Fully Dynamic Connectivity Data Structure

Tanin Tyler Lux & Vasily Ilin

April 27, 2020

Contents

1 Overview of the report

This is a report on a semester-long project conducted Vasily Ilin and Tanin Tyler Lux as part of the Spring 2020 iteration of the CS591, Graph Mining class at Boston University. We implement and benchmark the Dynamic Connectivity data structure from a 1995 paper by Henzinger and King [4].

Section 2 is concerned with a brief description of the problem, previous known results, and the main ideas of the Dynamic Connectivity data structure. Sections 3 and 4 constitute the heart of the project and the report. In section 3 we describe the data structures pertinent to the implementation and methods they support. In section 4 we describe the experiments and benchmarks we have done with the Dynamic Connectivity data structure, as well as provide visualizations of our findings. We present our conclusions, further work, data & code, and team responsibilities in sections 5, 6, 7, 8, respectively.

2 Introduction

A fully dynamic algorithm is a data structure that maintains the computation as changes are made. In our case, the goal is to maintain connectivity ("are nodes a and b connected?") of an undirected graph as new edges are inserted and deleted. Intuitively, one would expect that the amortized cost of such maintenance should be low. However before this paper by Henzinger and King [4], the best known bound was $O(\sqrt{n})$ for an update, and $O(1)$ for a query [3], where n is the number of vertices in the graph. Henzinger and King exhibit an algorithm with an exponentially better time for these operations, namely, polynomial in $O(\log n)$. Specifically we can create a data structure such that updates (inserts and deletes of edges) are made in $O(\log^3 n)$ amortized time, and querying connectedness ("are a and b connected?") is in $O(\log n)$, amortized. We would like to note that in Henzinger and King suggest storing a b-tree with $b = \log n$ on the last level and within the adjacency trees, thus improving querying time. We forgo this added complexity and only use binary trees as it still gives us the polylogarithmic guarantees.

2.1 Main Ideas of Dynamic Connectivity Data Structure and Algorithm

In order to achieve the polylogarithmic runtime guarantees in expectation, the structure of the algorithm maintains an edge decomposition of the given graph $G = (V, E)$. Specifically, we partition the graph G into $L = O(\log n)$ levels (a parameter which will be referred to as max level), with $\{G_i = (V, E_i)\}_{i \leq L}$ being edge-disjoint subgraphs corresponding to the levels, s.t. $E_i \cap E_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=0}^{L} E_i = E$. We say level i is below level j if $i < j$. For each i, we maintain a spanning forest F_i such that $\bigcup_{i \leq k} F_i$ is a spanning forest of $\bigcup_{i \leq k} G_i$. A spanning tree T on level k is a one of the trees in $\bigcup_{i \leq k} F_i$. Therefore the union of all spanning trees of level i is our spanning forest $\bigcup_{i \leq L} F_i$. Within our data structure we don't store any of the F_i's, but rather just the spanning trees of each level. It is important to note that for $i < j$, $\bigcup_{k \leq i} F_k \subset \bigcup_{k \leq j} F_k$. Thus the set of spanning trees stored on the max level span our graph G, and when answering queries about connectedness we only look at the spanning trees on the max level.

The purpose behind storing these edges in levels is that we want an edge decomposition such that tree edges that have dense cuts (larger number of candidate edges that could reconnect the tree after this edge is deleted) exist in lower levels, while tree edges with sparse cut exists in higher levels. With this decomposition, we can devise an algorithm such that if a tree edge is deleted, if this edge exists in a lower level, we can sample non tree edges randomly $O(\log^2 n)$ times such that with high probability we find an edge that reconnects the tree. If an edge exists in a higher level, sampling is inefficient, as we most likely won't randomly encounter a cut edge, so in this case we would rather just find all possible cut edges directly and choose one to replace our deleted edge. This description is just a high level intuition into the algorithm and hides many details. Specifically, the decision to sample is really determined by the weight of a tree, where we define the weight of a tree $w(T) =$ the number of non tree edges on that level whose endpoints exists in the tree, where we double count edges where both of its endpoints exist in the tree. Thus, in general, trees that exists in higher levels have larger weights than trees that exist in lower levels. More on how edges are moved up and down can be found in the pseudo code for the replace function found in [4]. To support insertion of edges and maintain the proper edge level decomposition we just need to keep track of how many edges were inserted at each level and periodically rebalance and move edges up after the number of edges added at a level exceeds a certain threshold.

