Dynamic shortest paths algorithms

Namrata Anand

May 31, 2016

1 Introduction

Dynamic graph algorithms, like their static counterparts, must efficiently compute functions on graphs; unlike static graph algorithms, however, dynamic graph algorithms must maintain this efficiency even as the underlying graph structure changes. Dynamic graph algorithms refer to the class of algorithms which support updates on the graph structure, such as edge insertions, edge deletions, and edge weight changes. *Fully dynamic* algorithms support interleaved insertions and deletions. *Incremental* algorithms support insertions and not deletions, and *decremental* algorithms support deletions and not insertions.

Dynamic graph algorithms must respond to queries on the graph as it changes. Consider a graph $G = (V, E)$ with m edges and n nodes, where V is the vertex set and E is the edge set. Some example query operations to consider are– determining connectivity of nodes, finding the exact or approximate distance between nodes, determining the size of the maximum matching in the graph, and constructing the minimum or maximum spanning tree of the graph.

Efficient algorithms have been built for the dynamic connectivity problem. Naively, we can compute the connected components of the graph after each update; this gives us an $O(m)$ update and $O(1)$ query time. The partially dynamic connectivity problem for undirected graphs can be solved with $O(\alpha(n))$ update and query time using the union-find data structure; here, $\alpha(n)$ is the Ackermann inverse function. The fully dynamic problem is harder, but can still be solved with $O(\log(n))$ updates and queries [1].
In comparison, algorithms for dynamic shortest paths problems have much less impressive run-times. For incremental/decremental SSSP (single-source shortest paths), a classic paper by Even and Shiloach [2] gives an amortized update time of $O(n)$ (total update time of $O(mn)$ over all deletions) and a query time of $O(1)$ for undirected, unweighted graphs. This algorithm was the fastest algorithm known for this problem for three decades after its publication, and it is still the fastest approach for exact SSSP in a partially dynamic graph. We will discuss theoretical results presented in [3], which give some intuition as to why this classic result is difficult to improve. The underlying data structure of the algorithm is a tree structure, now called an "Even-Shiloach Tree." In this paper, we implement the ES-tree algorithm for decremental SSSP and compare its performance with a naive approach based on the breadth-first search (BFS) algorithm.

Until 1999, algorithms for the fully dynamic APSP problem (all pairs shortest paths) were no better than recomputing static APSP after each update in the worst case. King gave an algorithm for directed, positive integer-weighted graphs with $O(n^{2.5}(C\log(n)^{0.5})$ update and $O(1)$ query time [4]. More recently, in 2004 two algorithms were built which improve efficiency for fully dynamic APSP. Demetrescu and Italiano [5] gave an algorithm, improved by Thorup [6], with amortized update time $\tilde{O}(n^2)$ and query time $O(1)$. Roditty and Zwick [2] presented an algorithm with amortized update time $O(m\sqrt{n})$ and worst-case query time $O(n^{3/4})$.

For dynamic graph algorithms, there is an inherent trade-off between fast query times and fast update times. For the fully dynamic APSP problem, there are two extreme cases to consider. To ensure $O(1)$ queries, you might rebuild the entire distance matrix for nodes in the graph after each update, giving an $\tilde{O}(n^3)$ update time. To ensure $O(1)$ updates, you might simply compute the single-source shortest path from a node $x \in V$ to return the queried distance between $x, y \in V$, giving an $O(n^2)$ query time. We see this trade-off in the two fully dynamic APSP algorithms given above. Demetrescu-Italiano's algorithm has fast

query times, but slow update times. Roditty-Zwick's algorithm has terribly slow query times but, for sparse graphs ($m < n^{3/2}$), has a faster update time.

In this paper, we discuss approaches for dynamic exact shortest paths problems on undirected, unweighted graphs. We first look at Even and Shiloach's algorithm, applied to the decremental SSSP problem. The rest of the paper will be spent on understanding how to extend the ES-tree algorithm to make headway on dynamic *APSP* problems. We discuss some theoretical results for fully dynamic APSP presented in [3]. Finally, we consider how we might utilize E-S trees to build a "quick and dirty" fully dynamic APSP randomized algorithm inspired by Roditty-Zwick.

2 Decremental SSSP

2.1 Even-Shiloach Trees

The naive approach for partially dynamic SSSP would be to run the static SSSP algorithm after each update to the graph. Using Dijkstra's algorithm, this would give an $O(m \log(n))$ update time (using, say, a binary search tree as a priority queue) or an $O(m + n \log(n))$ update time, using a Fibonacci heap as the underlying priority queue. Thorup [1] improved the static SSSP run-time to $O(m+n)$, so the naive approach could have a total update time of $O(m^2 + mn)$ over all deletions. In this paper, however, we are exploring dynamic path algorithms on unweighted graphs; the naive approach of running BFS after each update would give a total update time of $O(m(m + n))$ over all deletions.

The classic approach for decremental SSSP is a variation of the original algorithm from Even-Shiloach, who presented their algorithm in the context of solving the decremental connectivity problem for undirected, unweighted graphs. The ES-tree gives total update time of $O(mn)$ over all deletions.The algorithm pseudocode is presented in Algorithm 1 and adapted from [7]. Given a graph $G = (V, E)$ and a source node s, we compute the ES-tree as simply a BFS tree from the source node s and assign each node v in the graph a level $l(v) = d(s, v)$ corresponding to its initial distance from source node s. Any node u not in the same connected component as s is assigned level $l(u) = \infty$. The intuition here is that each edge deletion might alter the levels of nodes in the ES-tree.

Initialization. Compute the BFS tree E from source node s. For each node v, compute $l(v)$ and $N(v)$, the set of all neighbors of v in G and their levels.

