Decremental Tree Connectivity

Zack Swafford
Department of Computer Science
Stanford University
zswaff@stanford.edu

Alex Barron
Department of Computer Science
Stanford University
admb@stanford.edu

Sarah Tollman
Department of Computer Science
Stanford University
stollman@stanford.edu

1 Introduction

The decremental tree connectivity problem is a small part of the much larger set of dynamic graph connectivity problems. These in general refer to a graph on n nodes which can be changed in structure and can be queried for connectivity between two nodes. Without the dynamic or online component, any connectivity problem would be trivial–a simple $O(n)$ DFS during preprocessing could establish which nodes were connected to which others, allowing for queries in $O(1)$ time. Thus the problem includes queries intermingled with changes to the structure. Typically in a graph connectivity problem the nodes are taken to be constant and only the edges can be changed.

Different runtime bounds and tradeoffs can be found in a variety of different subproblems: if only edge insertions are allowed; if only edge deletions are allowed; if the graph is permanently restricted to a tree structure; if all of the changes happen in batches and any queries happen after a batch; if the results are amortized or expected or summed over a total of n operations; etc.

1.1 Problem Statement

Our particular problem is that of decremental tree connectivity. This means that the graph is restricted to be a tree when the model is initialized, and from there only edge deletions are allowed (at which point the graph will become a forest, but obviously cannot become more complex).

We say that there are m queries performed overall, and each could come at any time and must be correct (with no probability of error). Furthermore, there exactly $n - 1$ deletions, meaning that every single edge from the initial tree is eventually deleted. We then seek to improve the overall performance of the data structure, i.e. the sum total cost of all $m + n - 1$ operations.

1.2 Applicability

Generally, dynamic graph connectivity problems are applicable in a variety of contexts. For instance, networking and routing problems make use of graph structures and need to determine connectivity dynamically. Any other computing problem involving a graph is likely to have an interest in efficiently determining connectivity.

Decremental tree connectivity in particular is a more rare application than other dynamic connectivity problems, but it is actually a subproblem in many random methods for more general connectivity [3, 5]. With slight modification, this problem and its solutions can be applied to variations on UNION-FIND and leaf-finding problems [6].

The problem particularly excited us because research on it has stagnated the last few decades. It seemed unlikely that the problem was fully solved and explored, and our research indicates that it is still relevant in some contexts. We postulated that part of the reason the problem has remained essentially untouched since 1999 was the lack of a single, unified summary of the state of the problem. We hope that this overview (which should present every method necessary to implement these state of the art algorithms in a unified way) will enable further research. It is likely that this problem and its variants can be applied to more general dynamic connectivity problems, which are massively important in any application that uses a graph.

In the interest of presenting practical and useful results for further research, we have also created visualizations of the various methods which should clarify them and allow for some comparison. We have also conducted benchmark tests to practically compare the two state of the art methods, one of which is theoretically preferable but is hindered in its practical runtime by much larger overhead costs. Our code for both of these explorations is publicly available[1].

2 Methods

There are four different models that we explored to solve the complete decremental tree connectivity problem. Each data structure and associated algorithm is built on and an improvement to the previous. They are all explained below.

2.1 Unstructured

The unstructured method stores no auxiliary information about the graph; only the current status of the graph itself is retained (as it is in any model). A deletion is therefore very simple–the edge to be

[1] https://github.com/sktollman/DecrementalTreeConnectivity/

deleted is quite simply removed from the graph. Any individual query is then serviced with a full depth-first search (DFS) or any equivalent connectivity search.

2.2 DFS Labeling

In this method, a DFS similar to that mentioned above for the Unstructured method is preformed–but as a preprocessing step, not during query. As this search occurs, each individual node in the graph is additionally labeled with a number unique to its connected component. A connectivity query can now be implemented in a much more straightforward way simply by comparing labels of nodes–if two nodes share the same node label, they are in the same component and therefore are connected; otherwise they are not.

Deletion is slightly more involved. In this implementation, one of the two new connected components (which used to be just one component before the deletion) is selected and that entire component is relabeled with an entirely new index using a DFS to traverse the component.

2.3 Even-Shiloach

This data structure, proposed by Even and Shiloach in 1981 [2], is built off of the DFS Labeling mentioned above. In fact, the exact same preprocessing is done to augment the nodes with cluster information; the same very simple connectivity query implementation is also therefore possible.

