Top Trees: A Generic Data Structure for Tree Augmentations

Kai Fronsdal
Department of Computer Science
Stanford University
kaif@stanford.edu

1 Introduction

A dynamic problem is defined by its constantly changing underlying data. Many such problems lead us to work with a forest structure at some point, for which we must efficiently compute certain properties. However, the straightforward implementations for calculating these properties can be highly time-consuming. For instance, calculating the length of the path between two vertices by simply traversing the path can result in a time complexity of $O(n)$, where n represents the number of vertices. Naturally, we seek methods to optimize this runtime. Interestingly, this is feasible, and the derived data structure, known as a *top tree*, offers efficient computations for numerous properties via a relatively user-friendly interface.

The top tree falls under the umbrella of dynamic forest data structures. The dynamic forest problem requires the maintenance of a forest through edge addition and deletion operations [1]. Alongside these dynamic operations, the forest's vertices and edges can hold additional data. This supplementary information can assist in addressing a variety of problems such as the diameter problem (which aims to maintain the longest path in the tree such that the cumulative edge weight in this path ceases to increase), the 1-center global search (which seeks the node that minimizes the maximum distance to other nodes), and the 1-median global search (which pinpoints the node that minimizes the weighted distance to other nodes).

In this paper, our main focus will be the application of top trees to the intriguing problem of maximum flow, which can be addressed using a dynamic tree-based solution. The maximum flow problem seeks to optimize the flow rate within a flow network, wherein each edge's capacity signifies the maximum achievable flow through that edge. This problem is relevant in a variety of fields such as bipartite matching, circulation and demands, to name a few. In the forthcoming sections, we will delve deeper into the intricacies of the maximum flow problem and propose an innovative solution to address it, relying on information maintained on a dynamic forest.

However, it should be noted that top trees represent a highly complex data structure, and existing literature on the topic generally stretches to hundreds of pages. In this study, we will tactfully explore the process of creating a dynamic forest data structure, starting with the basics, and shedding light on the subject in a way that is easily graspable. We will conclude with a discussion of a potential expansion of top trees to operate on directed acyclic graphs and shed light on the complications encountered in this proposition.

2 Related Works

Aside from top trees, multiple dynamic tree data structures introduced [1] in support $O(\log n)$ time edge insertion and deletion in the dynamic trees problem mentioned in the precious section, such as ST-tree[2], topology trees[3], and RC-trees[4]. ST-tree divides trees into vertex-disjoint paths and represent each path by supplementary data structure such as splay trees. The nodes are then ordered by depth in the partitioned tree in the supplementary data structure. Topology tree is based on tree contraction and reduces the size of the tree in $O(\log n)$ rounds. RC-tree is a variant of topology trees that also supports $O(\log n)$ expected time per operation. Both have the flaw that the $O(\log n)$ expected time is only valid when degrees of the represented trees are bounded. In this paper, we will

focus on top trees, which have the advantage of providing a more generalized interface that allows for a range of different applications without modification to the data structure[1].

3 Augmented Binary Search Trees

Before we get into creating a top tree, let us turn out attention to a simpler data structure that serves a very similar purpose – to provide an easy way of calculating information about an underlying set of data. In this case, we would like to calculate information on one of the simplest data structures, the array. Binary Search Trees (BST) are a class of dynamic data structure able to perform search queries in $O(\log n)$ time. The dynamic nature of these data structures means we can insert and delete elements from the tree while maintaining logarithmic search times. In order to maintain this runtime, the data structure must be able to restructure the binary tree to prevent the tree from becoming too unbalanced – resulting in search runtimes degenerating to $O(n)$.

The Red-Black Tree is one efficient kind of BST that maintains balance through series of tree rotations. In fact, this is the only operation it performs to change the structure of the tree. Because of this property, it becomes easy to augment each node of the tree with additional information to efficiently compute properties of the underlying data later. If we can describe the addition information we want to augment the tree with in a local manner – that is in terms of the current node, its children nodes and perhaps its parent – we can slightly alter our tree rotation to automatically maintain each node's augmented data in terms of our description.

