DYNAMIC GRAPH CONNECTIVITY

A PREPRINT

Daniel Lee
daniel.lee@stanford.edu

Adam Pahlavan
adampah@stanford.edu

Sumer Sao
sumersao@stanford.edu

June 3, 2019

Keywords Dynamic Graph Connectivity · Hierarchical Tree Structure · Amortized vs. Randomized

1 Introduction

The goal in the dynamic graph connectivity problem is to answer queries of whether a path exists between two vertices in a graph with a fixed set of vertices but a dynamic set of edges. This problem has been the subject of great theoretical focus and has applications in many fields such as computational biology [1, 2, 3, 4]. To formally define the problem, we are given an undirected graph $G = (V, E)$, with $|V| = n$. We would like to support the following three operations:

- *IsConnected(a, b):* Given two vertices in the graph a and b, determine if there a path in G between these two vertices.
- *Insert(e):* Insert the edge e into G.
- *Delete(e):* Delete the edge e from G.

In this paper, we explore two landmark papers by Holm [5] and Kapron [6] that tackle this problem. Both support *IsConnected(a,b)* in $O(\frac{\log n}{\log \log n})$ time. The first uses a deterministic solution to support *Insert(e)* and *Delete(e)* in amortized $O(\log^2 n)$ time. The second uses a randomized solution to support *Insert(e)* in $O(\log^4 n)$ time, and *Delete(e)* in $O(\log^5 n)$ time.

The organization of the paper is as follows. In section 2, we introduce Euler tour trees, an important data structure for representing trees that is used both in the amortized and randomized solutions we study. In sections 3 and 4, we provide our own interpretation and explanation of the two landmark papers we studied. Embedded into our explanation, we hope to clearly describe some of the similarities and differences between the approaches in these two papers, as well as intuition for what motivated some of the novel approaches taken in each paper. Finally, in section 5, we explain our interesting component, which focuses on the comparing the hierarchical tree structures which comprise the backbone of each of these two data structures. Section 6 finishes with some concluding remarks on the isometries between these two data structures.

2 Important Primitive: Euler Tour Trees

Almost all solutions to the general dynamic connectivity problem involve maintaining some variant of a spanning forest to represent the current state of a graph G. The intuition for why is because a spanning forest is, in some sense, the least complex data structure that allows one to easily answer queries of whether two vertices are connected by some path in a dynamic graph where edges are being inserted and deleted. Therefore, one of the major questions that needs to be answered when tackling the dynamic graph connectivity problem is how to efficiently represent trees. In particular, we want some data structure that can efficiently support the following three operations:

1. `link(u,v):` Merging two separate trees (containing u and v respectively) into a single tree by using (u, v) to connect the two trees
2. `cut(e):` Removing an edge to split a tree into two new trees.

3. `findRoot(u)`: Find the root of the tree which contains u.

Both of the papers we studied [5, 6] tackle this problem of efficiently representing dynamic trees with the Euler tour tree (ET tree). An Euler tour of a tree is a path through the tree that begins and ends at some vertex root, and traverses each edge exactly twice. Since each edge is traversed exactly twice, each edge appears twice in the Euler tour (Fig. 1).

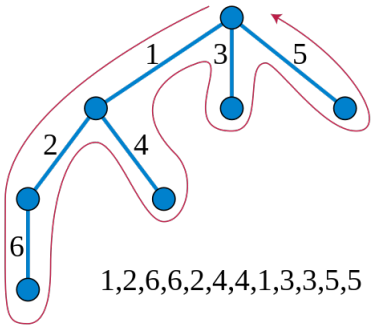

Figure 1: Simple tree and its Euler tour. Image taken from Wikipedia.

The Euler tour is an absolute workhorse of a data structure because it only takes linear space in the size of tree to store, and it can support the *link* and *split* operations in a constant number of links and cuts. The paths are stored as a series of edges in B-trees with $b = \Theta(\log n)$. However, the underlying mechanics for how links and cuts are performed in Euler tour trees are not the focus of this paper. Instead, we will invoke the Euler tour tree as a black box for tree storage for the rest of the project. All operations we need can be supported in $O(\frac{\log^2 n}{\log \log n})$. Additionally, `findRoot` takes $O(\frac{\log n}{\log \log n})$ to traverse up the height of the tree.

3 Solution 1: Dynamic Connectivity in Amortized $O(\log^2 n)$

In this section, we will examine the amortized $O(\log^2 n)$ solution to dynamic graph connectivity proposed by Holm et al. in 2001 [5]. This paper was published over a decade before the other paper we study by Kapron et al. in 2013 [5].

3.1 High-level Overview

As with most other solutions to dynamic graph connectivity, we maintain a maximum spanning forest F for the graph G using a forest of Euler-Tour trees, where each spanning tree represents a connected component in the graph. Both F and G contain all nodes, let's call the subset of edges in F tree edges, and all other edges not in F but in G we'll call auxiliary edges. Notice that, since F is a spanning forest, any auxiliary edge will connect two nodes in the same tree in F.

`Connected`(u, v) simply compare `FindRoot`(u) and `FindRoot`(v). If u and v share the same root, then they are in the same tree, so they are in the same connected component of G. If u and v don't share a root, they must be in different trees, so they are not connected.

`Insert`(u, v) We have 2 cases.

1. If `Connected`(u, v), then they are in the same tree, so we don't need to change F to maintain the invariant that F is a maximum spanning forest.
2. If not `Connected`(u, v), then they are in different trees T_u and T_v respectively, so we must use (u, v) to link the T_u and T_v.

`Delete`(u, v) is the difficult operation. Again, we have 2 cases.

1. If (u, v) is an auxiliary edge, then we don't need to alter F.
2. Else (u, v) is a tree edge. Denote T as the Euler Tour tree containing u and v. After cutting (u, v), let T_u and T_v correspond to the trees containing nodes u and v respectively. The cut-set of (u, v) is the set of all auxiliary edges linking T_u and T_v. For `Delete` to maintain the invariant that F is a maximum spanning forest for G, we must search for an edge in the cut-set of (u, v). If the cut-set of (u, v) is empty, we don't link T_u and T_v because they are in separate connected components in G. Otherwise, we use any edge in the cut-set to link T_u and T_v, because they are part of the same connected component of G.

The challenging operation is searching for an edge in the cut-set after deleting a tree edge.

3.2 Building Intuition

When we want to query the cut-set of (u, v), we must find all auxiliary edges with an endpoint in T_u, and check if the other endpoint is in T_v. So we want some efficient way to store and iterate through edges incident on a tree. We can do this by storing a boolean $hasIncident$ at each node in the tree, denoting whether the subtree rooted at that node contains any nodes with incident edges. To find an incident edge, start from the root and follow the path of booleans to the leftmost node where $hasIncident$ is true, then iterate through its auxiliary edges. To find the next incident edge, we do an in-order traversal, skipping over subtrees where $hasIncident$ is false. Using this augmentation, it costs $O(\log n)$ to find each next incident edge for a given tree.

Naively using this method to scan over all incident edges each time we want to find an edge in the cut-set could take $O(m \log n)$. Obviously we could also answer the query in $O(m)$ instead by just looking through all edges. So we would prefer some book-keeping technique to keep track of work we've already done so we can avoid repeating this work. This book-keeping can maintain some notion of connectivity within the graph. We would also like some way to bound how much total work we must do in a series with many `delete` operations.

Let's first play with the idea of associating a counter with each edge to keep track of every time we check if that edge is in a cut-set. To make things simpler later, let's also assume that we only increment the counter if we find that the edge is not in the cut-set. If the edge is in the cut-set we don't increment the counter. We want 2 things from this counter - we want to upper bound the number of times it can increase so we avoid re-scanning this edge, and **we want this counter to give us useful information about the graph's local connectivity.**

How should we think about a graph's local connectivity? Well, an edge's counter is increased only if we find that it is an internal edge of some tree (it is a tree edge or an auxiliary edge with both endpoints in the same tree). So, for an edge (u, v), a higher counter can correspond to some property of the tree containing u and v. Now the question is - what information about the tree is useful to avoid repeating work?