Update. Suppose we must delete edge (u, v). Since the nodes are neighbors, they must be within one level of each other. If u and v share the same level in E ($l(u) = l(v)$), then removing the edge (u, v) does not change any shortest-path distances in the graph from s. We only need to update the data structure to let it know that u and v are no longer neighbors; we delete key-value pair $(l(u), u)$ from $N(v)$ and $(l(v), v)$ from $N(u)$. Suppose $l(u) = l(v) - 1$. Again, we delete each node from the other node's neighbor set. The distance to the "higher" node $d(s, u)$ does not change; however, now if v has no neighbors with level $l(v) - 1$, it has to "drop" down in the ES-tree. In addition, if v drops down, all neighbors of v ($\{x \in V | v \in N(x)\}$) might have to drop as well. This recursive update procedure determines the update cost of the algorithm.

Query. Answering a query for the distance $d(s, v)$ for some $v \in V$ is simple; the algorithm simply returns the corresponding level $l(v)$ of v in the ES-tree E.

Correctness. The correctness argument relies on the correctness argument for BFS, which we will take as known. The focus of our correctness argument is on the UpdateLevels step.

- Suppose initially $d(s, i) = l(i)$ and $d(s, j) = l(j)$. If edge (i, j) is deleted and $l(i) = l(j)$, the algorithm is correct since we do not update $l(i)$ or $l(j)$ in this case. The distance estimates remain the same.

Algorithm 1: Decremental SSSP with ES-tree

// Inputs

- g: pointer to underlying Graph object $G = (V, E)$ with m edges and n nodes
- s: source node in G

// Data structures;

- Q: global set storing nodes whose levels might need to be changed
- N: pointers to an array of neighbor sets $N(i)$ for each node i in the graph
- E: the ES-tree, an initially empty Graph which is built via BFS during the Initialize phase of the algorithm
- l: integer array of length n where $l[i]$ gives level $l(i)$ in g

Initialize
- Run BFS from s in G up to depth m to compute $d(s, i)$ $\forall u \in V$
- **for** $i \in V$ **do**
  - $l(i) = d(s, i)$
  - **for** $(i, j) \in E$ **do**
    - Insert $(l(j), j)$ into $N(i)$

Query(i)
- Return $l(i)$

Delete(i, j)
- //delete edge in underlying graph
- **if** $g \rightarrow$ *hasEdge*(i, j) **then**
  - //if edge exists
  - $g \rightarrow$deleteEdge(i, j)
  - **if** $l(i) \neq \infty$ **then**
    - //check that we are in the source's component, else do not update
    - Insert $(l(i), i)$ and $(l(j), j)$ into Q
    - //to instead increase the weight of edge (i,j) to w(i,j), insert $(l(i) + w(i, j), i)$ and $(l(j) + w(i, j), j)$
    - Delete $(l(i), i)$ from $N(j)$
    - Delete $(l(j), j)$ from $N(i)$
    - UpdateLevels

UpdateLevels
- // drop nodes in Q until Q is empty
- **while** Q *is not empty* **do**
  - Select node y with the minimum key value $l(y)$ and remove from Q
  - **if** $N(y)$ *is empty* **then**
    - //node is disjoint
    - $l'(y) = \infty$
  - **else**
    - /node either stays in place or drops down a level
    - $l'(y) = \min_{z \in N(y)}(l(z)) + 1$ // for the wtd case, the update is $\min_{z \in N(y)}(l(z) + w(y, z))$
  - **if** $l'(y) > l(y)$ **then**
    - //if node drops, update level
    - $l(y) = l'(y)$
    - **for** $x \in N(y)$ **do**
      - //iterate through neighboring nodes
      - Update entry $(l(y), y)$ in $N(x)$
      - // for the wtd case, the update is $(l(y) + w(x, y), y)$
      - **if** x *is not in* Q *and* x *is in the same component as* s **then**
        - //drop x unless $l(x) = \infty$
        - Insert $(l(x), x)$ into Q

- Suppose initially $d(s,i) = l(i)$. If edge (i,j) is deleted, and $l(i) < l(j)$, the algorithm estimate $d(s,i) = l(i)$ is correct since we do not update $l(i)$ in this case.
- Suppose initially $d(s,j) = l(j)$. If edge (i,j) is deleted, and $l(i) < l(j)$, we drop j to level $\min_{z \in N(j)}(l(j))+1$. Now $d(s,i) = d(s, \text{argmin}_{z \in N(j)}(l(j))) + 1$. This is the new shortest distance to j from s. The only paths to j from s are through the neighbors of j. Therefore, the shortest path to j must be the shortest path to the neighbor set plus the distance from j to any of its neighbors.
- In the case when deletion of (i,j) separates i and j into two components, the level values of every node in the component not including s is set to ∞.

Run-time analysis.

- Initialize: BFS runs in $O(m+n)$. Each neighbor set $N(u)$ can be implemented as a heap or binary search tree over up to n keys (faster implementation described later). Each $N(u)$ can have at most n keys. Insertion into $N(u)$ takes $O(\log(n))$ time.
- Query(i): Query times for the ES-tree are $O(1)$, since the level of each node is stored in a lookup table.
- Delete (i,j): Checking the graph adjacency list to see if the edge exists takes at most $O(m)$ time. Deleting the edge from the adjacency list takes $O(1)$ time. At the start of each update, Q is empty, so two insertions into Q takes $O(1)$ time. Deleting keys from neighbor sets takes $O(\log(n))$ time.
- UpdateLevels: If no nodes are dropped after a call to UpdateLevels, the run-time of the function is $O(\log(n))$, the time it takes to look up $l'(y)$ for the two nodes in Q. If one of the nodes, say j, is dropped, there might be a cascade wherein the neighbors of j drop, as do their neighbors, etc. Therefore the worst case cost of deleting an edge is $O(\sum_{j \in V} degree(j) \log(n)) = O(m \log(n))$.

We see that the largest run-time factor of $O(m \log(n))$ comes from UpdateLevels. We can get an amortized update cost by the aggregate method. Over a sequence of deletions, for any node $v \in V$, $l(v)$ can increase no more than n times. The total cost is then $O(mn \log(n))$. This gives an amortized cost per update of $O(n \log(n))$ over a sequence of m deletions.