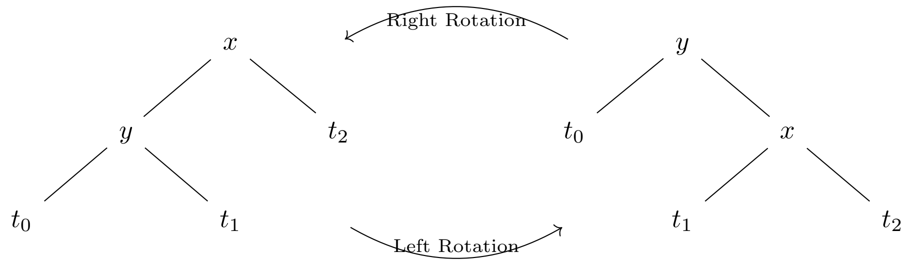
3 Data Structures and Implementation

3.1 Balanced Binary Tree

Before diving into our Fully Dynamic Connectivity Data Structure, we first want to explain two methods for maintaining balanced binary search trees. With a binary search tree, we know that we can make insertions in $O(h)$, where h is the height of the tree. Thus, it is advantageous to have a tree that is balanced where $h = \log n$, and we can insert and delete in $O(\log n)$. For any binary tree there is an order on the nodes, given by the *In-Order* of the tree: for any node, all nodes in its left subtree are smaller, and all nodes in its right subtree are bigger.

3.1.1 Rotate

The prevailing idea in both implementations is the use of rotates to balance a tree. Any given binary tree can be balanced by a sequence of rotations, and a rotate operation only changes a constant number $O(1)$ of parent-child relations while preserving the in-order traversal of the tree. There are two rotations we want to define: a **right rotation** and a **left rotation**.



Rotation: t_1, t_2, t_3 are the various subtrees of nodes x and y respectively

As the figure shows above, the parent child relation of nodes x and y are changed after a rotation, but their in order traversal remains the same. The right rotation in the figure corresponds to a right rotation at node y, while the left rotation corresponds to a left rotation at node x. For example the in order traversal of these two trees are

$$InOrder(t_0), y, InOrder(t_1), x, InOrder(t_2)$$

where $InOrder(t)$ is the in order traversal of tree t.

3.1.2 Join and Split

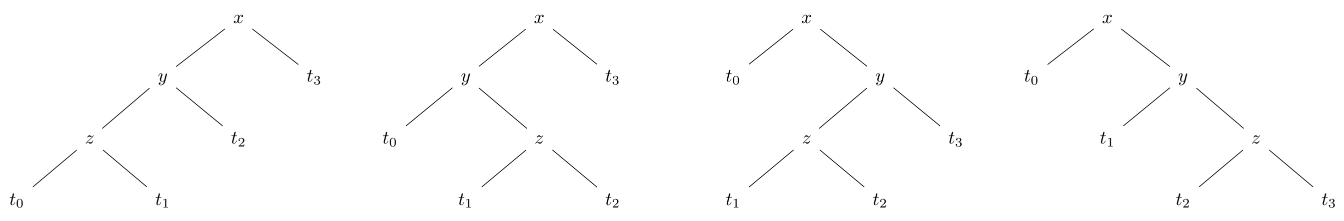
Within our balanced binary tree we define two functions, join and split. These functions will act in place of insert and delete, and can be implemented to work in $O(\log n)$ time. Join takes two trees, $t1$ and $t2$, and returns a new tree t such that the in order traversal of t is equal to the concatenation of the in order traversal of $t1$ and $t2$. Essentially, we are imposing the order that all the nodes in $t1$ are less than $t2$. With a split, we take as input a tree t and the position at which we want to split, and we return two trees $t1$ and $t2$ such that every node in $t1$ is before the position in the in order traversal of t, every node in $t2$ is after the position in the in order traversal of t, and joining $t1$ and $t2$ would result in the in order traversal of t. Split and join are (almost) inverses of each other. The "almost" part comes from the fact that there are two types of split on a position: a left split, which leaves the node at position in t_1, and a right split, which leaves the node at position in t_2.

3.1.3 Random Binary Tree

This implementation of a binary tree only guarantees $h = O(\log n)$ in expectation, that is inserts and deletes are $O(\log n)$ in amortized time [2]. While random binary trees are easier and faster to implement in practice, we lose out in the worst case runtime. The main idea of a random binary tree is that each node is assigned a random number (between 0 and 1 in our case) as its priority. We maintain an inner Heap-Order within the tree, meaning that for each node, the priorities of its children are smaller than its own priority. This is enforced through the join function by creating a dummy node whose left child is $t1$ and right child is $t2$, and rotating the dummy node down until it is a leaf, and then isolating this node. When we rotate the dummy node down, we rotate it with the child whose priority is larger, thus maintaining the Heap-Order which leads to $h = O(\log n)$ in expectation [2]. Notice that our join is $O(\log n)$ in amortized time, as we rotate at most h times where h is the height and each rotation takes constant time. The split is implemented in a similar manner, where we place the dummy node as a leaf node in our tree (node with no children) such that it is at the position we want to split in the in order traversal. Then we rotate this dummy node up until it reaches the root, and disconnect the dummy node, creating two split trees $t1$ and $t2$. This also runs in $O(\log n)$ in amortized time by a similar argument.