How about the size of the tree? Notice that, when we were looking for edges in the cut-set of (u, v), we arbitrarily decided to iterate through all edges incident to T_u and check if that edge also has an endpoint in T_v. But we just as easily could have looked through edges incident to T_v checking if they have an endpoint in T_u. What if, for any delete operation, we always iterate through the edges incident on the smaller subtree? Then we will have higher counters associated with edges that have endpoints belonging to smaller trees. **Then an edge with a higher counter means we've learned its endpoints are closer together in** F because at some point they were part of a small tree.

3.3 Edge Levels

Let's put these ideas together. For every edge e, let $level(e)$ denote its counter. We want some way to differentiate between sets of edges with endpoints that are "close" together, so we'll define F_i to be the sub-forest of F including all tree edges e with $level(e) \geq i$. Similarly, G_i is the subgraph of G including all edges at level $\geq i$. Let's keep a forest of Euler Tour trees to represent each F_i. We will maintain the invariant

F is a maximum spanning forest of G. And F_i is a maximum spanning forest for G_i. So if (u, v) is an auxiliary edge, u and v are connected in all forests $F_0 \dots F_{level(u,v)}$. And $F = F_0 \supseteq F_1 \supseteq \dots \supseteq F_{level_{max}}$.

We can utilize the idea that high levels correspond to "close" endpoints. Realize that, when we pick the smaller subtree to iterate over incident edges, this smaller subtree is at most half the original tree's size. So let's maintain the invariant

the maximum size (number of nodes) of a tree in F_i is $\lfloor \frac{n}{2^i} \rfloor$. Thus $level_{max} = \lfloor \log n \rfloor$ because at $level_{max}$ the maximum tree size is a single node.

We insert edges starting at level 0 - this is a new edge we haven't scanned yet, so we don't have much information about its endpoints. The maximum size of a tree in F_0 is n so we maintain our invariants. `Connected`(u, v) and `Insert`(u, v) remain the same, because $F = F_0$

`Delete`(u, v): If (u, v) is an auxiliary edge, then we don't need to alter F. Do nothing.

1. Else (u, v) is a tree edge. We must remove (u, v) from every forest level containing it, so we delete (u, v) from $F_0 \dots F_{level(u,v)}$. Now we want find an edge in the cut-set possibly linking T_u and T_v. Since $F_{level(u,v)}$ was a minimum spanning forest before deleting (u, v) and $F_{level(u,v)} \supseteq F_{level(u,v)+1} \supseteq \dots \supseteq F_{level_{max}}$, so we know any replacement edge can only exist at a level less than or equal to $level(u, v)$. To maintain the

property that $F_{level(u,v)}$ is a maximum spanning forest for $G_{level(u,v)}$ we start looking for replacement edges at $level(u, v)$.

2. for level $i = level(u, v) \dots 0$:
   (a) For the forest F_i, let T_u be the tree in containing u and T_v be the tree containing v. WLOG say $|T_u| \leq |T_v|$. The original tree $T = \{T_u, (u, v), T_v\}$, where $|T| \leq \lfloor \frac{n}{2^i} \rfloor$. So $|T_u| \leq \frac{|T|}{2} \leq \lfloor \frac{n}{2^{i+1}} \rfloor$. This means we can increment the level of any edge in T_u and still maintain our invariant bounding the size of trees on level $i + 1$.
   (b) Iterate through each level i incident edge (x, y) endpoint x in T_u
      i. If y is in T_v, we have found an edge in the cut-set. So link T_x and T_y in forests $F_0 \dots F_i$ - this ensures we maintain the invariant that each F_i is a maximum spanning forest for G_i.
      ii. Otherwise, we know $|T_u| \leq \frac{|T|}{2} \leq \lfloor \frac{n}{2^{i+1}} \rfloor$, so we can increment $level(x, y)$++ to "remember" this property of the graph to avoid repeating work.

3.4 Behind the Scenes

You may have realized we assumed that we can easily determine the size of T_u and T_v. To accomplish this, simply augment each Euler Tour tree node with the size of its subtree. This is also simple to maintain in $O(\log n)$.

We also assumed that we can iterate through each level i incident edge to a given tree. We already mentioned a way to augment the tree to iterate through incident edges. So simply repeat this idea for all forests F_i such that the $hasIncident$ for a node u denotes whether there exist any nodes with level i incident edges in the subtree rooted at u. Thus, we have a method to iterate through level i incident edges for trees in F_i. Maintaining the booleans is a simple $O(\log n)$ operation on inserts/deletes.

So now you're probably wondering - this idea of edge levels seems promising, but what's the runtime? The insert operation itself takes $O(\log n)$, but we know the edge's level can be incremented $O(\log n)$ times. As we just mentioned, each time we increment an edge's level, we did $O(\log n)$ work to find that incident edge and perform the update. So we can make `insert` be amortized $O(\log^2 n)$ to put down credit which pays for future edge level increments.

If `delete` removes an auxiliary edge, this is cheap and simple - it takes just $O(\log n)$. But when a tree edge e is deleted, we first remove it from $O(\log n)$ forests $F_0 \dots F_{level(e)}$. Removing an edge from a single forest takes $O(\log n)$ so this is an upfront cost of $O(\log^2 n)$. Then, we search the cut-set and iterate through incident edges, but this work is payed for by the credit placed down by `insert`. If we find a replacement edge on level i, we link trees in $O(\log n)$ forests $F_0 \dots F_i$. Each link costs $O(\log n)$, so this costs another $O(\log^2 n)$. Overall, the cost of `delete` is $O(\log^2 n)$

This seems fast! But how much space do we use? For each vertex in F_i, we must associate a set of incident level i edges, and we also distinguish tree edges from auxiliary edges. An auxiliary edge of level i must only be stored in F_i, so storing auxiliary edges takes $O(m)$ where m is the number of edges in G. Each tree edge with level i is a tree edge in $F_0 \dots F_i$, so each tree edge appears $O(\log n)$ times. And we have $O(n)$ tree edges (because this is a spanning *tree*). So tree edges take $O(n \log n)$ space. We also have $O(\log n)$ forests, each of which has $O(n)$ nodes. So our overall space complexity is $O(m + n \log n)$.

3.5 Where's the secret sauce?

This data structure achieves such impressive amortized runtime because it is excellent at re-using work we've already done. Think about how we do this - rather than checking all incident edges to T_u to find a replacement edge in the cut-set, we only consider incident edges with levels less than or equal to the level of the edge we just cut. Why?

Let's prove this by contradiction. We are cutting edge (u, v). Assume that a replacement edge $e = (x, y)$ exists, where $level(e) > level(u, v)$. Since e is a replacement edge, that means WLOG, x must be in T_u and y must be in T_v. So, in forest $F_{level(e)}$, u and v must be connected by edges with level at least $level(e)$. And since $F_{level(e)} \subseteq F_{level(u,v)}$ then these edges must exist in $F_{level(u,v)}$. But this means that, before cutting (u, v), u and v were connected by two different paths: (u, v) and this other path of edges with level greater than $level(u, v)$. This means that $F_{level(u,v)}$ had a cycle. We have reached a contradiction. A replacement edge cannot have a level greater than that of the edge we are cutting.

More intuitively, we increase edges' levels when we find that their endpoints are part of small trees. So when we cut an edge with some level l from some tree T, we don't consider any incident edges with level $> l$ because these edges are part of some small internal subtree of T that couldn't have spanned the cut-set.

4 Solution 2: Randomized Worst-Case Connectivity

Although the $O(\log^2 n)$ result of Holm is very fast, it is an amortized solution, and can be as bad as $O(m)$ in the worst case. In this section, we will discuss the fast worst-case data structure invented by Kapron in 2013 [6].

Prior to 2013, the state of the art solution was developed by Frederickson [7] in 1985 and was poly-logarithmic amortized, but with worst case run-time $O(\sqrt{n})$. For a while, it was an open problem whether there was a solution to the dynamic graph connectivity problem that was poly-logarithmic in the worst case. In the words of Patrascu and Thorup in 2007, it was "perhaps the most fundamental challenge in dynamic graph algorithms" of the day [8].

There are many reasons to be interested in a non-amortized worst-case solution. For one, the emergency planning question is a problem in graph theory where the goal is to preprocess a graph so that after a single batch of updates, connectivity queries can be answered quickly. Any amortized solution to the dynamic graph connectivity problem with a poor worst-case bound does not do well on the emergency planning problem. Alternatively, any algorithm with a good worst-case bound does perform well.