Notice that each node of the BST corresponds to some subarray of the represented data. For example, the root corresponds to the entire array and the leaf nodes corresponds to individual elements or subarrays of length one.

However, this effectively decouples the description of the augmented data and the maintenance of the balanced tree. Therefore, if we can describe some property of the data in a terms of splitting arrays into parts, computing the property of each half, and combining the results into the property of the entire array we can simply use this description in conjunction with an augmented Red-Black Tree with no knowledge of how the augmented Red-Black Tree works. All we need to know is how to describe how to combine properties of subarrays together to get the properties of a larger array. We can treat the augmented Red-Black Tree as a black-box.

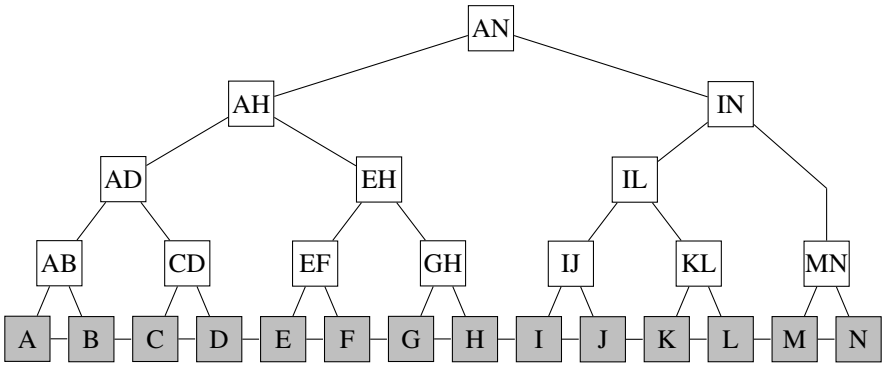

Figure 1: An example of an augmented BST on a array. Each vertex corresponds to a subarray. We can construct each vertex from its two children. The root corresponds to the entire array from A to N.

4 Top Trees

The insights we borrow from augmented search tree is that they support range queries that covers data in a wide range of the data structure but still use sub-linear time. A key intuition from the design is that we should maintain hierarchical layers on top of the data we store in the underlying data structure and walk over these layers to look for our augmented information.

A natural question that follows augmented BST is whether it is possible to generalize this idea from arrays to other data structures. It turns out that it is possible to generalize to forests. That is, there exists a black-box augmented data structure capable of computing properties of forests efficiently that only requires a simple description of how to combine together the properties of subtrees together to get the property of a larger tree. Similar to the case in augmented binary search tree on top of an array, we may wish to answer queries of nodes that lie far from each other in a tree, e.g., the length of their distance on the tree, by a hierarchical data structure on top of the tree. This data structure is called a Top Tree.

4.1 Clusters

In a BST we thought of our building blocks for the tree as subarrays, where we join together two smaller subarrays to make a larger one. Similarly, for an underlying tree T we can think of joining together connected groups of edges which we call *clusters*. To build a tree on top (which we aptly call the top tree) of the underlying tree T we merge together clusters of edges starting with the original edges in T – called *base clusters* – until we have one cluster representing all of T.

An important question is why don't we join together groups of vertices rather than edges? After all, when merging together subarrays, we were merging together groups of elements, not the connections between adjacent elements. While it is possible to create data structures built on this idea, generally it limits the maximum vertex degree possible to work with. The main culprit for this issue is that (unlike an array where each element is neighboring at most two other elements) any given vertex may be neighboring an unbounded number of other vertices. Thus aggregating information about all of its neighbors would take time proportional to the degree.

On the other hand, edges do not have this problem, as each edge is neighboring only two vertices which we call its *endpoints*. More specifically, for a given edge there are at most two endpoints that are shared with other edges. These endpoints that are shared with multiple edges are called *boundary vertices*. For instance, a vertex that is a leaf is not a boundary vertex since it only touches one edge.