3.1.4 AVL Tree

An AVL Tree is a balanced tree such that $h = O(\log n)$ in the worst case. Thus, this tree gives us a stricter bound for the height of the tree, but at a computational cost. Specifically, after an insertion or deletion of a node, we must rebalance the tree with a set of specific rotations. To detect an imbalance, we define the *balance factor* of a node to be equal to $|height(\text{right subtree}) - height(\text{left subtree})|$. To make computing height $O(1)$, we store the node's height at each node, update accordingly during rotations. Within a balanced tree, every node's balance factor must be less than or equal to one. If not, then we must rebalance. To rebalance, we have four possible scenarios:



Configurations: Left, Left Right, Right Left, Right

Within the Left configuration, we simply rotate left at node x. For the Left Right configuration, we rotate left at y, then rotate right at x. For Right Left, we rotate right at y and then rotate left at x. Lastly, for Right, we rotate right at node x. Our implementation of AVL trees supports only a join and split operation as described earlier. We can implement a join in $O(\log n)$ time, and is based off of pseudo code found here [7]. Our implementation differs slightly in that we use a dummy node in place of the key that is in the middle of t_1 and t_2. That is when joining t_1 and t_2 we create a dummy node that is after the in order traversal of t_1 and before the in order traversal in t_2. After this step, we delete this dummy node from the AVL tree in $O(\log n)$ time, including all the necessary balancing. Because the initial join phase is also done in $O(\log n)$ time, the total cost of our join is $O(\log n)$. For our split, we follow the same procedure as outlined in random binary tree. Only extra detail is that the two resulting trees could be unbalanced, so we just need to rebalance the two trees which takes a total of $O(\log n)$, so our split takes a total of $O(\log n)$. Our implementation of deleting from and rebalancing AVL trees was helped along by this great resource [6].

3.1.5 Weighted Balanced Binary Tree

A further addition we make to our balanced binary tree data structures is a weight component at each node. Each node maintains its own weight, and its sub tree weight, which is equal to its own weight plus the sub tree weights of each of its children. The purpose of incorporating a weight to each node is to allow for $O(\log n)$ random sampling of nodes which we need in our implementation of our Dynamic Connectivity Data Structure. To do this sampling we implement locate w function which finds in $O(\log n)$ the in order sequence $x_1, x_2, ..., x_k, x_{k+1}, ...x_n$ the node x_{k+1} such that

$$\sum_{i=1}^{k} weight(x_i) < w \leq \sum_{i=1}^{k+1} weight(x_i),$$

essentially performing binary search on the sequence $s_1, s_2, ..., s_n$, where $s_k = \sum_{i=1}^{k} weight(x_i)$.

3.2 Euler Tour Tree and Adjacency Trees

Both our implementations of Euler Tour Trees and Adjacency Trees inherit from our weighted balanced binary tree, where you can choose between the AVL or Random Binary Tree implementation.

3.2.1 Euler Tour

In [4] they define the Euler Tour sequence ET(T) of a tree T by the sequence generated by the following pseudo code, starting from ET(root):

```
ET(x):
    visit x
    for each child c of x:
```

```
    ET(c)
    visit x
```

Note that we can rebuild a tree from its Euler Tour, but a tree can produce many different possible Euler Tours, depending on which node is chosen as the "root" of the tree. To see an example consider the following tree where we give one possible ET(y) sequence:

$$\begin{tikzcd} & y \arrow[dl, dash] \arrow[dr, dash] & \\ x & & z \end{tikzcd}$$

ET(y): $y_1 \quad x \quad y_2 \quad z \quad y_3$

An Euler Tour $ET(t)$ supports three procedures, **change root** of tree t from a to b, **delete edge** (a, b) from tree t, resulting in trees t_1, t_2, and **join** two trees t_1, t_2 by edge (a, b) resulting in a new tree t. The change root procedure takes in as an input $ET(t)$ rooted at a and produces a new $ET(t)$ where t is rooted at b with a constant number of joins and splits (from Section 2.1.2). Delete edge takes as an input $ET(t)$ and from it produces two Euler Tours $ET(t1)$ and $ET(t2)$ corresponding to the two trees as a result of the deleted edge with a constant number of joins and splits. Joining trees $t1$ and $t2$ after inserting edge (a, b) take as an input $ET(t1)$ and $ET(t2)$ and returns $ET(t)$ in a constant number of joins and splits.