In 2013, Kapron et al. developed a new randomized approach which was polylogarithmic in the worst case [6]. In particular, the algorithm provides a worst case $O(\log^4 n)$ insertion, $O(\log^5 n)$ deletion, and $O(\log n/(\log \log n))$ query time. The price we pay is that the query responses may with small error be incorrect. In particular, the algorithm is always correct if it returns “yes” to a query, and is right with high probability if it returns “no” to a query. The novel approach Kapron employs to achieve these novel worst case bounds is the use of Monte Carlo methods to approach the dynamic graph connectivity problem.

4.1 Cut-set structure

Although it does not fully solve the problem with the above-stated correctness and time bounds, the cut-set structure is at the heart of Kapron’s contribution. This cut-set structure solves the problem of quickly finding a replacement edge between two trees if one exists when deleting an edge between two trees in the graph.

4.1.1 The XOR cut-set representation

Like many earlier manifestations of dynamic graph connectivity data structures, Kapron maintains a spanning forest for G using a forest of Euler Tour Trees, with one tree for each connected component. Kapron approaches queries and insertions in the same way as Holm. That is, on queries, we just need to return whether two vertices are in the same tree in our spanning forest. And on insertions, if the inserted edge is a new tree edge, we add it to our spanning forest. Otherwise, it is an auxiliary edge, so we don’t modify our spanning forest.

Just as in Holm, the difficult problem in this data structure is finding replacement edges on tree edge deletion. That is, when two trees are separated, how can we efficiently check if there is another edge that connects these two trees. Holm solves this problem by amortizing the cost of finding replacement edges over many deletions, and carefully keeping track of how deeply nested a replacement edge is within its tree through the level abstraction. Kapron uses a completely different (and much simpler approach). The basic insight is that we don’t need to keep track of all possible replacement edges between two trees. In fact, we only need to know one edge in the cut-set to properly reconnect two trees. Therefore, if we can construct a representation of the graph that can efficiently find single replacement edges when the number of possible replacement edges is both small and large, we can efficiently tackle the problem for all graphs.

Here’s the basic setup. If our graph has n vertices, then we will name each vertex with its binary index representation, which is $\log n$ bits long. We will name each edge e as $v_1|v_2$, the binary concatenation of labels for vertices v_1 and v_2. Notice that $v_1|v_2$ is a length $2\log n$ binary string, and for consistency, we will maintain that vertices are ordered in the label so that $v_1 < v_2$.

Each vertex v_i in F will keep a label of length $2\log n$ which is the bitwise XOR of all incident edges to v_i. To be clear, we include all incident edges that are both in the current spanning forest and not in this XOR’d result. For each tree T in F, we will label T with the bitwise XOR of the labels of all vertices in T.

$label(v_k)$ = XOR(v_i-v_j for all incident edge labels)
$label(T_k)$ = XOR($label(v_i)$ for all v_i in T_k)

It may not be immediately obvious how this XOR idea is useful. Consider the case where we have a single tree in our graph, and we are removing a tree edge. We would like to find a replacement edge between these two now separate trees T_1 and T_2. If we examine our split tree labels $label(T_1)$ and $label(T_2)$, we may find some useful information. If there are 0 edges in the cut-set between these two trees, then all edges are either entirely contained in T_1 or T_2. This

means that both $label(T_1)$ and $label(T_2)$ will be 0, since every edge is counted twice in each XOR calculation. So, if $label(T_1) = label(T_2) = 0$, that is a signal that there is no replacement edge between these two split trees, so we must separate the original tree into 2 different ones.

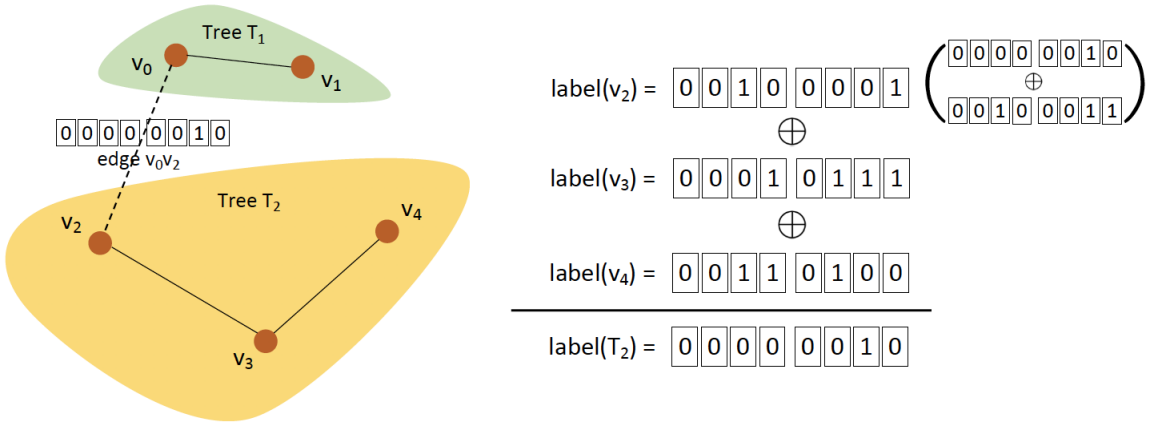

Figure 2: Process of finding replacement edge v_0v_1 between trees T_1 and T_2 when all edges are included in the cut-set structure. Since there is only one replacement edge, the labels of trees T_1 and T_2 give us this edge's name.

Imagine a second case where there is exactly one edge in the cut-set between T_1 and T_2. Then, both $label(T_1)$ and $label(T_2)$ will contain the name of this edge. This will give us a replacement edge directly, so we also have succeeded in this case (Fig. 2). The third and final case to consider is if there are multiple edges in the cut-set. This is where our simple algorithm fails, as now $label(T_1)$ and $label(T_2)$ contain the XOR of multiple edge names, which can not be easily separated into their individual edges. So in this case, although there are many possible replacement edges, we can't efficiently identify one. We will need to modify this structure to also work with cases where the number of possible replacement edges is large.

4.1.2 Randomization to the Rescue

It seems as if our XOR-based cut-set structure handles cut-sets of size 1 and 0 really well, but not any larger cut-sets. So, in order to handle the case of larger cut-sets well, we use randomization. Recall earlier the insight that we only need to find one possible replacement edge in a cut-set, even if the size of the cut-set is potentially large. That will be the driving intuition behind this approach.

The basic approach is to use Monte Carlo simulation and probabilistic amplification to try to simultaneously handle the cases where the number of replacement edges is small and large. We do this by running multiple parallel instances of the cut-set XOR data structure above. We will use $\lceil 2 \log n \rceil$ levels, and in each level $i = 1, 2, ...\lceil 2 \log n \rceil$, each edge will have a $1/2^i$ probability of being included in our cut-set data structure. When we want to find a replacement edge, we will query each parallel cut-set structure for a replacement edge, and return one if any of our cut-set structures succeed. Thus, it takes $O(\log^2 n)$ time to search a cut-set.

The intuition for why this works is that for large cut-sets, the large levels will have a good chance of only having one edge in their cut-set, and for small cut-sets, the lower levels will have a good chance of finding a single replacement edge. Note that we only require $O(\log n)$ levels, so we can run this parallel structure while staying in poly-logarithmic bounds.

As a side note, to see where exactly the $\lceil 2 \log n \rceil$ levels come from, a loose bound on the number of edges in a cut-set is n^2, as n^2 is the maximum number of edges in a graph with n vertices. Therefore, a loose bound on the number of levels we will need to be logarithmic in the size of the cut-set is $\lceil 2 \log n \rceil$.