It turns out this property is so nice that we will choose our merge operations in such a way that we ensure that all of our clusters will touch at most two vertices that also neighbor other clusters. We also call these vertices *boundary vertices*. We then call a cluster with only one boundary vertex a *leaf cluster* since it behaves a bit like an edge connected to a leaf vertex and we call a cluster with two boundary vertices a *path cluster* since there is the notion of a path between the two boundary vertices.

Since a cluster always has two endpoints, it can be thought of as an edge. Moreover, it can be naturally mapped to both a path and a subtree of the original tree. The path is the one between its endpoints; the subtree is the one induced by all base clusters that descend from it. We call the vertices in these subtrees that are not endpoints *internal vertices*.

This idea that a cluster implicitly encodes information about a path or subtree will play a central role in being able to augment the data structure as it provides a natural way to ask question about a given path. However, currently we can only ask about a path from u to v for which there is a cluster with u and v as endpoints. Thus, we will need some way to efficiently change the structure of the top tree so that there is a cluster with boundary vertices u and v. We call this operation `expose` and we will cover it more in a later section.

Another consequence of clusters having at most two boundary vertices is that we can then treat each cluster as an edge in a tree. Thus we only need to define merging for the structure of base clusters (aka the original edges) and they then easily extend directly to merging together larger clusters by simply treating each cluster as an edge.

4.2 Merging Clusters

The reason an augmented BST is so effective is that the height of the tree it built was logarithmic in the number of elements in the underlying array. If we want to construct a top tree with equivalent runtimes, we also need to ensure our top tree has logarithmic height. At each layer of the BST we effectively reduce the number of subarrays be about half the previous layer. It runs out that if can reduce the number of clusters through merging by a constant fraction at each layer of the top tree, we can unsure it has logarithmic height. Thus we continue this section by creating two merging operations that can meet this goal.

We already basically have one method of merging edges that we can steal from the augmented BST. An array can just be thought of as a *path segment* – a sequence of vertices of degree two. Just as we paired together elements of the array, we can pair together the edges in this path segment to merge. We call merging of two edges on a vertex of degree two a *compress merge*. An example of this kind of merging is illustrated in Figure 2.

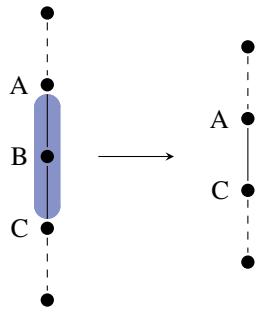

Figure 2: Example of compressing clusters AB and BC into a new cluster AC. Note that B has degree two. B becomes an internal vertex of the cluster (A, B) on the right.

We can also perform multiple compress merges in parallel. When we have a long enough path segment, we can perform multiple compresses on the entire length. Notice that each edge can only take part in one pairing. In order to perform multiple compress merges in parallel, we greedily pair up as many neighboring edges as we can. It is possible that this greedy approach does not optimally pair up the edges – that is, there might be a better set of pairings that results in more compress merges occurring. However, finding an optimal set of pairing is very challenging and would take a long time; but, as we will find out, this greedy approach is good enough for our purposes.

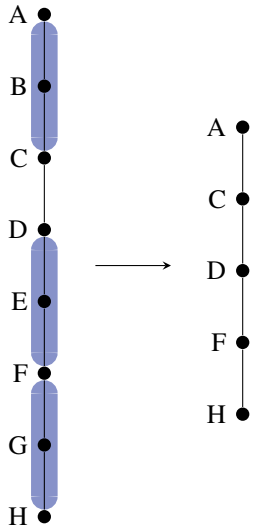

Figure 3: Pairing up the clusters in a path segment. We pair up neighboring clusters (indicated by blue) and merge them together to get the new parent clusters. Notice that not all edges were paired up.