Each edge within an Euler Tour can be characterized by four node occurrences (or three if one of the endpoints is a leaf node) such that we move over the edge when the Euler Tour goes from one endpoint of the edge and eventually back through to the other. For example the edge occurrences in our figure above for the edge (x, y) are $y_1 < x < y_2$. With these edge occurrences we can determine the sequence of splits and cuts in each of the procedures to produce the correct new Euler Tour. More exactly on the sequence of cuts and joins in each of the procedures and how to use these occurrences to represent an edge is outlined in [4, 1]. The important point is that because we can generate new tree encodings either through an insertion or deletion of a tree edge with a constant number of join and split operations, we can generate these new encodings, and thus trees, in $O(\log n)$ time.

3.2.2 Adjacency Tree

For each node within our graph, we maintain a weighted balanced binary tree of edges called an adjacency Tree. Each edge is given a weight of 1 and is a non-tree edge that is connected to our node. We support insertions and deletions of these non-tree edges into this adjacency tree through a constant number of joins and split operations, thus achieving $O(\log n)$ time amortized or in the worst case depending on whether we use random binary tree or an AVL tree.

3.2.3 Euler Tour Tree

The main idea of an Euler Tour in relation to our Dynamic Connectivity Data Structure is that we will store the sequence generated by $ET(t)$ within a weighted balanced binary tree such that the in order traversal of this weighted balanced binary tree corresponds to the Euler Tour sequence $ET(t)$. Note that each occurrence of a node in an Euler Tour is separate from any previous or future occurrences. For example, in $ET(y)$ above y_1 and y_2 correspond to the same node in the physical tree, but correspond to different nodes in the Euler Tour Tree. We designate one of these occurrences to be the active occurrence. Only active occurrences can have weights, and their weights correspond to the size of that node's adjacency tree (the number of non-tree edges connected to that node). This corresponds to the weight of the Euler Tour Tree t to be equal to $w(t)$ as briefly defined in Section 2.1.

3.3 Dynamic Connectivity Data Structure

As stated before, we have heavily referenced the C++ implementation from [1] in implementing this Dynamic Connectivity Data Structure within Python. However, it is important to discuss some differences in implementation. Due to the absence of pointers in Python, a lot of the code had to be adapted to function properly under these new circumstances. Also, we built the Dynamic Connectivity Data Structure on top of Python's NetworkX package, which is desirable due to its popularity within the Python community. That way, anyone can create various types of undirected graphs within NetworkX and use our Dynamic Connectivity to answer queries about the connectivity of the graph in a dynamic manner. Due to how NetworkX represents edges as tuples, we had to modify the implementation to support edge representations as edges, leading us to impose an ordering on the edges such that the source of an edge is always the node with the smaller numerical value, and the target is the node with larger numerical value. We also eliminated some unnecessary stored variables to trim down the space complexity. Additionally, we have implemented AVL trees alongside with random binary trees used in [1] in order to compare the performance of the two.

The Dynamic Connectivity Data Structure consists of the following attributes:

1. A reference to the NetworkX graph. Within the graph each node stores a data structure called DynamicConNode, and each edge stores a data structure called DynamicConEdge.

    - The DynamicConNode data structure stores a list, where the i^{th} entry holds the active occurrence of the node at level i. It also stores a list, where the i^{th} entry holds the adjacency tree of the non tree edges connected to this node at level i.

    - The DynamicConEdge data structure stores the level of the edge. It also stores the two adjacency trees of edges corresponding to the two nodes of the edge. Lastly, it stores the four tree occurrences that characterize the edge

2. Some constants (for example the number of levels) to ensure the $O(\log^3 n)$ amortized deletion time and $O(\log n)$ amortized query and insertion time

3. A list that stores the number of edges added to each level. This is used in conjunction with a list that stores the rebuild bound constants that determine after how many edges added to level i we need to rebuild level i and move edges to the level below i.

4. A list that stores the tree edges that exist in each level. This is accompanied by a list that stores the non-tree edges that exist in each level.

5. Dummy nodes for Euler Tour Trees and Adjacency Trees to use.

4 Experiments

In this section we present the performance of the Dynamic Connectivity data structure on various graphs and compare it to the performance of BFS as the baseline. For the performance evaluation we use two generated graphs: $G_{n,p}$ and a disjoint union of two complete graphs, and one real data set.