We will now show that this level scheme does a good job of finding a single replacement edge across all cut-set sizes. Let say our cut-set has k edges. We will show that the level $\lfloor \log k \rfloor$ has at least a constant probability of having exactly one replacement edge, as desired. This is isomorphic to the problem of showing that for k coinflips of a biased coin with $p = 1/2^{\lfloor \log k \rfloor}$, exactly one coin will show up heads. The exact probability of this occuring is

$$P(\text{success}) = \binom{k}{1} \left(\frac{1}{2^{\lfloor \log k \rfloor}}\right) \left(1 - \frac{1}{2^{\lfloor \log k \rfloor}}\right)^{k-1} > \left(1 - \frac{1}{2^{\lfloor \log k \rfloor}}\right)^{2^{\lfloor \log k \rfloor + 1}}$$

The inequality follows from the fact that the product of the first two terms on the left hand side is ≥ 1 and the exponent $2^{\lfloor \log k \rfloor + 1} \geq k$. Now, for $k = 1$, our cut-set is of size 1, so level 0 will always succeed. For $k = 2, 3$, the left hand side is greater than $\frac{1}{9}$. For $k \geq 4$, the right hand side is greater than $\frac{1}{9}$ (In fact, the right hand side approaches $\frac{1}{e^2}$. This follows from the result that as n grows large $(1 - \frac{1}{n})^n \approx \frac{1}{e}$).

Now, while only incurring an extra logarithmic cost, we can with constant probability find one edge no matter how many edges are in the cut-set. This idea is really beautiful!

4.1.3 Amplification of Independent Randomized Structures

The next step is a common one in randomized data structures. Once we bound the error of a randomized data structure to be negligible, we can often run multiple independent copies of that data structure in parallel to make the error negligible in our complexity parameter. That is exactly the strategy applied here. In particular, we will take the parallel data structure described so far and parallelize it across an additional dimension.

We will run $c' \log n$ copies of the original parallel data structure in parallel, incurring again only another logarithmic hit to our cost. Let's see where that takes us. We previously showed that the chance of finding a replacement edge in a cut-set assuming one exists is $\geq 1/9$. Therefore, the probability of failure among all $c' \log n$ copies is $\leq (1 - \frac{1}{9})^{c' \log n}$. For $c' = -c\frac{1}{\log(\frac{8}{9})}$ this becomes

$$\leq (1 - \frac{1}{9})^{-c\frac{1}{\log(\frac{8}{9})} \log n} = 2^{-c \log(\frac{8}{9}) \frac{1}{\log(\frac{8}{9})} \log n} = n^{-c}$$

Therefore, we can shrink the probability of not finding a replacement edge in the cut-set to n^{-c}, with only a $\log n$ hit to complexity.

4.1.4 Implementation Details

We have shown above that the probability of failure for a single cut is $< \frac{1}{n^c}$. If we perform a series of t edge updates, since each update touches two trees, we can use a loose union bound to bound the total probability of some error as $< \frac{2t}{n^c}$.

It is somewhat involved to maintain the labels of every ET tree for these $\log^2 n$ parallel simulations efficiently, especially since the trees can change often throughout the course of insertions and deletions. As mentioned earlier, this is not the focus of our paper, so for brevity, we choose to omit these details. What's important to note is that the parallel approach above only incurs a total poly-logarithmic $\log^2 n$ extra cost to all our operations. Accounting for the ET tree bookkeeping, the final cost is $O(m \log^2 n + n \log^3 n)$ for preprocessing, and $O(\log^3 n)$ for each edge update. The curious reader can refer to the appendix for pseudocode of inserts and deletes in Kapron's final algorithm.

4.1.5 Cool! Are we done yet? Nope!

We have a problem. In section 4.1.2, we built off of the assumption that the spanning forest maintained by the cut-set was independent of the randomness within the cut-set. This is how we arrived at the $\frac{1}{2^i}$ probability of each edge being included in the XOR for level i. While this is the case when we only insert edges, this is not true when we start to delete edges. But why?

At first, when we just insert edges, we maintain this $\frac{1}{2^i}$ because the cut-set makes random decisions whether to add the edge to level i. When we first delete an edge, the structure of the spanning forest determines the potential replacement edges, then the randomness within the cut-set picks one of these replacement edges. This replacement edge then determines the new structure of the spanning forest. For this first deletion, all of our probability assumptions were correct. However, now that we've performed this deletion, the randomness within the cut-set has now affected the structure of the spanning forest we are maintaining. For any future deletions, we can no longer guarantee that the $\frac{1}{2^i}$ probability an edge is included in the XOR for level i because the past delete operation has "tainted" the randomness.

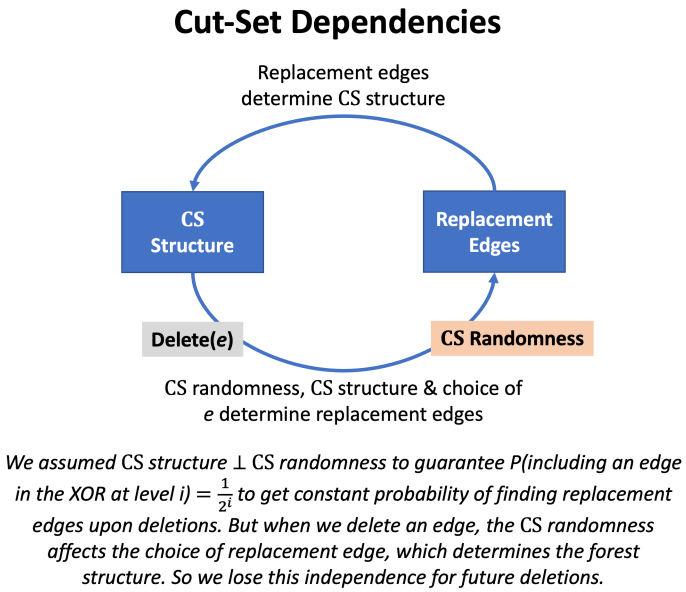

Figure 3: For a given cut-set, we refer to CS structure as the spanning forest we maintain for G, in other words, whether an edge is a tree or auxiliary edge.

This point is particularly clear when we delete replacement edges. When an edge is deleted, its value is XORed out of each row and copy of the cut-set. Then, the cut-set searches for a row and copy that has a single edge value in it to choose as a replacement edge. Now, let's say we perform a delete again, this time on the edge that was just chosen to be the replacement edge and elevated from an auxiliary to a tree edge. Well, its value is now going to be XORed out of our tables.

If we keep deleting the replacement edge in this manner, our current data structure in fact fails after $O(\log^2 n)$ deletions. This bound comes from the fact that there are $O(\log^2 n)$ rows in the amplified cut-set structure. In the best case, every row can contribute one replacement edge – and after the replacement edge is chosen, it is deleted on the next turn. Once a row is zeroed out, it can't have a value added again since we are not inserting any edges in this case. Thus, since every row can contribute 1 replacement edges, and there are $O(\log^2 n)$ rows, our data structure will be able to find $O(\log^2 n)$ replacement edges before failing. Thus, in a graph where there are over $O(\log^2 n)$ edges connecting two trees that were just disconnected, our data structure could fail under this sequence of deletes. This is an example of how sequences of deletes can drive the probability an edge is included down from $\frac{1}{2^i}$ to 0 to destroy our correctness.

Essentially, in order to maintain our probability assumptions and protect our cut-set's randomness from adversarial updates, we must dissociate the cut-set's structure from its randomness.

4.2 A Boruvka Tree of Cut-Sets

We will tackle the cut-set issue with a hierarchical tree structure similar to the Boruvka tree to achieve the probabilistic accuracy desired. This structure achieves $O(\log^4 n)$ insert, $O(\log^5 n)$ delete, and $O(\frac{\log n}{\log \log n})$ isConnected bounds.

4.2.1 High Level Overview

To fix this problem, Kapron took inspiration from Boruvka's algorithm which creates a hierarchical tree structure to construct minimum spanning trees. The structure is as follows. Consider building up a set of tiers from tier 0 to tier $\lceil \log n \rceil$. Each tier will represent some spanning forest of the graph. Tier 0 will be a forest of all nodes. Tier j is created by connecting each tree in tier $j-1$ to a neighboring tree if possible, discarding repeated edges.

The details of this with respect to our data structure are as follows.

Let $top = \lceil \log n \rceil$. At each level l from 0 to top, we maintain a cut-set structure for G with a spanning forest denoted F_l. Each of these cut-set structures have their own random bits associated with them as described in section 4.1.2. So we have essentially $O(\log n)$ independent copies of cut-sets to start. These cut-sets will later on decide how the

Boruvka tree structure will change on the $delete$ operation, namely on a level, which edge will be used to join two trees on all of the levels above it.