However, we also need a way of dealing with vertices that result in branching. Let us consider a very simple case where we have a single central vertex c with several edges connected to it and one leaf vertex connected as in Figure 4. We can merge together the leaf cluster associated with the leaf vertex with any of the other clusters connected to c effectively "flattening" the tree a bit. We call this merging with a leaf a *rake merge*. This "flattening" makes the tree approach something that looks like a path segment. If we can eventually merge enough clusters around c that c has degree two, we can then perform a compress merge.

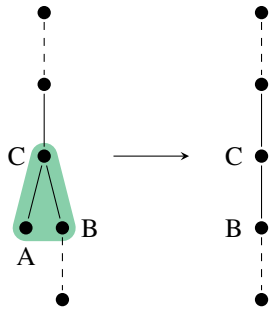

Figure 4: Example of raking the leaf cluster AC onto BC. Note that A has degree one and becomes an internal vertex in the cluster (B, C) on the right.

Similar to performing multiple compress merges in parallel, when we have multiple leaves around a single vertex we can perform multiple rake merges. Again, each edge can only be take part in one pairing. Thus if an edge was already paired for a compress merge, it cannot also take part in a rake merge. Below is an example of multiple rake merges occurring around a central vertex.

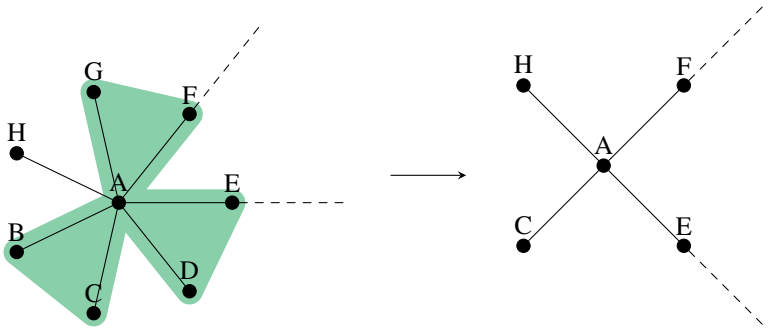

Figure 5: Pairing up the clusters around a central vertex A. Notice that while E and F are not leaves, D and G are, so (A, D) and (A, G) get raked onto (A, E) and (A, F) respectively. Since G and B were both part of a rake merge, (A, H) remains unmerged at this step.

It is easy to then combine these two merge operations to work in parallel with each other. Combining both compress and rake merges together we get that performing multiple compress and rake merges in parallel looks something like Figure 6.

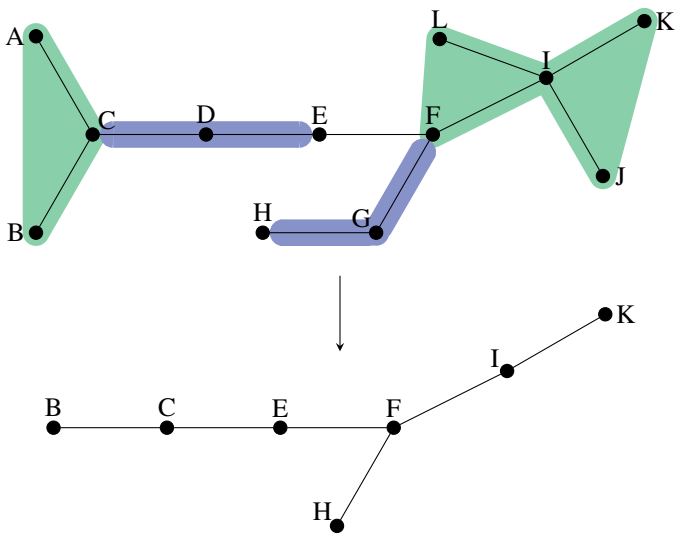

Figure 6: A single round of tree contraction. At the top is a tree we would like to contract. We have paired up clusters along the various path segments to be compressed (blue) and paired up the leaf clusters to be raked (green). Below is the new contracted tree constructed by merging the marked clusters from above.