4.1 Baseline

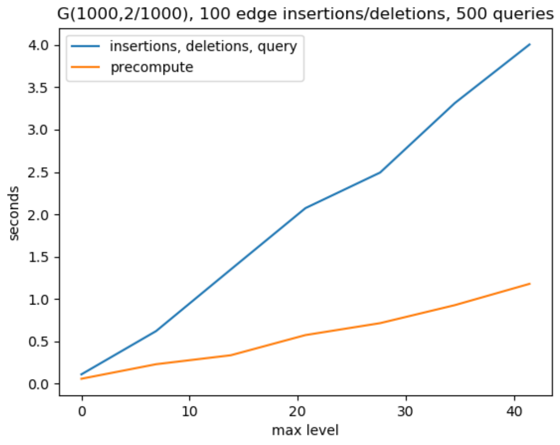
In order to put the performance of the Dynamic Connectivity data structure in perspective we used a baseline algorithm for answering the query "are nodes a and b connected?" This algorithm is the vanilla BFS on a. If b is connected to a's if and only if b is in a's connected component, which is detected via BFS. There are ways to optimize this baseline such as aborting BFS once b is found but we did not go in that direction. The relevant characteristic of BFS is that it takes $O(m)$ time, where m is the number of edges.

4.2 Erdos-Renyi graph

We have done benchmarking of the Dynamic Connectivity data structure on graphs sampled from $G_{n,p}$. We used $p = 2/n$, because it keeps the graph sparse enough not to have one giant connected component. The defining characteristic of $G \sim G_{n,2/n}$ is sparsity. In expectation, G has $\frac{n(n-1)}{2} \cdot \frac{2}{n} = n - 1 = O(n)$ edges.

Since our graph needs to be dynamic for the Dynamic Connectivity data structure to be of use, we decided to do 50 edge edge deletions, 500 edge additions and 500 queries for each benchmark. Let $T_d(n, L)$ be the random variable equal to the time to do the 100 edge additions and deletions on a graph $G \sim G_{n,2/n}$, and 500 queries using the Dynamic Connectivity data structure with max level L. Let $P(n, L)$ be the precomputation time for $G \sim G_{n,2/n}$.

Figure 1: $T_d(d, L)$ and $P(n, L)$ vs L

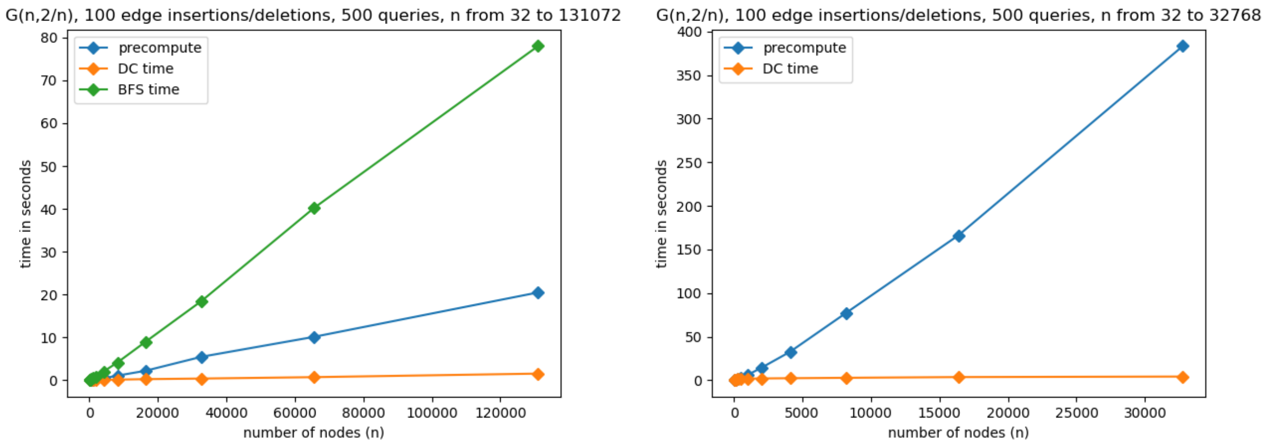


4.2.1 Max level

First, we measured $T_d(n, L)$ and $P(n, L)$ for $L = 0, \log n, \ldots, 6 \log n$ for $n = 1000$. The (somewhat) surprising finding is that the Dynamic Connectivity data structure works best with $L = 0$. The values of $T_d(d, L)$ and $P(n, L)$ are shown in figure 1. It is interesting to note that both relationships are basically linear.

To verify that this is not an artifact of the few number of nodes (only 1000), we have measured $T_d(n, L)$ and $P(n, L)$ for $L = 0$ and $L = 6 \log n$ for values of n in $\left\{2^k : 5 \leq k \leq 15\right\}$. The values are shown in figure 2. It is clear that the precomputation takes way longer for $L = 6 \log n$, and even though the scale does not allow us to see it, $T_d(n, 6 \log n)$ is higher than $T_d(n, 0)$.