Space usage. The ES-tree takes up $O(m+n)$ space total. The graph is reprsented as an array of linked lists and takes $O(m+n)$ space. The neighboring sets, represented as binary search trees, in total take $O(m)$ space. The array encoding node levels and the binary search tree Q for nodes whose levels need to be updated take $O(n)$ space.

A faster implementation. The run-time of the ES-tree algorithm implemented over binary search trees is $O(mn \log(n))$ for the case when $n << m$. The naive algorithm runs BFS over a binary search tree after each deletion. The run-time is $O(m^2 + mn)$. We see that the ES-tree algorithm as implemented only outperforms the naive algorithm when $m >> n \log(n)$. Running ES-tree over sparse graphs would give us a sizeable slowdown relative to the naive algorithm. Can we implement the ES-tree algorithm for unweighted, undirected graphs to get the $O(mn)$ total update cost presented in Even and Shiloach, while maintaining the same space usage as our current implementation?

We can—the trick is to eliminate the "extract-min" step relying on the priority queue in UpdateLevels. This allows us to represent the neighbor sets as unordered sets rather than priority queues, eliminating the log factor in the total update cost. The pseudocode and analysis are given in the appendix. To differentiate this implementation from the one given above, we call this algorithm "Fast ES-tree." The algorithm has total update cost $O(mn)$ and query time $O(1)$

Extensions to ES-tree.

Incremental SSSP. The ES-tree algorithm described above can be altered to handle up to m insertions instead of deletions. If we begin with an empty graph, for every $v \in V$, $v \neq s$, $l(v) = \infty$. On each insertion

(u, v), we add key-value pair $(l(u), u)$ to $N(v)$ and $(l(v), v)$ to $N(u)$. If u, v share the same level, their distances to the source node do not change after the edge insertion. If their levels differ, instead of a recursive "drop" procedure, we will have a recursive "rise" procedure, where an added edge might cause a node and its neighbors to decrease their level in the graph (nearing the source node).

Weighted graphs. Algorithm 1 will work for weighted graphs, with few modifications. Each delete can be thought of as an edge-weight increase to infinity. The delete function can essentially be replaced with an increase-weight function; more detail is included in the pseudocode.

Implementation. We implemented the ES-tree and Fast ES-tree algorithms, as well as a naive approach for comparison, in C++. For the naive approach, we compute all shortest paths from the source after each update using BFS over a binary search tree (the set container in C++ STL). This gives a total worst case update time of $O(m^2 \log(n))$ over all deletions and a space usage of $O(m + n)$.

All the algorithms were built over a base random Graph class. The Graph class builds an undirected, unweighted graph of m edges and n nodes by first building a random MST over the n nodes and then adding in the remaining $m - n$ edges uniformly at random. The graph is stored as an adjacency list where each node stores a pointer to a linked list of neighboring nodes. The class supports (random) edge insertions and deletions, can store a mapping from each vertex to the component it belongs to, and keeps a list of the total edge set as well.

The ES-tree implementation is also built over a binary search tree, adding an additional $\log(n)$ factor relative to the run-time published in [2]. In addition to building and storing the ES-tree as a Graph, we store the neighbor sets $N(u)$ for each $u \in V$ in binary search trees and keep a binary search tree Q as a global priority queue to house nodes whose levels might need to be dropped. We also store an integer array mapping each node to its current level in E. Nodes disjoint from the source node s are assigned level $m + 1$, an impossible distance in the graph, rather than infinity. The data structures for the Fast ES-tree implementation are the same, except we store the neighbor sets $N_1(u), N_2(u), N_3(u)$ for each $u \in V$ as hash tables (STL unordered sets) and represent Q as a FIFO queue (STL queue).

Finally, we built test/timing scripts for the algorithms, allowing us to assess the total time and time per update for each algorithm over a series of random edge deletions. The code is available at `https://github.com/nanand2/decrementalSSSP`.

2.2 Experiments

We ran experiments on the ES-tree and naive decremental SSSP algorithms to see how well the theoretically predicted update costs aligns with the actual implementation.

For the naive algorithm, the expected total run-time over m deletions is $O(m(m + n)) = O(m^2 + mn)$. However, in practice we expect to see an $O(mn)$ update cost. If $m \sim n$, then an $O(mn)$ update cost follows naturally. If on the other hand $m \sim n^2$, then we can expect BFS on a dense, nearly complete graph to reach every node within a few iterations, before it traverses each edge, and again we expect an $O(mn)$ cost. We do see an exactly linear relationship between total update cost over m deletions and m for constant n (Figure 1). The interesting thing here, is this shows us in practice, the naive approach for decremental SSSP scales with graph density exactly as our Fast ES-tree algorithm is predicted to scale. We expect, however, that the Fast ES-tree will have a lower update cost by a large constant factor.

For the ES-tree algorithm implemented with binary search trees, we have a predicted total update cost of $O(mn \log(n))$. In Figure 2, we show results for updates on a graph with $n = 100$ and m varied. For each value of m, we build a new random graph and delete n random edges. We expect to see a linear relationship between total update cost and m. However, we find that very sparse graphs do not obey the expected trend; in fact, there are very high total update costs for sparse graphs; this cost decreases linearly until a critical point (around edge density $\sim$ 20%), and then the expected positive linear trend is seen as edge density increases. That this trend is consistent across experiments with different values for n and different starting values for m seems to imply it is dependent on both m and n and is a function of the graph sparsity. While it might be an artifact of the implementation (for example, an issue with STL's set structure), it is clear