The added efficiency of the Even-Shiloach method is introduced in the edge deletion method. Instead of running a naive DFS on one or the other of the connected components at random to relabel it, a parallel DFS is conducted starting from both components at the same time. The DFS terminates when one or the other of the searches does, and the component that finished first (and is therefore smaller) is relabeled with the new unique cluster label. The other component retains the whole component's original label. Because this search strategy need only search over the smaller component, it takes less amortized time and is more efficient in the total query time sum.

2.4 Alstrup-Secher-Spork

Our final method, which is theoretically optimal in total query time and total edge deletion time, was proposed by Alstrup, Secher, and Spork in 1997 [1]. In preprocessing, the graph is partitioned into smaller subtrees, creating a series of trees which join together to form one large-scale outer tree. In their paper, Alstrup et al. call this design the *MicroMacroUniverse*, where the smaller-scale trees are *microtrees* and the outer tree is the *macrotree*. We will use this terminology interchangeably with the terminology due to Fredrickson [4] (who initially proposed the method we use to segregate the tree in this way) wherein the microtrees are called *clusters*.

Explanation, analysis, and pseudocode for the method used to partition the initial tree into clusters are available in Appendix A. Suffice it to say that the microtrees have size $O(\log n)$, there are $O(n/\log n)$ of them overall, and each one contains only two *boundary nodes*, or nodes connected to other clusters. Boundary nodes are also reused as the only nodes in the macrotree. The size of the microtrees allows us to apply a variation on the Method of Four Russians to the problem. As per usual with this method, the segregation into micro- and macro-problems allows for the removal of a log factor in the runtime analysis. In our case here, the Even-Shiloach method is applied to the macrotree; during preprocessing the macrotree is preprocessed as dictated under that model.

Then, each of the microtrees is preprocessed to allow for very efficient (i.e. $O(1)$) *microdeletions* and *microqueries*. This is made possible by the small size of the clusters–each one's connectivity can be fully represented in a machine word under the transdichotomous machine model. In short, each cluster is given a machine word representing the edges that are connected as 1 and those disconnected as 0. Then each node is given a machine word written in the same schema representing its connectivity within the cluster. This microtree preprocessing is explained and properties about it proven in much more detail in Appendix B.

During an edge deletion operation, if the edge is in the macrotree, it is deleted there within the Even-Shiloach model. If the edge is within a cluster and a microdelete is therefore required, the representation of the edge in the cluster's machine word list of edges is flipped to 0 to represent that the edge no longer exists.

Queries can then be implemented using a combination of microqueries and macroqueries. If both of the vertices queried are in the macrotree, then a simple macroquery on the Even-Shiloach model of the macrotree can determine their connectivity. If both are in the same microtree, a microquery on that microtree will suffice. Microquery is implemented with bitwise logic and is explained fully in Appendix B. Otherwise, a more complex approach is required. Because one of the invariants of the clusters is that each only has two boundary nodes, the boundary nodes that each vertex can reach within its cluster can be determined with two microqueries, for a total of four microqueries between the two vertices. These boundary nodes, as explained above, are the nodes in the macrotree. There are then a maximum of four macroqueries necessary to determine whether each of the up to two boundary nodes connected to the first vertex is connected to one of the up to two boundary nodes connected to the second. If any of these connections exist, the vertices are connected; otherwise, they are not.

Because there are a constant number of these micro- and macroqueries required for any query, and because overall these queries are more efficient in the sum over all queries, the total time taken querying is bounded more tightly in this method than any other and it is theoretically superior.

3 Correctness

Our discussion of the correctness of these methods will be brief; for the most part their correctness is apparent through their design.

3.1 Unstructured

This model maintains no data. When a query is performed, a DFS is sufficient to determine connectivity.

3.2 DFS Labeling

During preprocessing, a single additional label is added to each node to demarcate its cluster. So long as these labels are maintained correctly, query's implementation, which compares labels between nodes to see if they are labeled as in the same cluster, is trivially correct. These labels are maintained through the lifetime of the data structure by running a DFS and relabeling the newly created cluster as distinct from the other when the two are separated by a deletion.

3.3 Even-Shiloach

Even-Shiloach, much like DFS Labeling, is correct so long as the labels are maintained. In this model the relabeling is accomplished in a slightly more involved fashion by choosing the smaller new cluster, but this does not affect the correctness argument.

