell(e) = \min_{f\in C} \ell(f)$.

Lemma 1. *[5] Assume invariant 1 and that F is a minimum spanning forest. Then, for any tree edge e, among all replacement edges, the lightest edge is on the maximum level.*

Proof. For any two replacement edges e_1 and e_2 for e with $w(e_1) < w(e_2)$, let C_1 be the cycle induced by e and e_1, C_2 be the cycle induced by e and e_2, and $C = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$. Then since F is a minimum spanning forest, e_1 is the heaviest edge on C_1 and e_2 is the heaviest edge on C_2, but e_2 is heavier than e_1, so e_2 is the heaviest edge on C, so by invariant 1, e_2 is the lowest edge on C, so e_1 is on a higher level. □

Now to show that Invariant 1 is maintained, we show that it is maintained during the search for a replacement edge, since the deletion of the edge creates no new cycles. Suppose an edge e gets its level increased or becomes a tree edge. If e is not a unique nontree lowest heaviest edge on any cycle C, then Invariant 1 cannot be violated. Otherwise, let i be the level of e. By the definition of our replacement procedure, every edge in T_v and every edge incident to T_v

lighter than e has level $> i$. Since e is lowest on C and it is a unique nontree lowest heaviest edge on C, any nontree edge in C heavier or equal to e has level > 1. So all edges in C incident to T_v have level $> i$. Assume for the sake of contradiction that C leaves T_v. Then it must have some edge $f \neq e$ leaving T_v, but if $f \geq i$ then it would be a replacement edge, and since we are searching for replacement edges in decreasing order, we know there are no replacement edges with level $> i$, so f has level at most i, which contradicts the fact that all edges in C incident to T_v have level $> i$. Thus, C does not leave T_v, so e is not a replacement edge, and since all other edges in C have level $> i$, Invariant 1 is maintained when the level of e is increased.

Now, because Invariant 1 is maintained and due to Lemma 1, whenever a tree edge is deleted, it is replaced by a lightest replacement edge, so the spanning forest remains minimal. We have the same time bounds as before, with each initial edge contributing a potential cost of $O(\log^2 n)$, so that any sequence of deletions takes $O(m \log^2 n)$ time.

3.3 Fully Dynamic MSF and Fully Dynamic 2-connectivity

For the fully dynamic minimum spanning forest problem, Holm et al [5] give an algorithm with $O(\log^4 n)$ amortized update times based on reducing the fully dynamic minimum spanning forest problem to a series of decrementally dynamic minimum spanning forest problems. The idea is to maintain $\lceil \log_2 m \rceil$ decremental data structures. Then to insert new edges or replacement edges we need to update this set of decremental data structures. Implementing these updates involves a careful application of top trees. The full description and analysis is rather detailed, so refer the interested reader to Holm et al [5].

For the fully dynamic 2-connectivity problem, Holm et al [5] present an algorithm with $O(\log^4 n)$ update times based on their method for fully dynamic connectivity. In this problem, we must consider not only a spanning forest but also *covering edges*. A nontree edge (v, w) is a *covering edge* for a tree edge e if e is on the cycle induced by (v, w). Maintaining cover information on paths can be done with top trees; a rigorous treatment of the method is given in [5].

4 Conclusions

We surveyed some dynamic graph algorithms that should have good performance in practice; however, there are still questions of theoretical interest. It remains unknown whether there is an deterministic dynamic graph connectivity algorithm which has *worst-case* polylogarithmic runtime for updates and queries. The algorithm due to Holm et al does not satisfy this since the time bounds are amortized. There is also a new Monte Carlo algorithm due to Kapron, King, and Mountjoy [6] which has polylogarithmic worst-case time bounds but is not deterministic. The best deterministic worst-case result so far is still $O(\sqrt{n})$ update and $O(1)$ query due to Frederickson (1985) [2]. It would also constitute interesting work to find polylogarithmic (amortized time)

algorithms for k-connectivity when $k = O(1)$ (this problem is currently open for $k > 2$). It seems that polylogarithmic algorithms based on the algorithms of Holm et al should exist since two vertices are k-connected iff they are $(k-1)$-connected with any edge of the graph removed; however, at the time of writing, we have been unsuccessful in solving this problem.

References

[1] Alstrup, S., Holm, J., de Lichtenburg, K., and Thorup, M. 1997. Minimizing diameters of dynamic trees. In *Proceedings of the 24th International Colloquium on Automata Languages and Programming (ICALP)*. Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, New York, pp. 270 280.

[2] Frederickson , G. N. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.* 14, 4, 781798. (Announced at STOC, 1983.)

[3] Henzinger, M. R., and King, V. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 27th annual ACM Symposium on Theory of Computing (STOC)*. ACM, 519-527.

[4] Henzinger, M. R., and King, V. 1999. Randomized fully dynamic graph algorithms with polylogarith- mic time per operation. *J. ACM 46*, 4 (July), 502536. (Announced at STOC, 1995.)

[5] Holm, J., Alstrup, S., de Lichtenburg, K., and Thorup, M. 2001. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Journal of the ACM (JACM)*, 48 (4). 723-760.

[6] Kapron, B., King, V., and Mountjoy, B. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 1131-1142.