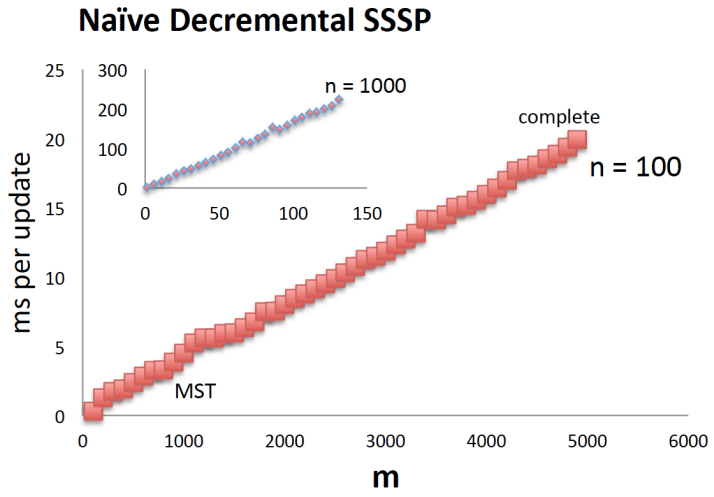

Figure 1: Experiment results for the naive approach for decremental SSSP (iterative BFS) on random, unweighted graph: Graph shows update cost in milliseconds per update over 100 deletions for a random graph with $n = 100$ as m varies from 100 to 4950; results averaged over 3 trials. As the graph varies from a minimum spanning tree to a complete graph, a stable linear trend is seen. The same trend is seen for $n = 1000$ nodes (inset) as m varies from 1000 to 150K over 1000 deletions. Note the order of magnitude difference in ms per update as n increases, as well as the tremendous difference in update cost between the ES-tree and the naive approach, even for small graphs.

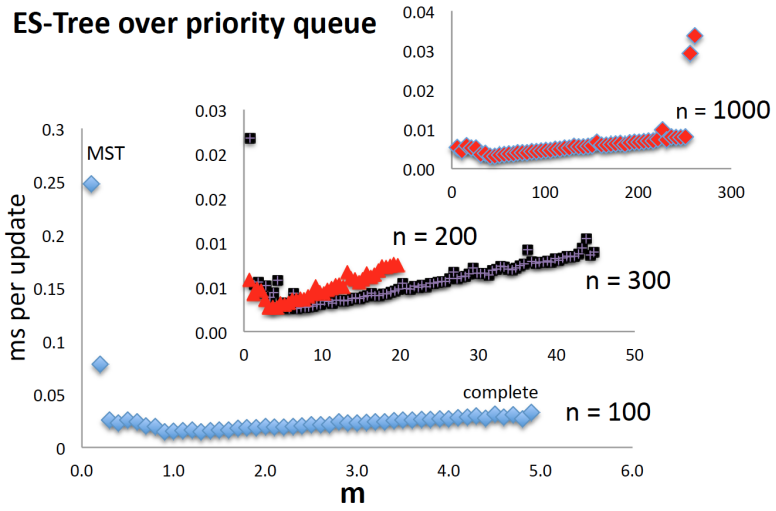

Figure 2: Experiment results for the ES-Tree implemented over a binary search tree as priority queue: Graph shows update cost in milliseconds per update over 100 deletions for a random graph with $n = 100$ as m varies from 100 to 4950; results averaged over 5 trials. As the graph varies from a minimum spanning tree to a complete graph, the update cost first drops and then rises linearly. The same trend is seen as we increase the number of nodes in the graph n (keeping the number of deletions equal to n). The relative magnitude of update cost remains stable even as the number of nodes increase.

from looking at the update cost that for sparse graphs, the ES-tree algorithm implemented over a priority queue is still preferable to the naive approach.

For the Fast ES-tree algorithm, the theoretically predicted total update cost over a series of edge deletions

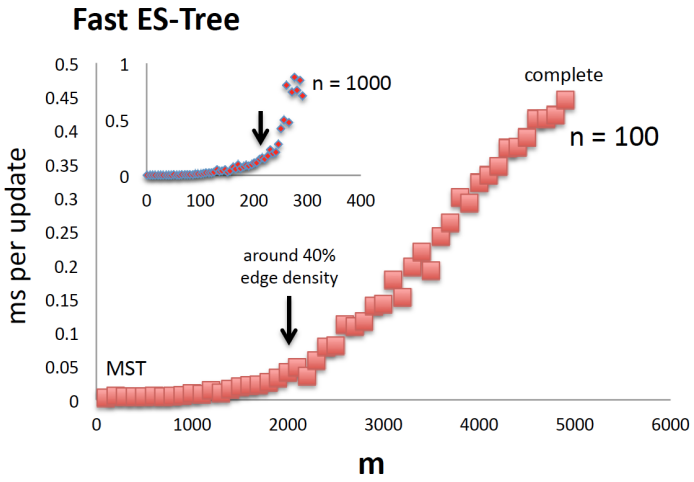

Figure 3: Experiment results for the Fast ES-tree. Graph shows update cost in milliseconds per update over 100 deletions for a graph with $n = 100$ as m varies from 100 to 4950. As the graph varies from a minimum spanning tree to a complete graph, around 40% edge density there is a shift to a steep linear trend which continues as graph density increases. The same trend is seen for $n = 1000$ nodes (inset) as m varies from 1000 to 300K, with a shift around 40% edge density (x-axis in 1000s of edges).

is $O(mn)$. In Figure 3, we show results for updates on a graph with $n = 100$ and m varied. For each value of m, we build a new random graph and delete n random edges; we average these results over 5 trials. We expect to see a linear relationship between total update cost and number of edges m, and the experiment aligns well with theory. For low values of m (where the graph is sparser), we see a slow, almost quadratic rise, which reaches a critical point (edge density $\sim 40\%$) and then steepens into the expected linear relationship. When the graph is denser, there is a higher likelihood of an edge deletion causing a cascade of level "drops"– we expect to see sparser graphs have a lower total cost and to see denser graphs near the $O(mn)$ bound. Here the clarity of the transition between total update cost for sparse versus dense graphs is really nice. We see a similar relationship when we increase n to 1000 nodes (Figure 3, inset).