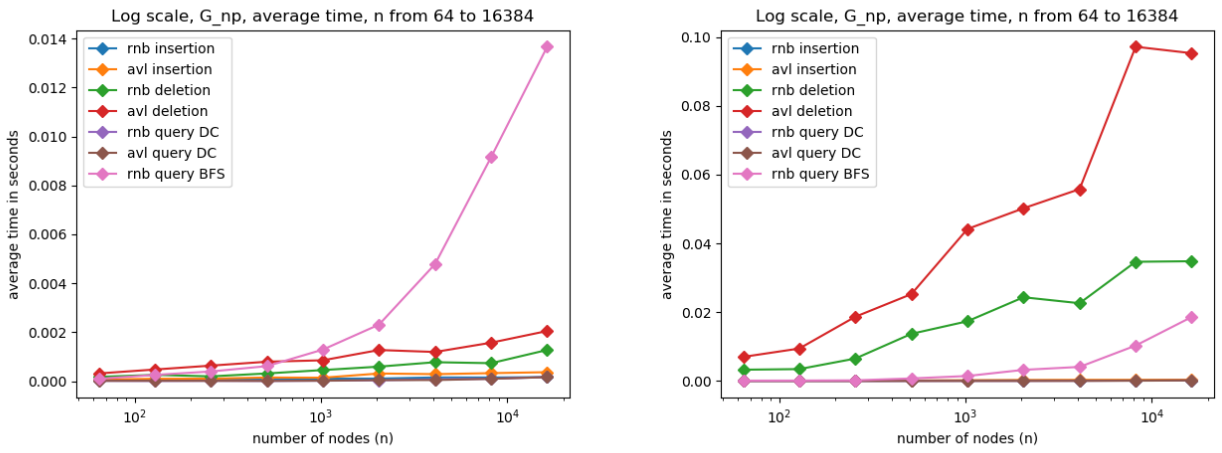
Figure 2: $T_d(d, L)$ and $P(n, L)$ for $L = 0$ (left) and $L = 6 \log n$ (right)



4.2.2 Benchmarks and Comparison with BFS

Now when we know that the maximum level should be kept at 0, we can compare the performance of the Dynamic Connectivity data structure with BFS. Figure 3 shows the performance of the Dynamic Connectivity data structure compared to that of BFS for $L = 0$ and $L = 6\log n$. We see that the Dynamic Connectivity data structure outperforms BFS if $L = 0$ but does worse when $L = 6\log n$. We can also observe that edge deletion is the most costly operation. The number of edge deletions is 50, while the number of edge insertions and queries is 500.

Figure 3: average times for different max level on $G_{n,p}$



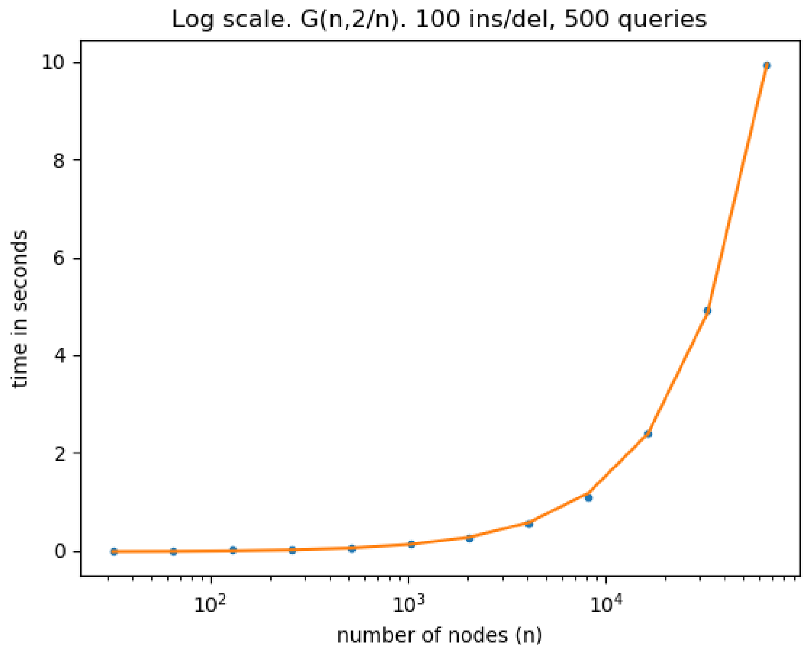
4.2.3 Polynomial fit

To demonstrate that the performance of the Dynamic Connectivity data structure is indeed polylogarithmic in the number of nodes, we fitted a quadratic polynomial to $T_d(n, 6\log n)$ plotted against n, with max level set to 0. Figure 4 shows that the fit is almost perfect, as far as human eye can see. We conclude that $T_d(n, 6\log n)\ \log^2 n$, as expected.