3.4 Alstrup-Secher-Spork

This method is obviously the most involved and its correctness the least trivial. As shown in Appendix A, the clustering algorithm correctly partitions the tree into microtrees with at most two boundary nodes each. As shown in Appendix B, the preprocessing done on these microtrees is sufficient to correctly implement microquery and microdelete within the clusters.

Creating the macrotree under the Even-Shiloach model, with boundary nodes as vertices and edges between boundary nodes as vertices, correctly allows for macrodelete and macroquery because that model is correct.

Finally, a delete on this model updates the macrotree if necessary and a microtree if necessary, so all of the submodels are maintained. A query on two boundary nodes is trivially correct because Even-Shiloach is; a query within a cluster is correct by Appendix B. Any other query is implemented with a combination of micro- and macroqueries, making use of the fact that if any path exists between the two they are connected. If a vertex is connected to a boundary node in its cluster, that node is

connected to a boundary node in the other vertex's cluster, and that boundary node is connected to the vertex, a path exists. Query exhaustively tries all four of these possible paths, and if any exist it is true. If not, there is no path, and query returns false. This is correct.

4 Theoretical Runtimes

See below for a summary of the various methods' theoretical runtimes. As with correctness, we only examine and prove runtimes which are not trivial and apparent.

Table 1: Data Structure Comparison

Method	Preprocessing Time	Total Delete Time	Total Query Time
Unstructured	$O(1)$	$O(n)$	$O(m \cdot n)$
DFS Labeling	$O(n)$	$O(n^2)$	$O(m)$
Even-Shiloach	$O(n)$	$O(n \log n)$	$O(m)$
Alstrup-Secher-Spork	$O(n)$	$O(n)$	$O(m)$

4.1 Unstructured

The unstructured model obviously does no preprocessing, so this is $O(1)$. It also has no overhead for deletion, but simply removing each of the $n-1$ edges from the graph in $O(1)$ each makes the total delete time $O(n)$. Finally, there are no good guarantees at all on the query time. Each of the depth first searches in the worst case must search the entire graph of n nodes, and there are m of these searches. The total query time is therefore $O(m \cdot n)$.

4.2 DFS Labeling

Preprocessing for DFS Labeling, unsurprisingly, only involves a DFS to label. This can be conducted on the graph of n nodes in $O(n)$. Queries are very simple, and involve only a comparison of the labels of the two vertices. m of these queries makes their total time $O(m)$. Deletions under this model require another DFS on many of the nodes. In the worst case, these searches will require traversing $n-1, n-2, \ldots, 2$, and 1 nodes respectively. Unfortunately the sum of this series cannot be bounded past $O(n^2)$, so the total work in all deletions is $O(n^2)$.

4.3 Even-Shiloach

The Even-Shiloach structure implements preprocessing and querying in exactly the same way as DFS Labeling, so the bounds on these runtimes are the same. The improvement of this model appears with the deletion operation.

Instead of doing the DFS to relabel one the new components starting with one of the two nodes random, it does the search starting from the two affected nodes in parallel. Because the search terminates when the first search does, it is guaranteed to search in the size the smaller component, which allows us to tighten the theoretical bound on the total runtime–even though the constant factor is higher because both searches must be conducted and then the smaller component traversed again to relabel it.

The worst-case runtime in this case occurs when every edge deletion creates two components of exactly the same size; any other balance will create a smaller component that is smaller, and therefore cause less work. This worst-case possibility creates a sequence of $O(\frac{1}{2}n), 2 \cdot O(\frac{1}{4}n), \ldots, 2^{\log n} \cdot O(\frac{1}{n}n)$ operations. There are $O(\log n)$ different sizes, and $O(\frac{1}{2}n)$ operations need to be performed in total for each different size of grouping, so in the worst case the sum total of all of the deletions will take $O(n \log n)$ time.

4.4 Alstrup-Secher-Spork

Preprocessing the Alstrup-Secher-Spork model involves first clustering the tree into microtrees. As per Appendix A, this takes $O(n)$. Next, preprocessing each of the microtrees takes $O(t)$ for microtrees of size t as per Appendix B; since all of the trees' sizes in total is $O(n)$ this phase is also $O(n)$. The macrotree is on $O(n/\log n)$ nodes–a non-obvious result that is explored in Appendix A. Preprocessing an Even-Shiloach model takes $O(n_{ES})$ (where n_{ES} is the number of nodes in the Even-Shiloach model) so creating this structure takes $O(n/\log n)$ time. Therefore preprocessing as a whole takes $O(n)$.