The way in which we build each F is different than the way the minimum spanning tree algorithm constructs each level, however we do see some similarities in the dependencies between each level.

In our version, each F starts out with the same structure. Namely, when we are initializing our structure, we see if an edge connects two unconnected components of F_{top}. If they do, then we add this edge to all levels except for the 0^{th}. We will explain why we do this later. As we use our cut-sets on each level to support updates, we will maintain the following four invariants.

1. The tier 0 forests are the vertices of graph G.

2. Each tree in a forest on level l must have the following properties. Either it is a maximal spanning tree of a component of G, or it is matched. By matched, we mean that there is an edge on level $l+1$ that connects T with some other tree on level l.

3. We also see that regardless of if a tree on level l is matched or maximal, $F_l \subseteq F_{l+1}$, because for any edge included on level l it must be included on levels higher than it.

4. The structure of a forest F_l is independent of the cut-sets on tiers l and higher.

We also can see that because of invariants 2 and 3 there are at least 2^l nodes in each matched tree on level l. This is because in matched tree, we we combine at least 2 trees every subsequent level. Note, that nothing is stopping us from combining more than 2 trees in a level. Our only requirement is that every non maximal tree is matched. So a tree a could be matched to tree b which could be matched to tree c. We see this in figure 2 below: On level F_1, there are trees of size 3. We also can't bound the size of maximal trees per level because if they're maximal it means that they're also maximal in the graph and don't need to grow. These facts imply that the top level spanning forest F_{top} is in fact the maximum spanning forest.

In addition, the fact that we can only give a lower bound for the size of each tree on each level being 2^l implies that we need at least $\log n$ levels in this structure to guarantee that the top level is a maximum spanning forest of G. This is necessary to support $IsConnected$ queries quickly. F_0 containing just the vertices of the graph plays into that, but also is important because of how deletes work. When we discuss deletes more carefully later on, we see that F_0 essentially asks as the base case in our process. Since it contains no edges, no deletes will ever affect it. However, it can still be used to find replacement edges for tiers above it.

4.2.2 Supporting Each Operation

With this overall structure in mind, we can now detail the operations this tree supports.

$IsConnected(x, y)$: We can see that the only way x and y are connected is if they are in the same tree. How can we check this? Well we have a spanning forest of G in the top tier of our Boruvka structure. Thus, all we have to do is see if x and y are in the same tree in this spanning forest. If they are, return "yes" otherwise return "no". With the use of our Euler tour trees we can find what tree each of x and y are in, in $O(\frac{\log n}{\log \log n})$ time.

$Insert(e = \{x, y\})$: First insert this edge into the XOR for each cut-set structure on each tier. The reason why we must add this edge to every cut-set is because this edge is now part of the graph. Since each cut-set is supposed to keep track of edges that exist in cuts in the graph, they must be updated with this information that there is a new edge in the graph.

Additionally, if this edge connects two unconnected trees in the spanning forest on the top level F_{top}, we must add this edge to F_{top}. However, this might violate invariant 2, that every tree on a level l that isn't maximal, must be matched. Let's say that the tree containing x was maximal on levels $l > 0$ prior to inserting e. Then, connecting x on the top level of the Boruvka tree means that all of the trees containing x on levels $0 < l < top$ are now unmatched, since they are no longer maximal, and don't have edges on level $l+1$ connecting them to another tree on level l. Note that the tree containing x on level $top - 1$ is still matched since the edge we just added on the top level connects it to some other tree.

In order to fix this, we simply add e to all levels greater than 0. This scenario is detailed in the following figures below.

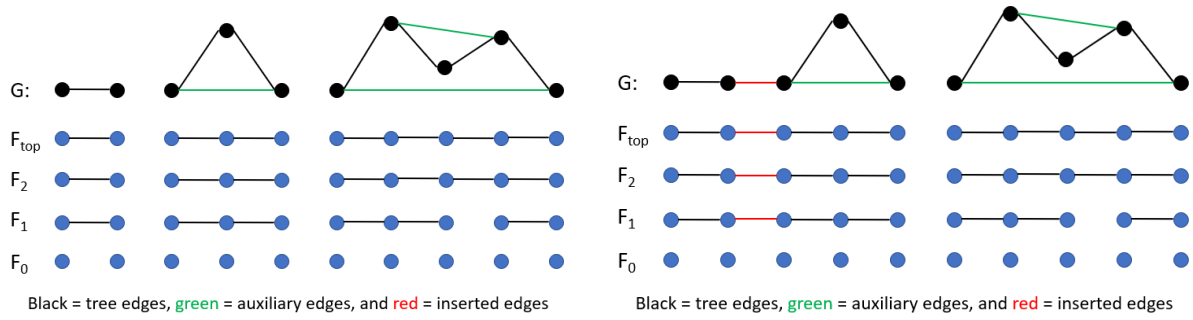

Figure 4: Since this new edge connected two trees in the top layer, the two leftmost trees in level F_1 are now unmatched. Thus, it must be added to all layers greater than 0.

This may not always be necessary, say if a tree only became maximal in level F_8 it isn't necessary to add this edge to levels below it. However, keeping this process consistent is easier and allowed. Inserting this edge to all of the tiers won't create any problems in our lower tiers. If an edge exists in a lower tier, we know it must also exist in our higher tiers, since each spanning forest of level l is a subset of the spanning forest of level $l+1$. So inserting our edge into all spanning forests greater than 0 won't violate this property. We also know we won't create any cycles in any levels because if nodes x and y were already connected somehow in a lower level, they would not be disconnected in the top most level.

Note when constructing our structure, we are simply calling insert on each edge in the graph. This is why the spanning forests on each level begin with the same structure.

$Delete(e = \{x, y\})$: Per usual, this is the most difficult step. What we know we must do is maintain the invariants described before. The hardest to do are to ensure that every nonmaximal tree on a level is matched, and to ensure that the structure of a forest on level l is independent of the cut-sets on tiers l and higher.

The first thing to do is to remove $\{x, y\}$ from the XOR of each cut-set structure that contain it, in any tier. Next, we remove this edge from all F structures that contain it. If we do end up finding an F that contains it, removing it might violate the property that every nonmaximal node has a sibling, namely the trees containing x or y. We run the following process to fix this.

First, we begin at the lowest ancestor of x in the Boruvka tree such that the tree containing x is unmatched on it's tier. To do this, we don't actually have to do any searching. Let's say that the lowest tier from which $\{x, y\}$ is deleted from F is l. The only tree that becomes unmatched is the tree containing x on level $l-1$. Note, that $l > 0$ since F_0 contains no edges. We then call a $reconnect$ subroutine detailed below on the tree containing x on level $l-1$. This $reconnect$ guarantees that the tree containing x on level $l-1$ is no longer unmatched. However, trees in higher tiers may still be unmatched. Thus, we iterate up through the levels greater than l checking if the tree containing x is unmatched, and fixing it until all of the trees are matched. Then we do the same for the tree containing y.

Note that only the trees containing x and y can possibly become unmatched. This is because we are only ever modifying these trees to begin with and adding edges to other trees, so we will not turn any previously matched tree to unmatched. We might add one of the trees containing x or y to another tree in the reconnect step, but if this tree is unmatched still contains x or y. Thus, we can see that these repairs truly only need to fix local connected components. This is shown clearly in the figures below.

The $reconnect$ subroutine works as follows. Given some tier k, and some tree T, we will find an edge on the $(k+1)^{th}$ level to connect T to so that it is unmatched. To do this, let's search for a replacement edge for T in the cut-set structure on tier k. This will either return an edge or not. If it doesn't, we claim that T is a maximal tree, and doesn't need a match. In the case where it returns an edge, T does need a pair. Let the edge $\{u, v\}$ be the edge returned by this search. If there is already some path containing $\{u, v\}$ in a tier $k' > k+1$, then we remove one of the edges on this path, for all such k'. Then, we simply add this new edge into all tiers $k' \geq k+1$. This removal is necessary, since otherwise we would be creating a cycle in our tree. Note, this removal is done for all tiers $k' \geq k+1$ in which a cycle otherwise would have been induced.