Surprisingly these two operations are all we need to ensure we can merge a constant fraction of clusters in a given tree which we will later show in section 4.4. However, the proof boils down to noticing that each compress merge corresponds to a vertex of degree two and each rake merge corresponds to a vertex of degree one (the leaf). We can then show that at least half of the vertices in any tree have degree one or two. Unfortunately we cannot actually merge half of the clusters in a given tree, but we will also show that any merge on a given vertex of degree one or two only prevents a constant number of other vertices of degree one or two from also taking part in a different merge.

4.3 Constructing the top tree

The complete algorithm for constructing a top tree from an arbitrary underlying tree T is now remarkably straightforward once we have defined these two merging operations and how they can work in parallel. In fact it will look very similar to how we construct an augmented BST on an array. Just as in constructing the augmented BST, we will proceed in rounds, where we apply a series of merge operations on clusters to *contract* the tree – in other words, make a smaller tree by merging together pairs of clusters.

The full algorithm for constructing a top tree is then as follows.

1. Initialize $T_0 = T$ and $i = 0$. Initialize top tree $\mathcal{T}$ with base clusters taken from T_0.
2. Greedily pair up neighboring clusters in T_i that can form valid compress or rake merges.
3. Perform the relevant compress and rake merges from the pairs found in step 2 to construct T_{i+1}
4. Insert new clusters from step 3 into $\mathcal{T}$ with pointers to the clusters that were merged to create them.
5. Increment i and repeat from step 2 until T_i has only one cluster

We work layer by layer from the bottom up. $T_0, T_1, \ldots, T_{O(\log n)}$ represent increasingly compressed versions of the original underlying tree. For example, in Figure 6 the tree at the top might represent some tree T_i. After pairing up all of the neighboring clusters that can form compress or rake merges as in step 2 and performing step 3 we get the contracted tree T_{i+1} below. Note that when a cluster is

not paired at a given layer, we can simply move it up a layer rather than making a duplicate cluster at the next layer.

For a complete example of constructing a top tree, see Figure 7. Examining the root cluster of the top tree, we can see that the single cluster in the tree representation at that layer has two endpoints. While these are not boundary vertices since they do not connected to root cluster to other clusters, the cluster still represents a path between them. Thus we call these endpoints of the root cluster the *external boundary vertices* of the top tree.

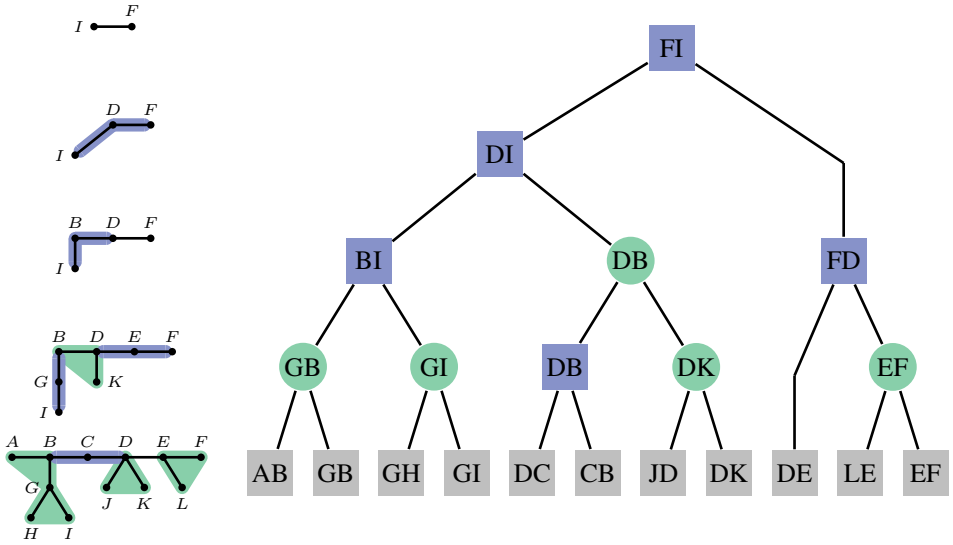

Figure 7: An example of building a top tree from scratch. On the left are the tree representations of each contraction round from T_0 on the bottom to T_i at the top. In green are the pairs of clusters that will be raked together in the next layer and in blue are the pairs of clusters that will be compressed together. On the right is the constructed top tree. Each vertex in the top tree corresponds to a cluster in the tree at the same layer on the left. The green circles correspond to clusters that were formed by raking two clusters in the layer below and the blue squares correspond to clusters that were formed by compressing two clusters in the layer below. The grey squares are the base clusters.