Queries are each $O(1)$ because macroqueries in the Even-Shiloach model are $O(1)$ and microqueries are also $O(1)$ as shown in Appendix B. There are a constant number of these subqueries per query (4 micro- and 4 macro- in the worst case), so each query is $O(1)$ for a total query time of $O(m)$.

Any edges within clusters take $O(1)$ time to delete as shown in Appendix B, and there are no more than $n-1$ of them, so intra-cluster deletions are worst-case $O(n)$ in sum. Because the macrotree is built on $n_{ES} = O(n/\log n)$ nodes, the total runtime of all deletions in the Even-Shiloach model, as proven in the previous section, is

$$\begin{aligned} O\left(n_{ES} \log\left(n_{ES}\right)\right) &= O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) \\ &= O\left(\frac{n}{\log n}\left(\log n - \log\log n\right)\right) \\ &= O(n). \end{aligned}$$

Therefore, in sum, the deletion operations are bounded within linear $O(n)$ time.

The Alstrup-Secher-Spork model is theoretically optimal for the sum of the delete and query operations. There are m queries, and it is impossible for them to take less than time $O(1)$, so in sum they must take $O(m)$. The same argument holds analogously for the deletions in $O(n)$. Obviously constant factors can be changed and lessened, but in runtime analysis this structure is optimal. Further, we argue that preprocessing is also theoretically optimal in runtime analysis. Without any preprocessing on an initially unstructured and simple graph, it is known that there is no way to answer connectivity queries in less than $O(n)$. For any improvement on this, the whole graph must be traversed, and that takes $O(n)$. Thus any improvement over the Unstructured model requires at least as much theoretical runtime as this model.

5 Practical Runtimes

In the interest of creating practical and useful results for future research, we have also conducted benchmarking tests on each of the models and have made the code for them available as well to enable intelligent decisions in an application-dependent context.

In principle, the Alstrup Secher Spork model is optimal, and certainly in the limit it is. However, if queries are extremely rare and preprocessing is undesirable, the Unstructured model actually offers an improvement in that it requires no preprocessing.

Even beyond this perhaps quite unusual case, we postulated that in practice for most applications the Even Shiloach model would be preferable because it requires much less space overhead, has a much smaller constant factor on preprocessing and querying, and the additional $\log n$ factor during query is unlikely to outweigh the higher constant factors except for unthinkably large n. In practice, this is indeed what our benchmarking tests show. Figures 1-3 below show preprocessing, total query, and total delete times for variety of different values of n. In these tests, the number of queries m is equal to n.

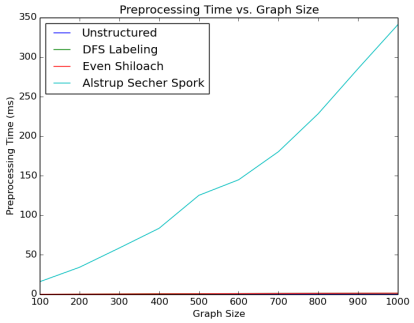

Figure 1: Preprocessing

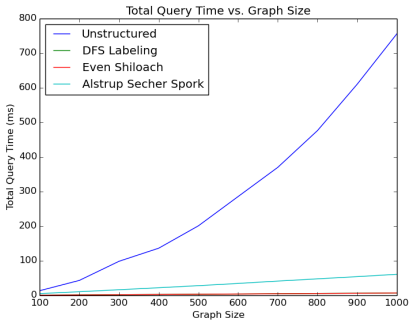

Figure 2: Query

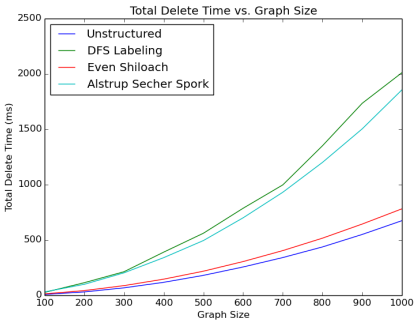

Figure 3: Delete

We see that our expected results are very much borne out in practice. Alstrup Secher Spork takes a comparably massive amount of time to preprocess. Queries are obviously much, much worse in the Unstructured model–but the two other less naive methods are still far better than better than Alstrup Secher Spork because their constant overhead is smaller.