This process is depicted below in the following diagrams. These diagrams have been modified from Kapron's original paper to include the graph being queried.

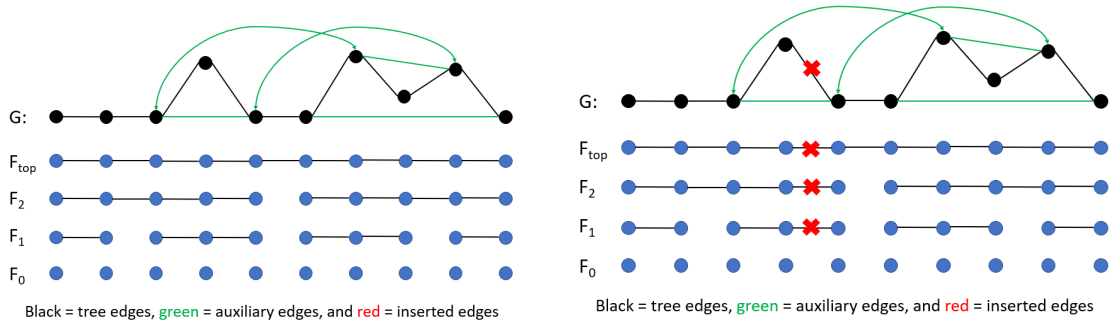

Figure 5: Consider the initial structure on the left. We delete the edges shown on the right.

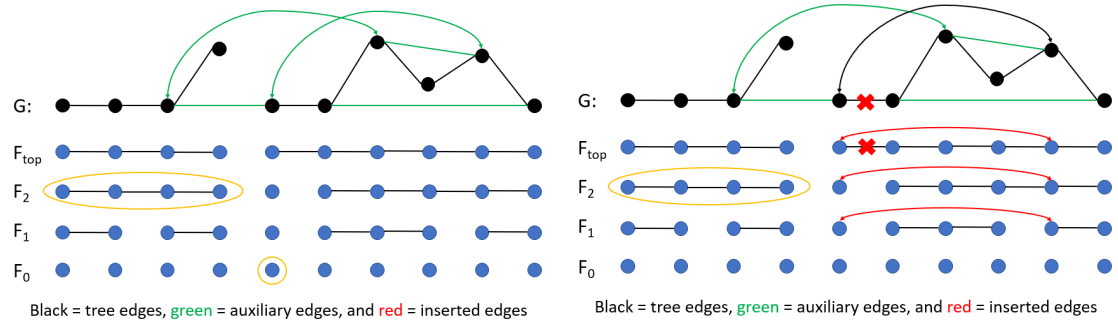

Figure 6: The circled trees are now unmatched. We will first handle the tree on level 0. We query the 0^{th} cut-set and find the red replacement edge. We add it to all levels higher than us, but create a cycle in level F_{top}.

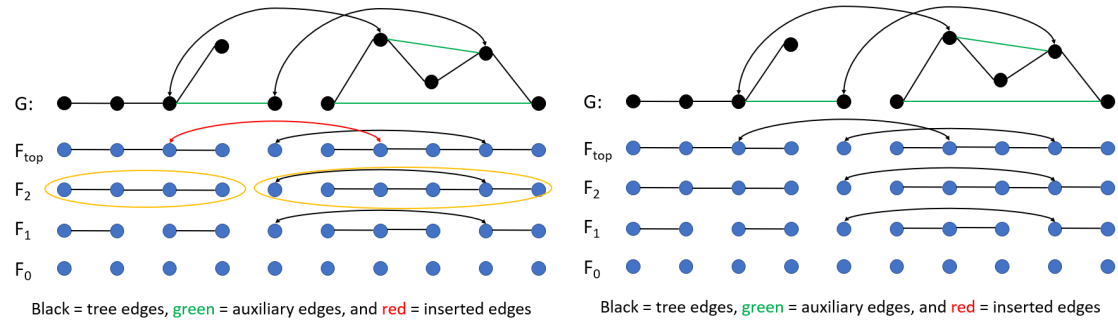

Figure 7: Deleting the cycle edge in F_{top} created another unmatched tree in F_2. We query the 2^{nd} cut-set for a replacement edge. This replacement edge also happens to fix the unmatched tree caused by the other endpoint of the initial node that was deleted so we are done!

4.2.3 Correctness

We will now show that each of these operations don't violate any of the invariants we have stated (refer to page 9 for a refresher). We have discussed the details of why we need each of these invariants, except for why the structure of a forest F_l is independent of the cut-sets on tiers l and higher. Since the explanation of why that invariant is needed is quite detailed, we leave it to the next section, but still explain how our procedure maintains it.

Invariant 1: This invariant is not affected by updates, thus it always holds among inserts and deletes.

Inserts: In an insert, if an edge is added to F_{top} it is added to all levels greater than 0. Thus, $F_l \subseteq F_{l+1}$ is still maintained and **Invariant 3** holds. In addition, this edge being added is completely independent of the cut-set's structure, since it is only added to F if the two components being joined are disconnected in G. If these components are disconnected in G, the cut-sets do not include the edge being added anywhere. Thus, the adding this edge to a cut-set structure at any level doesn't make it dependent on any other cutset structure above it, and **Invariant 4** holds. We can also just think of it as, no cutset is used to add this edge to a structure, thus it maintains the invariant.

Inserting an edge cannot cause a non-maximal component to become unmatched, since adding an edge would never disconnect an edge that connected two trees on a lower level. However, inserting an edge can make a previously maximal tree, non-maximal, and thus now unmatched. This scenario was depicted in Figure 4. If a tree on some level below top was previously maximal that means that the same tree shows up in top. Thus, in order for this maximal tree lower down to become non-maximal, the same tree must be the one being joined in level top. But since we add the edge to all levels greater than 0, this lower non-maximal tree is now matched, and **Invariant 2** holds.

Deletes: In a delete, if an edge is made into a tree edge at tier l, it is also made into a tree edge at all tiers above l. If an edge is deleted at some tier k it is also deleted from all tiers below it. Thus, $F_l \subseteq F_{l+1}$ and **Invariant 3** still holds.

As we know, a delete may cause a tree to be unmatched, violating invariant 2. In order to solve this problem, we call reconnect on the lowest level tier with an unmatched tree. More specifically, if edge $\{x, y\}$ is deleted, then only trees T_x and T_y can become unmatched. We also will show that this repeated calling of reconnect will terminate, since after an unmatched tree is matched on tier k, there will only be unmatched trees in tier's higher than k. When we find a replacement edge e on tier $k+1$, we add this edge to all tiers higher than k. Thus, we have fixed the problem on our level, and no other problems may arise below us since we are only adding edges above us. We then can fix any cycle that is formed at a higher tier by edge e being added by deleting another edge on this cycle. Let's say a cycle was formed at tier $k'' > k+2$, then using a similar argument to what was used in section 4.2.2, this causes an unmatched tree to be formed at tier $k+1$ which is greater than tier k. Thus, our lowest unmatched tree only continues to move up, until there are no unmatched trees left. Thus, **Invariant 2** is maintained, and all trees are either matched or maximal.

For **Invariant 4**, we need to show that structure of spanning forest maintained by the cut-set on tier k is only dependent on the randomness of cut-sets on tiers $0 \dots k-1$. First recall that the cutset on tier k is used to find a replacement edge for an unmatched tree on tier k. Let's once again denote this replacement edge e. This e is added to the F structures for all $k' > k$, but not k itself. We know that the cutset on tier k is only dependent on the randomness of the cut-sets in tiers lower than it, and not the tiers higher than it. Thus, e is independent of all of the tiers higher than it, and it is only added to those tiers. Hence, this invariant is still maintained, as the $F_{k'}$ for $k' > k$ have only been affected by k. However, we also conduct a delete operation on an edge that might be part of a cycle. Let's call this edge e' that may have to be removed on some tier $k'' > k+1$. This e' is determined by e, since the choice of e is what determines which edges may or may not become part of a cycle. However, since e and thus e' was chosen by the cutset on tier k, tier k'' is dependent on the cut-set at tier k. This doesn't violate our invariant however, since the structure of $F_{k''}$ is allowed to be dependent on cut-set k, since k is lower than k''. Thus, this deletion of e' also doesn't violate our invariant.