Each of these algorithms, as expected, show a roughly linear increase in update operations cost as m increases. The relative update cost is lowest for the ES-Tree, comparably low for the Fast ES-Tree, and around 2 orders of magnitude larger for the naive approach across experiments.

2.3 How hard are partially dynamic SSSP problems?

The ES-Tree is a powerful algorithm, and since 1981 has stood its ground as the fastest deterministic algorithm for partially dynamic SSSP. It remains the fastest deterministic algorithm for this problem with no additive or multiplicative error [7]. In thinking about new approaches to partially or fully dynamic SSSP, it is helpful to consider *why* it has not been easy to improve Even and Shiloach's algorithm. Roditty and Zwick [3] offer some intuition via a reduction from difficult static problems to partially dynamic SSSP problem.

They begin by showing a clever reduction from static APSP to the weighted SSSP problem. Since our focus for this paper is on unweighted graphs, we will pass over this, with the note that their reduction shows that any improvement of the ES-tree algorithm will give improved run-times for the static APSP problem.

They next give a reduction from Boolean matrix multiplication to partially dynamic SSSP problems on unweighted graphs, reproduced here.

Theorem 2.1. *Let $\mathcal{A}$ be a decremental algorithm for the unweighted, undirected SSSP problem and let $init_{\mathcal{A}}(m,n)$, $update_{\mathcal{A}}(m,n)$, and $query_{\mathcal{A}}(m,n)$ be the initialization time, amortized update time, and amortized query time of A on a graph with m nodes and n edges. Then, there is an algorithm that multiplies two*

Boolean $n \times n$ matrices with a total number of m 1s in $O(init_{\mathcal{A}}(m+2n, 4n) + n * update_{\mathcal{A}}(m+2n, 4n) + n^2 textquery_{\mathcal{A}}(m+2n, 4n))$ time.

Proof. Recall that the Boolean product of $m \times k$ binary matrix A and $k \times n$ binary matrix B is given by the $m \times n$ matrix C where

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \ldots (a_{ik} \wedge b_{kj}).$$

The Boolean product is exactly analogous to matrix multiplication; the usual sum and product operations are replaced by the logical OR and logical AND operations.

Given A, B, $n \times n$ binary matrices, let $C = AB$ be their Boolean product. Construct a graph $G = (V, E)$ as follows:

- Let there be $4n$ vertices on the graph and 4 disjoint subsets of size n called s, u, v, and w. Then $V = \{s_i, u_i, v_i, w_i | 1 \leq i \leq n\}$.
- Construct edges between vertices (s_i, s_{i+1}) and (s_i, u_i) for $i = 1, \ldots, n$.
- If $a_{ij} = 1$, place an edge between u_i and v_j
- If $b_{ij} = 1$, place an edge between v_i and w_j

There are $m + 2n - 1$ edges. Query the distance between s_1 and w_j for $j = 1, \ldots, n$. The distance estimate gives us information about the Boolean matrix multiplication— $d(s_1, w_j) = 3$ iff $c1j = 1$. That is, if there is a path to w_j from u_1 through at least one v_i, then $d(s_1, w_j) = 3$ and it follows that $c_{1j} = \cdots \vee (a_{1i} \wedge b_{ij}) \vee \cdots = 1$ (See Figure 4, left panel).

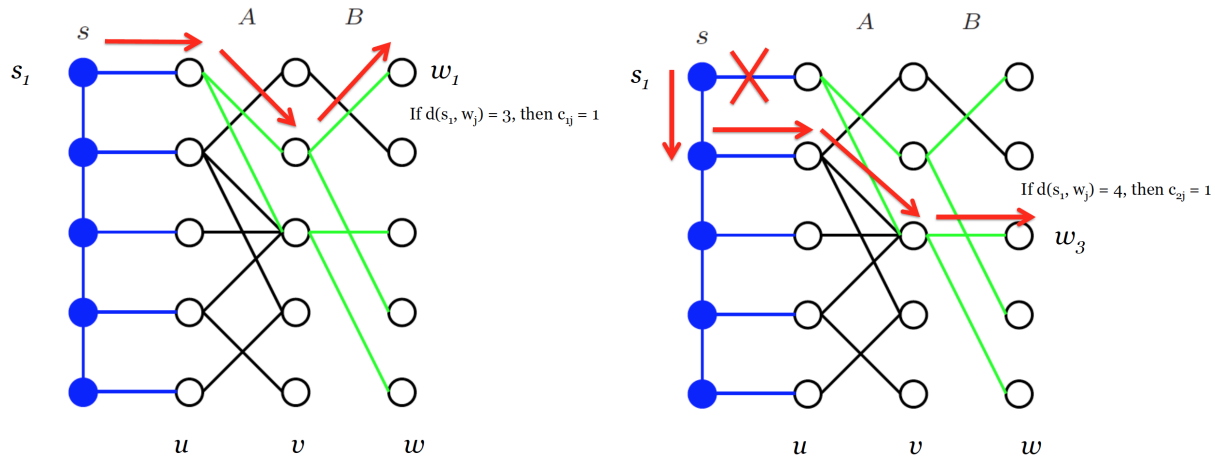

Figure 4: Figure adapted from [3]. Reduction from Boolean matrix multiplication to unweighted partially dynamic SSSP problem.

Delete (s_1, u_1). Now $d(s_1, w_j) = 4$ iff $c_{2j} = 1$ (Figure 4, right panel). Delete (s_2, u_2). Now $d(s_1, w_j) = 5$ iff $c_{3j} = 1$. Continue until all (s_i, u_i) are deleted. In the process, we update the decremental SSSP structure n times and query the algorithm n^2 times. In the process, we compute the Boolean product of matrices A and B, represented in the graph.