Deleting can be done in much the expected times, but the simplicity and improvement of Even Shiloach over DFS Labeling is perhaps the biggest surprise from all of the benchmarking tests. Even Shiloach is comparable in delete time to the Unstructured model, which simply removes the edge from the graph! Clearly searching only the smaller cluster is a massive practical improvement over searching and relabeling one at random.

We also examined total operation time over the lifetime of the data structure for a variety of ratios of m to n. Depending on the application, it is likely important to choose a data structure to suit the size and the expected number of queries. Figures 4-6 below show total operation time for various n with $m = n$, $m = 100n$, and $m = n/100$, respectively.

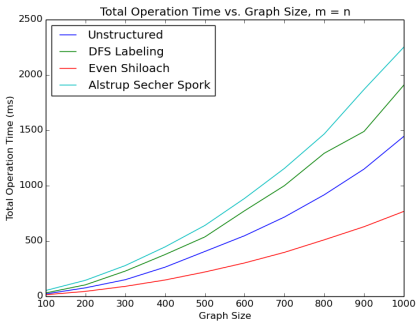

Figure 4: Total Time, m = n

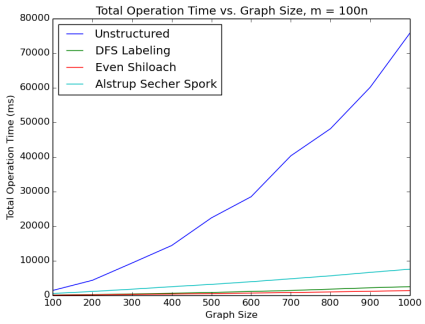

Figure 5: Total Time, m = 100n

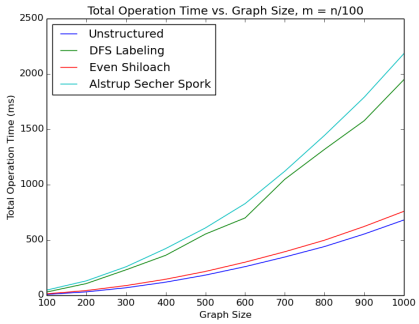

Figure 6: Total Time, m = n/100

Except in the case where there are disproportionately many queries (and the Unstructured model unsurprisingly underperforms), our theoretically optimal data structure is practically speaking the worst. In every case, Even Shiloach is either the best performer or on a par with it. It is obviously on a similar level with DFS labeling for every algorithm except delete, but as we saw in the individual breakdown Even Shiloach is quite a bit better in that case.

We therefore conclude that for any non-hypermassive graph, Even Shiloach is the preferable method for solving the decremental tree connectivity problem. Its implementation is actually comparable in complexity to that of the Unstructured or DFS Labeling methods; each includes only one depth first search. It takes the same amount of space as DFS Labeling as well (one extra word per node), which is also obviously much less than Alstrup Secher Spork. For all practical intents and purposes, Even Shiloach is the winner.

6 Modeling

We also created visualizations of the four data structures we have discussed to aid understanding.[2] Our visualizations run simultaneously for visual comparison. The user can create the initial tree, opt for our default, or generate a random tree. The program does not allow the user to preprocess a graph that is not a valid tree. Once the graph is set, the user can preprocess the tree, then run a sequence of queries and deletions as she pleases. The user can clear the graph to restart at any point. We believe that this visualization makes the data structures significantly easier to understand.

During the DFS for Unstructured, the current node is highlighted in red, enqueued nodes are yellow, and searched nodes are grey. The preprocessing DFS for DFS Labeling and Even-Shiloach is almost identical, except that searched nodes are colored according to their connected component. At query time, if the query is successful the node being queried will be highlighted in green.

[2]The visualization is published at http://web.stanford.edu/~stollman/DecrementalTreeConnectivity/

The border color of a node corresponds to its group; for Unstructured there are no groups, for DFS Labeling and Even-Shiloach a group is a connected component, and for Alstrup-Secher-Spork a group is a microtree. The groups for the Macro Even-Shiloach structure in the Alstrup-Secher-Spork data structure are represented by the background colors of the nodes. The groups are set during preprocessing and changed during deletion.

For Alstrup-Secher-Spork, the boundary nodes and boundary edges are enlarged. The bit string for each node is a part of its label, and the bit string for each cluster is in the label of a boundary node of the cluster.