4.3 Disjoint union of complete graphs

We saw that the Dynamic Connectivity data structure outperforms (with $L = 0$) BFS on $G_{n,2/n}$, a sparse graph. We now compare the performance on a graph $H_n = K_{n/2} \sqcup K_{n/2}$, a union of two complete graphs on $n/2$ nodes. Same as above, we compare the average time for each operation: deletion, insertion and query. The number of edge deletions is 50, while the number of edge insertions and queries is 500, as before. Figure 5 shows that on H_n the Dynamic Connectivity data structure with max level set to $6\log n$ outperforms that with max level set to 0. Another difference of H_n compared to $G_{n,p}$ is that edge deletions take as long as BFS queries. Both difference are likely due to the density of H_n: H_n has $(n/2)(n/2 - 1) = O(n^2)$ edges.

Figure 4: polynomial fit to $T_d(n, 6 \log n)$ vs n



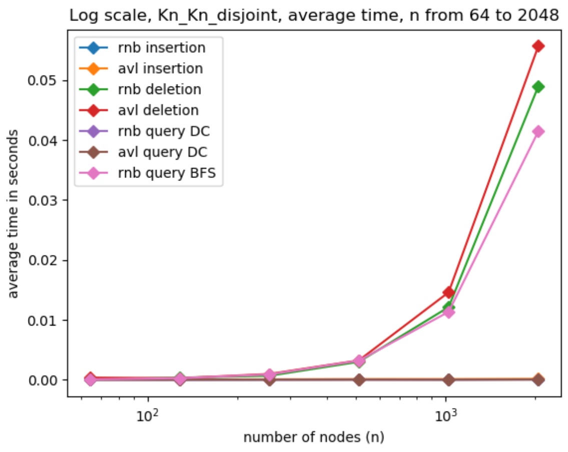
4.4 Real Data set

To show how the Dynamic Connectivity data structure performs on a real data set we benchmarked it on the email-Eu-core temporal network [5]. The data set consists of 986 nodes, and over 300,000 edge additions, each representing a sent email. Since there are no edge deletions in the data set, we had to simulate it by randomly deleting edges. We performed 100,000 edge additions, out of the 300,000 in the data set, 10,000 edge deletions, and 500,000 queries. The average time it took for each operation, including querying using BFS, is shown in figure 6. We see that the Dynamic Connectivity data structure is more efficient than BFS if the number of queries is at least twice the number of edge deletions. This testing was done with max level of the Dynamic Connectivity data structure set to 0.

4.5 Low max level can be very bad

As we saw, the denser the graph is, the better the Dynamic Connectivity data structure does with max level set to $6 \log n$, as opposed to 0. To see a concrete example where setting max level to 0 can be very bad consider H_n augmented with k bridge edges between the two complete graphs, for $k = O(1)$. Consider an adversary who is aware of the structure of the graph, and whose goal is to make the Dynamic Connectivity data structure do badly. The adversary will remove the k bridge edges one by one. As shown in [4], if the max level is $6 \log n$, edge deletion takes (amortized) polylogarithmic time. However, if max level is 0, deletion takes $\Omega(m) = \Omega(n^2)$ time. Let's see

Figure 5: average times for different max level on H_n



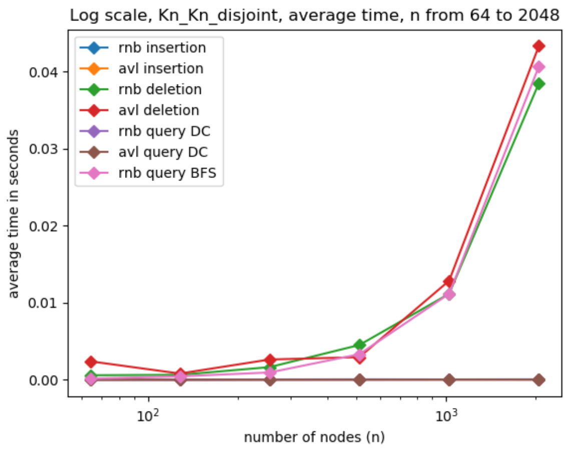
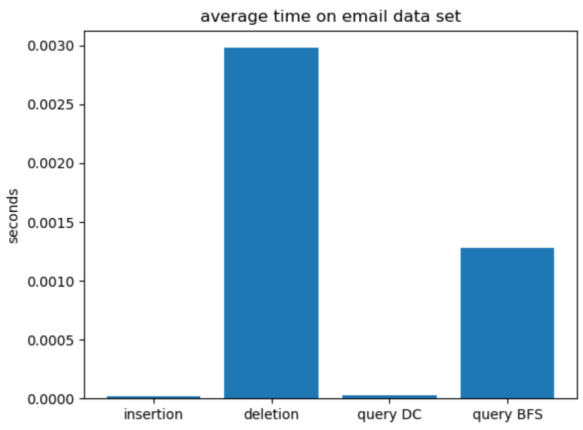


Figure 6: Average times



why.