□

This reduction shows that partially dynamic SSSP is at least as hard as combinatorial Boolean matrix multiplication. Via the reduction, the ES-tree algorithm gives an $O(mn + n^2)$ time algorithm for Boolean

matrix multiplication, and a better dynamic SSSP algorithm would naturally give a better Boolean matrix multiplication algorithm. The fastest known combinatorial algorithm for this task runs in O(mn) time [3], and finding faster algorithms for this task is a well-known and challenging problem.

3 Toward fully dynamic APSP

3.1 Randomized decremental APSP

Now armed with a working and powerful decremental SSSP algorithm, we can try to extend the algorithm to make headway on the decremental *APSP* problem. The simplest approach would be to maintain ES-trees from every node in a graph $G = (V, E)$, updating each tree after every deletion. Queries for the shortest path between u and v would be answered by finding the ES-Tree with u as the source node and returning the level of v in that tree. The query time of this algorithm would still be $O(1)$, but the update time would jump to $O(mn^2)$. We can improve on this simple approach, by randomizing the algorithm, so that we get correct distance estimates not exactly, but with high probability. The hitting set lemma is useful here.

Lemma 3.1. *Hitting Set Lemma*
Let $S_1, \ldots, S_n$ be a collection of sets such that for all $i = 1, \ldots, n$, $|S_i| \geq R$ and $S_i \subseteq V = \{1, \ldots, n\}$. For any $c > 0$, there is a randomized algorithm whic runs in time $O(n)$ and finds a subset $S \subseteq V$ with $|S| \leq \frac{n(1+c)}{R} \log(n)$, such that with probability $\geq 1 - n^{-c}$ it holds that $S \cap S_i \neq \emptyset$ for all i. The algorithm does not need to know the values of $S_1, \ldots, S_n$. [1]

The proof for the lemma is given in [1]. The intuition for the hitting set lemma is that you are constructing a set S which tries to "hit" at least one of $S_1, \ldots, S_n$. The probability of S missing any one set is by construction n^{-1-c} and by the union bound, the probability of S missing all sets is n^{-c}. By setting c, we can ensure the probability of set S not being a hitting set is very low.

We can use the hitting set lemma to sketch out the following randomized decremental APSP algorithm, which returns correct distance estimates between *any* nodes with high probability, with total update time $\tilde{O}(mn^{1.5})$ and query time $\tilde{O}(\sqrt{n})$ [1]. We do the following:

- Construct a random hitting set S over the shortest paths between nodes in G of length at least $\sqrt{n}$ by selecting $O(n^{1/2} \log n)$ nodes uniformly at random.
- Construct $|S|$ ES-trees using each node in $s \in S$ as a source node. In addition, construct BFS trees up to height $\sqrt{n}$ for all nodes in the graph.
- Query each ES-Tree with root s for $d(s, u) + d(s, v) = D(u, v)$. Also query the BFS tree rooted at u for $d(u, v)$. The first takes constant time for each ES-Tree, since you just return the levels of u and v and sum. The second takes $O(1)$ time (the distance estimates are calculated during pre-processing). Return the minimum distance estimate found over the $n^{1/2} \log n + 1$ estimates. The query time is therefore $O(n^{1/2} \log n) = \tilde{O}(\sqrt{n})$.
- After an edge deletion, update all ES-Trees as in decremental SSSP (amortized $O(n|S|)$). Reconstruct the $\sqrt{n}$-depth BFS trees ($O(\sqrt{n} * n)$).

Correctness: If $d(u, v) \leq \sqrt{n}$ then querying the BFS tree from u will return the correct estimate. If $d(u, v) \geq \sqrt{n}$, then with high probability at least one of the ES-trees will return the correct distance estimate by the hitting set lemma.

Run-time analysis: For each edge deletion, you must update all ES-trees and ($\sqrt{n}$)-depth BFS trees. This takes over all deletions (worst case m deletions) in total $O(\underbrace{mnn^{1/2} \log n}_{\text{source node}} + \underbrace{m\sqrt{n}n}_{(\sqrt{n})-\text{trees}})$ time for a total update time of $\tilde{O}(mn^{1.5})$. Querying each ES-Tree takes constant time, since you just return the levels of

u and v and sum. This gives a query time of $O(|S|)$. Querying a single BFS tree takes $O(1)$ time (in our implementation, distance estimates are stored during pre-processing). Since we have to return the minimum distance over $|S|$ structures, the overall query time is $O(n^{1/2} \log n) = \tilde{O}(\sqrt{n})$.

Space usage: The ES-trees together take up $O(|S|(m+n))$ space and the BFS trees take $O(n(m+\sqrt{n}))$ space. The total space usage is $O(mn + n\sqrt{n})$

Implementation and Experiment. The randomized decremental APSP algorithm "rand-APSP" was implemented in C++ over the Fast ES-Tree class and follows exactly from the description above. A few key results are presented in Figure 4. The algorithm in practice is beautifully aligned with the theory; it scales just as predicted.