The Alstrup-Secher-Spork data structure only works when the maximum degree of each node is 3. We discussed above the algorithm for converting a tree into one that satisfies these requirements. This is animated, if applicable, at the beginning of preprocessing.

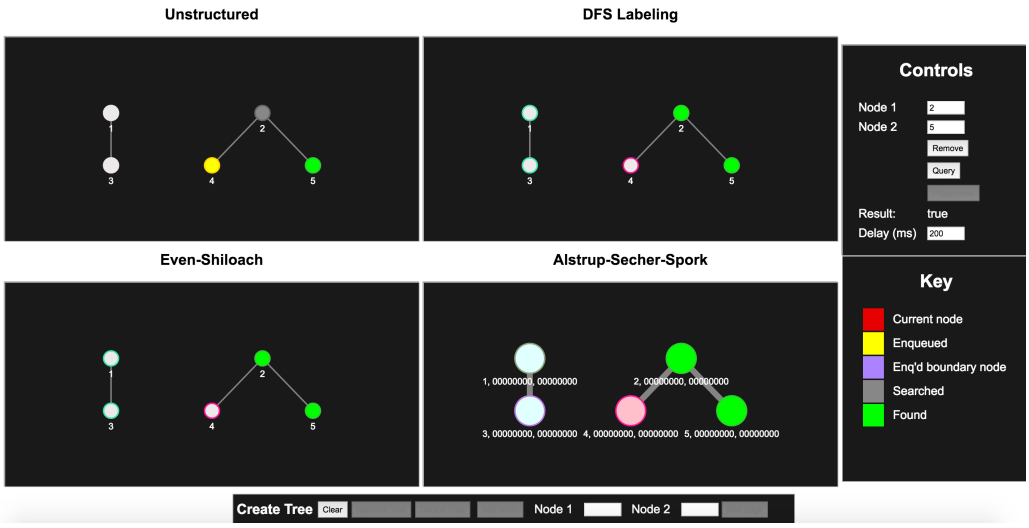

Figure 7: After a successful query

7 Further Work

Part of the exciting aspect of this comprehensive overview of the decremental tree connectivity problem is that we do not exactly know how this problem will be needed and used. It is certainly our expectation that this problem can be modified and altered such that it provides a useful background for many larger and more interesting dynamic graph connectivity problems.

It is our hope that this practical overview of the methods can provide a complete basis for understanding of the problem and the state of the art of its solution. The analysis is aggregated and complete where it was once disparate; the visualization tool should aid understanding and serve to elucidate particularly the complex and intricate Alstrup Secher Spork method; and the actual run-

time benchmarks will be useful in practice for anyone who needs to solve this problem in an actual application.

In summary, Alstrup Secher Spork is the theoretically optimal structure to solve this problem. However, Even Shiloach is our recommendation in almost any practical setting because it performs optimally or near-optimally in all categories for any typical n. Further, it has much smaller memory overhead and is much, much easier to implement. Alstrup Secher Spork is primarily interesting as a pedagogical application of the Method of Four Russians; we recommend that further research and exploration center around the Even Shiloach model.

8 Appendices

A Tree Clustering Methodology

This recursive method of clustering, initially proposed by Frederickson in 1985 [4], guarantees that the tree is partitioned into clusters of size $O(\log n)$ and that each cluster has at most two boundary nodes, which are the two invariants we require. Pseudocode for the method is below.

Listing 1: Microtree Clustering Psuedocode

```
T = the tree
n = T.numVertices
z = log(n)

//make equivalent tree with all vertices' degrees <= 3
for v in T.vertices:
  d = v.degree
  if d > 3:
    N = v.neighbors
    T.remove(v)
    T.addVertices(v1, v2,... vd)
    T.addEdges((v1, v2), (v2, v3),..., (vd-1, vd))
    T.addEdges((v1, N1), (v2, N2),..., (vd, Nd))

//cluster the nodes
clusters = []
r = random vertex from T with r.degree == 1
clusters += cluster(r)[0]

def cluster(v):
  clus, deg, sz = {v}, v.degree, 1
  for neighbor in v.neighbors:
    n_clus, n_deg, n_sz = cluster(neighbor)
    if deg + n_deg - 2 <= 2 and sz + n_sz <= z:
      clus, deg, sz += n_clus, n_deg, n_sz
    else:
      clusters += n_clus
```