Thus all of the invariants hold by these operations, and the top level structure, F_{top} is a maximum spanning forest of G.

4.3 Proof of Probability and Complexity

We will now show with what probability this data structure succeeds, and how quickly it can handle each operation.

In previous sections we have discussed how delete works at length. The key takeaway is that it will call $reconnect$ at most $2 \cdot top$ number of times. This is because whenever an unmatched tree is reconnected, the next unmatched tree can only be at a higher level. Thus, in the worst case there can be top number of levels and $reconnect$ calls for the tree containing x and the same number of calls for the tree containing y.

If we let $c' > (c+3)\frac{1}{\log(\frac{8}{9})}$ be the constant in each cutset data structure, a call to $delete(e = \{x, y\}$ where e is independent of the values of each cutset maintains the invariants with probability $1 - \frac{1}{n^{c+2}}$. This is because we make $2 \cdot top$ calls to reconnect, and the probability that a single call to reconnect fails is $\frac{1}{n^{c+3}}$, as shown before. Thus, the probability that a call to delete fails is $\frac{2\log n}{n^{c+3}} < \frac{1}{n^{c+2}}$.

Thus, for any sequence of n^2 deletions and any number of insertions, the algorithm maintains the invariants and succeeds with probability $1 - \frac{1}{n^c}$. This is because the probability that any call to delete fails is $\frac{1}{n^{c+2}}$. So for n^2 calls to delete, the probability that the algorithm fails by the union bound is $\frac{n^2}{n^{c+2}} = \frac{1}{n^c}$.

Note that we could have shown this for any number of operations n^d for some constant d, however we chose $d = 2$ since there are at most n^2 edges in a graph at a time.

$Inserts$: This requires inserting an edge into $O(\log n)$ cut-sets. Each insert takes $O(\log^3 n)$ time for a total of a $O(\log^4 n)$ runtime.

$Deletes$: This requires doing at most two $reconnects$ per tier. Each reconnect takes time $O \log^4 n$ due to the linking and cutting required by the ET tree, details we won't go into. Thus, since there are at most $O(\log n)$ calls to reconnect, delete takes a total of $O(\log^5 n)$ time.

4.3.1 Why Does this Work

Let's now try to gain some more intuition as to why this scheme solves the previous problem we had. With the single cut-set, we had the issue that, upon delete operations, the cut-set's randomness determined its structure. This caused the probability whether an edge is included in a given level to no longer be completely random - it became a function of past delete operations. The Boruvka tree fixes this by completely separating, for any given cut-set, the structure of the spanning forest it maintains from the randomness it uses to find replacement edges.

The key insight is that a single cut-set using its randomness to update its own structure creates dependencies in the future. Instead, in the Boruvka tree, a single cut-set cannot affect its own structure, but it uses its randomness to update the structure of higher-tier cut-sets, thus creating dependencies up the height of the tree. All of these dependencies are unidirectional - the structure of any given tier is the result of answers to past cut-set queries and randomness in lower tier cut-sets. Similarly, the randomness of a cut-set on a given tier can only affect the structure of cut-sets on tiers above it. Since the dependencies are unidirectional, the structure of any cut-set spanning forest is independent of its own randomness. We can guarantee that any edge is included in the XOR for level i with probability $\frac{1}{2^i}$ so we now meet our probability bounds for correctness. The following diagram helps to show how this works.

The above reasoning follows from Invariant 4. Invariants 1-3 provide structural guarantees for each spanning forest so that answers to cut-set queries only affect the structure of higher-level cut-sets, while ensuring that the top level always maintains a maximal spanning forest for the graph. Invariants 1 and 3 ensure that we can safely propagate replacement edges up the graph and always fix unmatched trees starting from the lowest level where such an unmatched tree exists. Invariant 2 ensures that the top level will always contain a maximal spanning forest, because it lower bounds the size of non-maximal trees at level i to $\frac{n}{2^i}$. This is because it guarantees that non-maximal trees will be joined at each level.

Overall, the Boruvka tree is a clever way to use cut-sets to find replacement edges while ensuring that we don't introduce cyclic dependencies between any given cut-set's spanning forest structure and its own randomness upon delete operations.

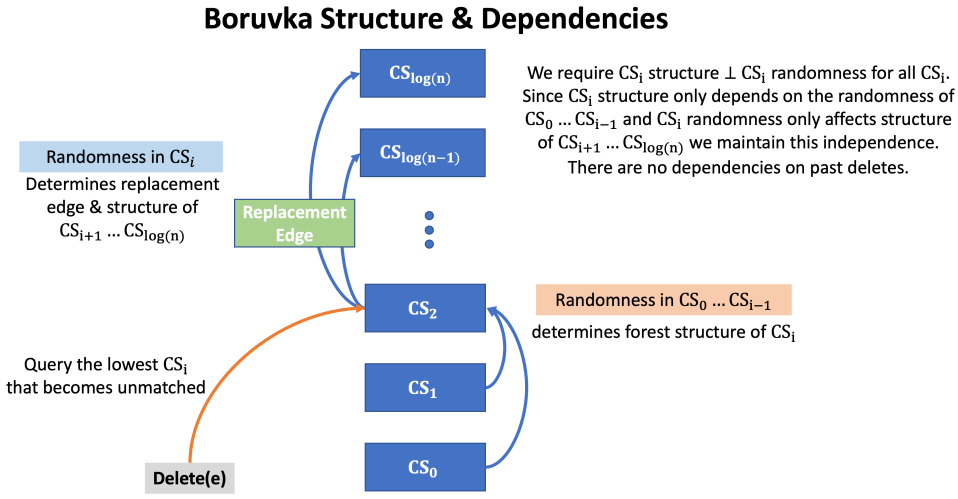

Figure 8: The Boruvka structure dissociates an individual cut-set's structure from its randomness.

4.3.2 Implementation Details

A few notes on how implementing this data structure can be done. The first note is that we don't actually implement the Boruvka tree structure, we just maintain it implicitly. To do this, we represent each cut-set indexed by tier. We also augment an ET tree with the number of nodes in it to check if a tree is unmatched. We simply check to see if the parent has the same size.

In order to implement $reconnect$, we need to have some way of checking for what tier a cycle might form on. If we weight each edge by their tier number, all we need to do is find the maximum weight edge on a path. To do this, we can use link-cut trees which support this exact operation in a dynamic tree in $O(logn)$ time.

We also need to address the fact that we have only shown our probability of correctness holds for n^2 deletes. Thus, we must use a clever trick to allow our data structure to continue to answer queries with the same probability bound if there are more than n^2 deletes.

We first split up the sequence of updates into intervals that contain n^2 deletes (and any arbitrary amount of inserts), and label them from $0, 1, ...k$. We will maintain two copies of our data structure, denoted D_0 and D_1. We will use D_0 to answer the queries in the intervals that are labeled with an even number, and D_1 to answer the queries in the intervals that are labeled with an odd number. In the intervals that a data structure isn't being used, it goes through a rebuild. Let's say we're on the i^{th} interval and WLOG D_0 is being used, then D_1 will be getting itself rebuilt. In the first half of interval i, D_1 will be initialized with the state of the graph at the beginning of interval i. This will take at most $O(n \log^4 n + m \log^3 n)$ time. When $m = n^2$ this becomes $O(n^2 \log^3 n)$ time. Thus, we can fit this rebuild inside the first half of the interval since each operation takes at least $\log^4 n$ time, and there are $O(n^2)$ of these operations in the first half. Then, in the second half of the interval, D_1 will process the queries in interval i at twice the speed as D_0 in order to "catch up." Thus, rebuilding after every n^2 deletes allows for the probability bound to be kept. Note, that we are technically handling $2n^2$ deletes in a row, but this is within a constant, and our probability bound still holds.

5 Hierarchical Tree Structures . . . are pretty interesting

Wow that was a lot! A good portion of our exploration/interesting component went toward explaining each data structure as clearly as possible. We spent a lot of time trying to understand the intricacies of the data structures and details that the authors glossed over or were unclear about. We hope our explanations so far have been helpful. Now we'll take a step back and admire the magic of hierarchical tree structures (HTS) in each of these two solutions to dynamic graph connectivity.