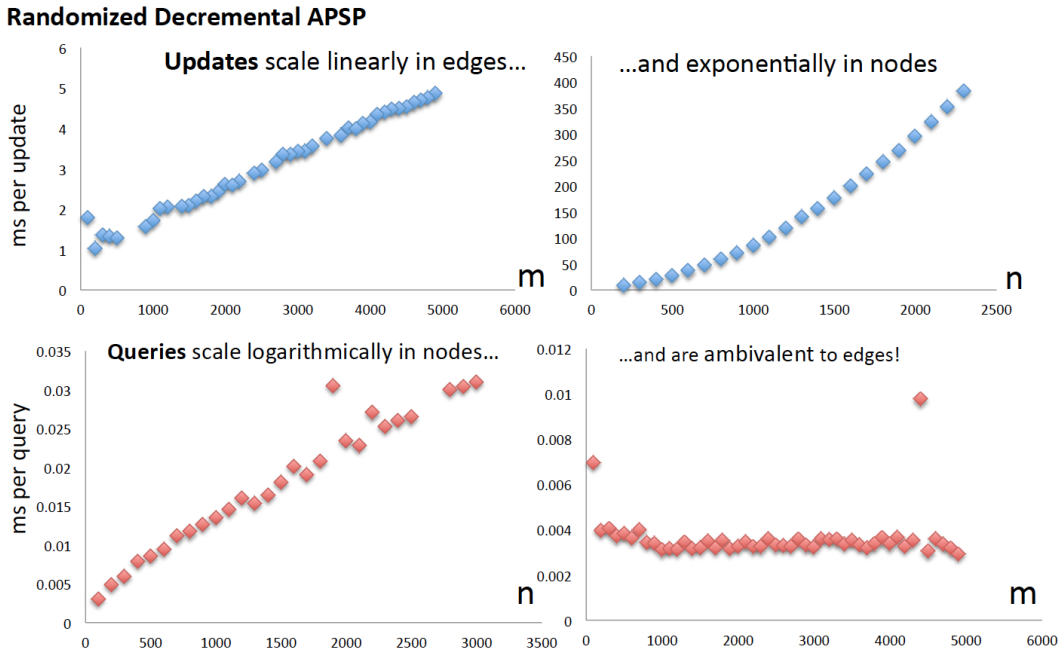

Figure 5: Experiment results for the randomized decremental APSP algorithm. Graphs shows update/query cost in milliseconds per update over 100 deletions/queries on a random graph.(Top left) Expected update time is $\tilde{O}(mn^{1.5}\log(n))$. Keeping n constant at 100 nodes, we see updates scale as $\sim O(m)$. (Top right) Keeping m constant at 5000 edges, we see updates scale as $\sim O(n^p)$, $p > 1$. (Bottom left) Expected query time is $\tilde{O}(\sqrt{n})$. For $m = 5000$, queries scale $\sim \tilde{O}(\log(n))$ as n increases (See Figure 6 for comparison with $log(n)$, $\sqrt{(n)}$). (Bottom right) There should be no dependence on m for query times; as expected, query costs are constant as m varies.

In terms of its run-time, this algorithm looks similar to the algorithm for fully dynamic APSP given by Roditty and Zwick [3]. Their randomized algorithm has amortized update time $O(m\sqrt{n})$ and worst-case query time $O(n^{3/4})$. rand-APSP has a better query time but a worse run-time by a factor of n. A closer look reveals that Roditty-Zwick's algorithm involves a few components we have already seen– the hitting set lemma, ES-trees, and a randomized decremental APSP algorithm! We explore this fully dynamic algorithm and see how we might leverage what we have built to find a simple solution to the fully dynamic shortest path problem in unweighted graphs; this analysis is relegated to the appendix, because it is a "quick and dirty" analysis, but still quite interesting!

4 Conclusion

In this paper, we have explored dynamic shortest paths algorithms on unweighted graphs, focusing on implementation and analysis of decremental SSSP derived from the seminal paper by Even and Shiloach [2].

We have selected and presented a result from [3] to give the reader an indication of the difficulty of decremental SSSP problems. We have also described and implemented an extension of the decremental SSSP algorithm into a decremental APSP algorithm via randomization. As a fun exercise (see appendix), we attempt to bridge partially dynamic and fully dynamic shortest paths problems by replacing components of a fully dynamic APSP algorithm with decremental SSSP/APSP components we have implemented, whose run-times we have tested; we then predict the theoretical behavior of the new, gutted algorithm.

All implemented algorithms can be found hosted at `https://github.com/nanand2/decrementalSSSP`

5 References

[1] Vassilevska-Williams, Virginia. Course notes, CS 267: Graph Algorithms, Winter 2016. Stanford University. `http://theory.stanford.edu/~virgi/cs267/`

[2] Shiloach, Yossi, and Shimon Even. "An on-line edge-deletion problem." Journal of the ACM (JACM) 28.1 (1981): 1-4.

[3] Roditty, Liam, and Uri Zwick."On dynamic shortest paths problems." AlgorithmsESA 2004. Springer Berlin Heidelberg, 2004. 580-591.

[4] King, Valerie. "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs." Foundations of Computer Science, 1999. 40th Annual Symposium on. IEEE, 1999.

[5] Demetrescu, Camil, and Giuseppe F. Italiano. "A new approach to dynamic all pairs shortest paths." Journal of the ACM (JACM) 51.6 (2004): 968-992.

[6]Thorup, Mikkel. "Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. Algorithm Theory-SWAT 2004. Springer Berlin Heidelberg, 2004. 384-396.

[7] Henzinger, Monika, Sebastian Krinninger, and Danupon Nanongkai."Dynamic approximate all-pairs shortest paths: Breaking the $O(mn)$ barrier and derandomization." Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on. IEEE, 2013.

6 Appendix

6.1 Fast ES-tree algorithm

This algorithm is analogous to Algorithm 1 and only varies slightly in its implementation and run-time. We keep three disjoint sets for each node $v \in V$, $N_1(v) = \{i \in V | l(i) = l(v)-1\}$, $N_2(v) = \{i \in V | l(i) = l(v)1\}$, and $N_3(v) = \{i \in V | l(i) - 1 = l(v)\}$ [1]. We drop a node v if in the course of deleting an edge and updating the data structure, we find that $N_1(v)$ is empty. This indicates there node's current level is not the correct shortest path estimate to the node, and we must drop it down at least one level. The drop procedure is outlined in pseudocode in Algorithm 2. Over a sequence of deletes, each node can be dropped no more than m times. This gives a total update cost of $O(mn)$.

6.2 "Quick and dirty" fully dynamic APSP with ES-Trees

As a quick, final exercise, we will try to alter Roditty-Zwick's algorithm for fully dynamic APSP to only rely on structures we have built and predict the update and query time of our chimeric algorithm. Roditty-Zwick's algorithm for fully dynamic APSP is given in detail in [3], and we will only discuss it at a high level here. The algorithm involves a series of generalized insertions and deletions. Insertions are always around an insertion center— all inserted edges have the insertion center in common. Deletions are of arbitrary sets of edges.