First, we must show that the first section of the code does indeed transform the tree T into an 'equivalent' tree (for some definition of equivalent) whose vertices all have degree ≤ 3 and is still of size $O(n)$. In our case, equivalence is important in terms of connectivity only. The graph is still a tree because no cycles are introduced. Further, if we consider the new vertices as a whole $V = v_1, v_2, \ldots, v_d$, this group is entirely connected and also connected to all of v's original neighbors N. Thus the connectivity is exactly the same. The end user will not have access to the new edges, so V will always be fully connected and a connectivity query from any v_k will be equivalent to a query from the original v. Further, exactly one edge goes from each neighbor vertex to one of the v_ks; this is the edge that should be deleted when required. Each of the v_ks has either 2 or 3 incident edges, and all nodes not split have degree ≤ 3, so all vertices in the tree have this property. Finally, there are a maximum of $2n-2$ incident edges in the entire tree, and each of the splits creates exactly

$v.degree$ new nodes, so there are an absolute maximum of $2n - 2 = O(n)$ new nodes created. The tree as a whole therefore clearly still has size $O(n)$ as required.

We will now prove why this method creates clusters which maintain the necessary invariants. Because each potential addition to a cluster is attempted and examined before it is added, there is no point at which the variable $clus$ is not an appropriate cluster. Other half-formed clusters are potentially merged with it, but only if their size does not exceed $z = \log n$. If it does, the two clusters remain separate. Furthermore, the resulting cluster's total degree is correctly calculated as the two clusters' separate degrees minus two, $deg + n\_deg - 2$. This resulting degree is never allowed to exceed two, so each potential cluster is accepted only if it will have two or fewer boundary nodes. Any cluster with degree 3 must have exactly one node in it, so there is trivially only one boundary node in that cluster. Thus the clusters have size at most $O(\log n)$ and at most two boundary nodes. Because no cluster is ever formed which violates the invariants, they are generally maintained.

This algorithm takes $O(n)$ time. For the same reason that the size of the graph does not exceed $O(n)$, the first part of the algorithm which rectifies the tree to degree 3 takes $O(n)$ time. It does not need to do more than constant operations for each of $O(n)$ nodes. The second portion of the algorithm makes one recursive call for each node, and in each call loops over all of that node's neighbors. Again, there are exactly $n - 1$ edges in the graph, so the loop is executed exactly $2n - 2$ times throughout the n calls to $cluster(v)$. Only constant operations are performed in the loop, so overall this portion of the code and therefore the clustering algorithm in general takes $O(n)$ time.

The final requirement that we have of this clustering algorithm is that it make $O(n/\log n)$ clusters from the original tree of n nodes. This does not immediately follow from the fact that each cluster is $O(\log n)$; this is an upper bound so each cluster could theoretically have size 1 and there would then obviously be $O(n)$ clusters. Neither Frederickson [4] nor Alstrup et al. [1] explore this proof in a satisfactory manner, and it is nontrivial. We will sketch a proof for this fact here.

First, we must prove a lemma about complete trees that have no vertices of degree greater than 3, like our rectified tree. Namely, if a, b, and c are the number of vertices with degree 1, 2, and 3 respectively, then $a = c + 2$. There are clearly no other nodes of any degree. The number of nodes in the tree n therefore has $n = a + b + c$. Further, the total degree of the tree is $2n - 2 = a + 2b + 3c$. Subtracting the former from the latter twice yields $-2 = -a + c \implies a = c + 2$.

In short, we can now show that there will be $O(n/\log n)$ clusters because the clustering algorithm merges clusters and allows them to grow primarily based on whether they have size $\log n$–i.e. the limitation on the total degree of the cluster is minimal.

Adding a degree-2 node to a cluster does not change that cluster's degree, so we can completely ignore their effect in this instance. Because there are commensurate numbers of degree-1 and degree-3 vertices, and the two have opposite effects (decreasing the overall degree of a cluster by 1 and increasing it by 1, respectively), the two effects balance each other out in expectation.

While it is possible to design trees in an adversarial way, the starting node for the clustering algorithm is selected at random. There is no general structure which can ruin the balance of the two types of nodes, so in expectation this limitation can be entirely neglected. Therefore the clusters grow until they hit the limit of size $\log n$. Thus their size is $\Theta(\log n)$, which means that there will be $n/\Theta(\log n) = O(n/\log n)$ microtrees created by this process.