When an edge is deleted, it splits its respective tree (on level 0, since max level is 0) into two trees T_1 and T_2. WLOG, the weight w_1 of T_1 is smaller than the weight of T_2. However, in our case, the two weights are asymptotically equal, and are $\Omega(n^2)$, since the weight of a tree is proportional to the number of non-tree edges, and there are $\Omega(n^2)$ of those in each $K_{n/2}$. When we sample non tree edges of T_1, looking for a reconnecting edge, there are at most k edges that will reconnect, and $\Omega(n^2)$ edges to choose from. Thus, the time it takes to find a replacement edge in this case is a geometric random variable with expectation $\Omega(n^2)$. Moreover, by Markov inequality, the probability it takes $\Omega(n^2)$ samples to find a replacement edge is at least $1/2$ (an extremely loose bound but good enough). Thus, with probability at least $1/2^k$, removing all k edges between the two complete graphs will take $\Omega(n^2)$ time, aka very bad.

5 Conclusion

5.1 Optimal max level

As we saw in our experiments above, the optimal max level for sparse ($m = O(n)$) graphs is 0, at least for graphs with no more than 10^5 nodes. However, for dense graphs ($m = \Omega(n^2)$) the optimal max level is $6 \log n$, as suggested in [4]. This is backed up by our example in section 4.5.

5.2 Dynamic Connectivity vs BFS

Throughout our experiments we saw that the Dynamic Connectivity data structure performs edge insertion and queries extremely fast, orders of magnitude faster than BFS. Edge deletion is the bottleneck of the data structure, sometimes taking as long as linear time $O(n)$, like in figure 5, and can be even worse, as shown in figure 3. When it comes to the real data set, as long as edge deletions are rare (at least twice more rare than queries), the Dynamic Connectivity data structure outperforms BFS.

5.3 AVL Tree vs Random Binary Tree

From our plots, we see that AVL trees perform slightly worse than or random binary tree. While AVL trees do a better job in maintaining a balanced tree, the number of rotations it does to achieve this is much larger than that of a random binary tree. Because the random binary tree is balanced in expectation, as we deal with larger and larger graphs, we expect that the difference between the best achievable height in a perfectly balance tree as in an AVL tree and the random binary tree to only vary slightly. Thus, the improvement of queries to the AVL tree (like locating a node) relative to the random binary tree are small and not proportional to the extra computation needed to keep the AVL tree perfectly balanced.

6 Further Work

Further work might include:

- Finding out the density of the graph, where switching from max level 0 to max level $6 \log n$ is viable. This includes theoretical work, as well as more experimentation.

7 Data

The data, plots and code are available at the Gihub repository github.com/Vilin97/CS591-Project.

8 Team Responsibilities

Tyler did most of the implementations: Random binary trees, AVLTree, Euler Tours, Dynamic Connectivity. Vasily wrote Adjacency Trees and a couple methods here and there. Vasily wrote the tests of correctness and did the benchmarking. We worked together on the report.

References

[1] D. Alberts. *Implementation of the dynamic connectivity algorithm by Monika Rauch Henzinger and Valerie King*. Freie Universität Berlin, Fachbereich Mathematik / B. Freie Univ., Fachbereich Mathematik, 1995. URL: `https://books.google.com/books?id=9jDjGwAACAAJ`.

[2] C. R. Aragon and R. G. Seidel. “Randomized Search Trees”. In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. SFCS ’89. USA: IEEE Computer Society, 1989, pp. 540–545. ISBN: 0818619821. DOI: `10.1109/SFCS.1989.63531`. URL: `https://doi.org/10.1109/SFCS.1989.63531`.

[3] David Eppstein et al. “Sparsification—a technique for speeding up dynamic graph algorithms”. In: *Journal of the ACM (JACM)* 44.5 (1997), pp. 669–696.

[4] Monika Rauch Henzinger and Valerie King. “Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation”. In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. STOC ’95. Las Vegas, Nevada, USA: Association for Computing Machinery, 1995, pp. 519–527. ISBN: 0897917189. DOI: `10.1145/225058.225269`. URL: `https://doi.org/10.1145/225058.225269`.

[5] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[6] Department of Math/CS - Emory College of Arts and Sciences. URL: `http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL.html`.

[7] Wikipedia. *AVL tree — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-4-2020]. 2020. URL: `https://en.wikipedia.org/wiki/AVL_tree`.