The algorithm requires three structures

1. A randomized decremental APSP algorithm
2. Full-depth shortest paths trees for vertices in a hitting set of size $O(cn \log(n)/k)$
3. Depth-k ES-trees for every insertion center

The algorithm runs in phases. At the beginning of each phase, the decremental APSP algorithm is initialized and all full depth trees are built from the hitting set. The set of insertion centers is empty. After a generalized edge insertion, all full depth trees are rebuilt and the depth k ES-tree is built for the insertion center. After a generalized edge deletion, all full depth trees are rebuilt and all depth k ES-trees are updated.

To query the algorithm for a path length $d(u, v)$, we query the decremental APSP algorithm, each full depth tree, and each depth k trees. The minimum value estimate is with high probability the correct shortest distance.

Many details are omitted, including the proof of correctness, to get to the interesting part, which is: what would the update and query time of the algorithm be if we used the randomized decremental APSP algorithm (built on ES-trees) from section 3.1?

Querying involves

- Querying the decremental APSP algorithm: $\tilde{O}(\sqrt(n))$
- Querying the depth-k ES-trees: $O(|C|)$. We can select $|C| \leq \sqrt(n)$ by starting a new phase if $|C|$ exceeds $\sqrt(n)$
- Querying each pair of full depth trees in the hitting set: $O(n \log(n)/k)$

The total cost of each query is $O(\sqrt{n} \log(n) + n \log(n)/k)$.

For the updates we have that

- For the randomized decremental SSSP, total updates are bound by $\tilde{O}(mn^{1.5})$
- Each insert creates a depth-k ES-tree; total updates are bound by $O(mk)$.
- Each insert or delete requires rebuilding $O(cn \log(n)/k)$ full depth trees; cost is $O(mn \log(n)/k)$

Algorithm 2: Decremental SSSP with Fast ES-tree

// Inputs

- g: pointer to underlying Graph object $G = (V, E)$ with m edges and n nodes
- s: source node in G

// Data structures;

- Q: global set storing nodes whose levels might need to be changed
- N_1, N_2, N_3: pointers to an array of neighbor sets for each node i in the graph
- E: the ES-tree, an initially empty Graph which is built via BFS during the Initialize phase of the algorithm
- l: integer array of length n where $l[i]$ gives level $l(i)$ in g

Initialize
- //same as Algorithm 1

Query(i)
- //same as Algorithm 1

Delete(i, j)
- **if** $l(i) = l(j)$ **then**
  - Erase i from $N_2[i]$
  - Erase j from $N_2[j]$
- **if** $l(i) < l(j)$ **then**
  - Erase j from $N_3[i]$
  - Erase i from $N_1[j]$
  - **if** $N_1[j]$ *is empty* **then**
    - Push j onto Q
    - UpdateLevels()
- **if** $l(i) > l(j)$ **then**
  - Erase i from $N_3[j]$
  - Erase j from $N_1[i]$
  - **if** $N_1[i]$ *is empty* **then**
    - Push i onto Q
    - UpdateLevels()

UpdateLevels
- // drop nodes in Q until Q is empty
- **while** Q *is not empty* **do**
  - Pop first (arbitrary) node y from the queue
  - $l(y)++$
  - **for** i *in* $N_2[y]$ **do**
    - Insert y into $N_3[i]$
    - Erase y from $N_2[y]$
  - $N_1[y] = N_2[y]$ //update N_1 to equal previous N_2
  - **for** i *in* $N_3[y]$ **do**
    - Insert y into $N_2[i]$
    - Erase y from $N_1[y]$
    - //check if neighbor now needs to be dropped a level
    - **if** $N_1[i]$ *is empty and* $l(i) < m + 1$ **then**
      - //node stops dropping after it reaches level $m + 1$
      - Push i onto Q
  - $N_2[y] = N_3[y]$
  - $N_3[y] = \emptyset$
  - //determine if you have to drop node another level
  - **if** $N_1[y]$ *is empty and* $l(y) < m + 1$ **then**
    - Push i onto Q

The total amortized cost of each update is $O(mn^{1.5}\log(n) + mk + mn\log(n)/k)$. We find the optimal $k = \sqrt{\log(n) + (n/m)log(n)}$. For $m >> n$, $k = \sqrt{log(n)}$, amortized update cost is $O(mn^{1.5}\log(n) + m\sqrt{log(n)}) = O(mn^{1.5}\log(n))$, and query cost is $O(\sqrt{n}\log(n) + n\sqrt{\log(n)})$

As expected our fully dynamic algorithm is quite weak! But for small n, it might serve as a feasible fully dynamic APSP algorithm.

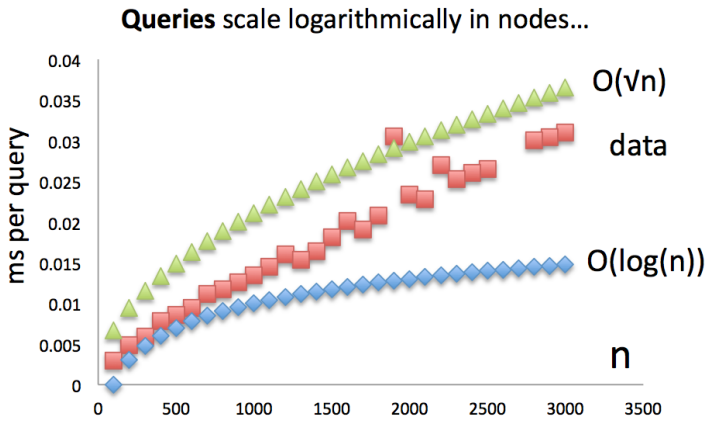

Figure 6: Experiment result for the randomized decremental APSP algorithm. Expected query time is $\tilde{O}(\sqrt{n})$. For $m = 5000$, queries scale $\sim O(\sqrt{(}n) \log(n))$ as n increases.