B Microtree Analysis

The microtrees are created per the algorithm in Appendix A with the invariants that they are size $t = O(\log n)$ and they have at most two boundary nodes. We now examine how to implement preprocess, microdelete, and microquery in times $O(t)$, $O(1)$, and $O(1)$ respectively for such clusters.

B.1 Preprocessing

To preprocess the microtrees, Alstrup, Secher, and Spork [1] suggest utilizing the transdichotomous machine model assumption. Because the number of nodes in the cluster t is $O(\log n)$, the number of edges $t - 1$ is as well, so the assumption states that this size can fit in a machine word. During preprocessing, a mapping is created from each edge in the microcluster to the index of a bit in a machine word w for that cluster. All edges initially exist, so they are 1s; any excess bits are 0. The map and w are maintained throughout the lifetime of the data structure.

Next, each node v in the cluster is assigned its own connectivity word $v.w$, initially 0. A DFS is then conducted starting from any node in the cluster. Each vertex is assigned its parent's word with an additional bit flipped from 0 to 1 corresponding in the mapping to the edge taken from that parent to the vertex. $v.w$ is now a word representing the edges necessary to get from v to the arbitrary root of the cluster for each node v.

This entire preprocessing step takes $O(t)$ if t is the size of the microtree. Creating the mapping from edges to bits takes time relative to the number of edges $(t - 1)$, and conducting the depth first search also takes time $O(t)$ with no additional big-O time for the bitwise logic required at each step.

B.2 Microdelete

Microdelete is very simple. The cluster's mapping is followed and the bit in w representing the deleted edge is flipped to 0. This takes time $O(1)$ because it involves a lookup and a single bitwise change. The invariant that w has 1s for all edges in the cluster and 0s otherwise is also maintained.

B.3 Microquery

Bitwise logic allows microquery to also operate in time $O(1)$. For two vertices in the microtree v_1 and v_2, microquery can be implemented by first taking the bitwise exlusive or of the two vertices' words. $p = v_1.w \oplus v_2.w$ has bits on for each of the edges in the path between v_1 and v_2, and only those bits on. Bits off in both words are off in the result, these are edges not necessary to get between either node and the root. Bits that the two both had on are now also off; these are edges that the two vertices shared on their path to the random root of the cluster but are clearly not necessary to traverse on the direct path between the two. Only bits that one had and the other did not are on; these represent edges on the only path between v_1 and v_2.

Finally, microquery must check to see if the necessary edges still exist in the cluster. The cluster's word w has been keeping track of all edges still in the cluster. This means $\neg w$ represents as 1s all of the edges that don't exist. Therefore $p \wedge \neg w = 0$ is a simple boolean logic formula that is true if the vertices are connected and false if not. If any of the 1s in $\neg w$, representing edges which don't exist, align with any of the 1s in p, representing the edges needed to make the path, the result of their bitwise and will not be 0 and the statement will be false; otherwise, all the needed edges are there and the statement is true.

Because microquery only makes use of table lookups and bitwise logic, it is also $O(1)$ as required.

References

[1] Alstrup, Stephen, Jens Peter Secher, and Maz Spork. ”Optimal on-line decremental connectivity in trees.” Information Processing Letters 64.4 (1997): 161-164.

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

[3] Thorup, Mikkel. ”Decremental dynamic connectivity.” Journal of Algorithms 33.2 (1999): 229-243.

[4] Frederickson, Greg N. ”Ambivalent Data Structures for Dynamic 2-edge-connectivity and k smallest spanning trees.” Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on. IEEE, 1991.

[5] Henzinger, Monika Rauch, and Valerie King. ”Randomized dynamic graph algorithms with polylogarithmic time per operation.” Proceedings of the twenty-seventh annual ACM symposium on Theory of computing. ACM, 1995.

[6] Gabow, Harold N., and Robert Endre Tarjan. ”A linear-time algorithm for a special case of disjoint set union.” Journal of computer and system sciences 30.2 (1985): 209-221.

[7] Eppstein, David, et al. ”Sparsificationa technique for speeding up dynamic graph algorithms.” Journal of the ACM (JACM) 44.5 (1997): 669-696.

[8] Alberts, David, Giuseppe Cattaneo, and Giuseppe F. Italiano. ”An empirical study of dynamic graph algorithms.” Journal of Experimental Algorithmics (JEA) 2 (1997): 5.