It's amazing how many similarities these HTS share, given that their purposes and functionality are so different. Just to make comparison easier, I'm going to refer to Holm's F_0 as the highest level and $F_{level_{max}}$ as the lowest level. Both Holm and Kapron use hierarchical tree structure where edges have associated levels used to query the cut-set. Both Holm's amortized HTS and Kapron's Boruvka tree store a maximum spanning forest for the graph at the highest level and store individual nodes at the lowest level. Both are organized such that each level of the forest is a subset of all the higher level forests. In Holm, this is necessary to effectively distinguish which edges to consider as part of the cut-set, whereas, in Kapron, this property allows us to maintain unidirectional dependencies. Both have $\log n$ layers, but for fairly different reasons. Holm's has $\log n$ layers because this is the maximum number of times you can consider an edge as part of the cut-set, since we always search the smaller tree. Kapron's has $\log n$ layers because we need this many layers to guarantee that the highest level will always have a spanning forest, no matter the sequence of insertions and deletions.

In Holm's HTS, there exists an upper bound of $\frac{n}{2^i}$ on the size of trees at level i, but in Kapron's Boruvka tree, there is actually a lower bound of $\frac{n}{2^i}$ on the size of non-maximal trees at level i. This is very interesting. In Holm, this upper bound is necessary to provide information on local connectivity for edges at level i and bound the amount of work we perform. In Kapron, this lower bound arises from the fact that any non-maximal tree on level i must be a proper subset of a tree on level $i + 1$ (so we connect any two non-maximal connected components every level). This is necessary to ensure that all trees on the top level are maximal.

The most interesting difference is how each takes advantage of edge levels. Since Holm's amortized solution aimed to re-utilize AND limit the amount of work we do, his edge levels represented local graph connectivity and reduced the work required by our cut-set queries. The invariants relating edge levels to tree sizes were invaluable for conveying information about the distance of edges' endpoints and limiting the number of times we must consider them as replacement edges.

On the other hand, Kapron's powerful cut-set structure could not make strong correctness guarantees while functioning alone, so he had the ingenuous idea of using levels within the Boruvka tree to redirect dependencies and dissociate a

cut-set's spanning forest structure from its randomness. Here, the level an edge appears is random - it depends on the structure of the graph and the random decisions of cut-set structures at lower levels. In this case, edge levels can be seen as a sort of barrier, guarding a single cut-set's spanning forest from being affected by its own randomness.

In summary, we could described Holm's HTS as a clever bookkeeping scheme, while Kapron's HTS is a way to separate the randomness that answers queries from the randomness that determines the structure. Thus, these two structures are tackling completely different problems even if their structure is similar.

We also found it pretty cool that, since Holm's structure is deterministic, it can support queries to return the path between two nodes, whereas since Kapron's structure is randomized, it cannot support queries to return the path between nodes or else the underlying spanning forest would be revealed, leaving it susceptible to adversarial sequences of deletes.

A small detail that was inquired about in our project proposal was if finding the lowest unmatched tree after deleting an edge was able to be localized, instead of searching the entire tree. As we have discussed, it turns out that it can be localized. The lowest tier on which an edge is deleted can only unmatch the trees that contain both vertices in the edge in the tier below it. If there was a lower tier that became unmatched, this would mean that it's parent tree had the edge that was deleted. But this is a contradiction, since we've claimed that the unmatched trees we've found are the nodes directly below the lowest edge deleted.

We see that this doesn't change the complexity of the delete operation, since we still need to scan up the structure to find any future unmatched trees that are formed. We think this is probably why Kapron et al. ommitted this fact, to simplify the explanation.

Another small detail that we found is regarding the proof of why the cutset structure is accurate to within n^{-c}. In their proof they state that if we use the constant $c' = -c\log(\frac{8}{9})$ this can be done, however when plugging into the analysis at the top right of page 1134 in their paper, they claim that $(1-\frac{1}{9})^{-c\log(8/9)\log n} \leq n^{-c}$. This just isn't true, since $\log(\frac{8}{9}) < 1$. Instead, the constant needed to be chosen is $c' = -c\frac{1}{\log(\frac{8}{9})}$ as we have done. To see this more clearly, we can think about why this is necessary by going back to change of base formula. We have that $\log_{\frac{8}{9}} n = \frac{\log n}{\log(\frac{8}{9})}$. Thus with our constant, we are essentially plugging in $-c\log_{\frac{8}{9}} n$ to the exponent which becomes $(1-\frac{1}{9})^{\log_{\frac{8}{9}} n^{-c}}$ which then simplifies exactly to n^{-c} as desired.

6 Conclusion: Towards a unified theory of dynamic graph connectivity

We hope our project did a good job of providing insight into what drove some of the novel ideas behind these two landmark papers. We're interested in seeing what potential progress can be made towards achieving the proven lower bound of $O(\log n)$ amortized updates and deletes, and what can be done to trim off some of these $\log$ factors in the deterministic algorithm. It would be interesting to see if it's in fact necessary to use randomization to solve this problem deterministically, or if there is some other property that could be used to handle these updates. Finally, it will be especially interesting to note if future data structures end up using this hierarchical tree structure to accomplish the same goals that Holm and Kapron did. The idea of having clever bookkeeping to support updates and deletes is common even in other disciplines in computer science. The first example that comes to mind is the fenwick tree, where certain bits are in charge of certain ranges of queries. Though not the exact same, the motivation is similar. We wonder if some sort of similar HTS may be used in other randomized algorithms to derandomize some dependency that a structure might have.

We hope to continue to explore these questions later on in our theory career!

7 Appendix: Pseudocode for Kapron Cut-Set Insert and Delete

```
Insert (v_i-v_j):
    T_i = Tree for v_i                  //Found using Euler Tour tree
    T_j = Tree for v_j
    for each version w:
        for each level l:
            with probability 1/2^l:
                Label_wl v_i = XOR(Label_wl v_i, v_i-v_j)
                Set_wl(v_i) += v_i-v_j

                Label_wl v_j = XOR(Label_wl v_j, v_i-v_j)
                Set_wl(v_j) += v_i-v_j

                Update Label_wl Ti , Label_wl T_j
    If Ti != T_j:
        Link Ti and T_j in F using v_i-v_j

Delete (v_i-v_j):              //Assuming v_i-v_j already exists
    If v_i-v_j in F:
        remove v_i-v_j from F
        T_i = Tree for v_i
        T_j = Tree for v_j
        v_k-v_l = searchcut-set(T_i)
        if v_k-v_l is not None:
            Link T_i and T_j in F using v_k-v_l
    for each version w:
        for each level l:
            if v_i-v_j in Set_wl(v_i):  //if true, also true for Set_wl(v_j)
                Label_wl v_i = XOR(Label_wl v_i, v_i-v_j)
                Set_wl(v_i) -= v_i-v_j

                Label_wl v_j = XOR(Label_wl v_j, v_i-v_j)
                Set_wl(v_j) -= v_i-v_j

                Update Label_wl(T_i) , Label_wl(T_j)

searchcut-set(T_i):
    for each version w:
        for each level l:
            if Label_wl(T_i is an edge in G:
                return Label_wl(T_i) //because this is a cut-set edge
    return None
```

References

[1] Eran Eyal and Dan Halperin. Dynamic maintenance of molecular surfaces under conformational changes. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 45–54. ACM, 2005.

[2] Eran Eyal and Dan Halperin. Improved maintenance of molecular surfaces using dynamic graph connectivity. In *International Workshop on Algorithms in Bioinformatics*, pages 401–413. Springer, 2005.

[3] Chandrajit Bajaj, Rezaul Alam Chowdhury, and Muhibur Rasheed. A dynamic data structure for flexible molecular maintenance and informatics. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 259–270. ACM, 2009.

[4] Igor Ulitsky and Ron Shamir. Identification of functional modules using network topology and high-throughput data. *BMC systems biology*, 1(1):8, 2007.

[5] Jacob Holm, Kristian De Lichtenberg, Mikkel Thorup, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[6] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. Society for Industrial and Applied Mathematics, 2013.

[7] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[8] Mihai Patrascu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pages 263–271. IEEE, 2007.