4.4 Top Tree Height

A necessary – but not sufficient – condition for top trees to provide $O(\log n)$ time operations is that the height of the tree is itself $O(\log n)$. We will first prove that we can apply enough rake and compress operations to a tree (or subtree) such that we eliminate a constant fraction of clusters each round. Since each rake and compress operation corresponds to a vertex of degree one or two respectively, we will show that at least half of all vertices in any tree have degree one or two and then that we will always be able to perform rake and compress operations on a constant fraction of these vertices. To show this second part, we will show that each rake and compress operation can block at most three other degree one or two vertices from participating in a rake or compress operation.

Lemma 1. *Let T be some tree or subtree and let D_i be the number of vertices with degree i within T. If T has at least two vertices then*

$$\sum_{i=1}^{\infty} D_i(i-2) = -2 \tag{1}$$

Proof. Notice that $\sum_{i=1}^{\infty} iD_i$ is the sum of the number of edges touching each vertex. Clearly each edge touches two vertices and the total number of edges in a tree is $n-1$ so

$$\sum_{i=1}^{\infty} iD_i = 2(n-1)$$

$$\sum_{i=1}^{\infty} iD_i = 2\left(\sum_{i=1}^{\infty} D_i\right) - 2$$

$$\sum_{i=1}^{\infty} D_i(i-2) = -2$$

□

Corollary 1. *At least half of the vertices in a tree have degree one and two.*

Proof. In equation (1) the vertices with degree 1 have a negative contribution to the sum, the vertices with degree 2 have zero contribution and higher degree vertices have a positive contribution. Since the total sum is negative, there must be more vertices with degree one than of degree three and higher. Thus at least half of the total number of vertices are degree one (rakeable) or degree two (compressible). □

We call these vertices of degree one or two *potential merges* as they are the only vertices that have the potential to become internal (and thus is not included in the new contracted tree) due to a merge operation. Thus by Corollary 1, at least half of the the vertices in a tree are potential merges. Recall that each merge operation corresponds to a single vertex becoming internal.

Lemma 2. *The total number of vertices in T_{i+1} is at most $\frac{7}{8}|T_i|$ if T_i has more than two vertices.*

Proof. Let $n = |T_i|$. First assume that $n > 8$, since if $2 < n \leq 8$ we will always have at least one rake or compress operation we can perform. From Corollary 1, we know that at least $n/2$ of the vertices in T_i are potential merges. We will show that at least a quarter of these vertices will become internal vertices in T_{i+1} due to a merge operation.

To do this, we will examine how many potential merges each compress and rake operation may prevent from becoming internal. In most cases, they can block at most 2 other operations, one on either side. However, there are a few cases where it is possible for it to block three operations. Let us look at an edge (v, w).

Case 1. We rake (v, w) onto another edge (v, x) and w becomes internal. Since this is a rake merge, w must have degree one, so there are no compress operations we can perform on (v, w). It is possible that another leaf (v, u) is now blocked from raking onto (v, w). We now look at (v, x).

If x has degree one, then we are done. Similarly, if x has degree higher than two. However, if x has degree two then we have blocked (v, x) from being compressed with its other neighboring edge (x, y). Moreover it is possible y is a leaf and now (x, y) cannot be raking onto (v, x). Thus in total, raking (v, w) onto (v, x) has resulted in one vertex w becoming internal and three u, x, and y from becoming internal.

Case 2. We compress (v, w) with (w, x) and w becomes internal. If v has degree one, then we have blocked (v, w) from raking onto (w, x) making v internal. If v has a degree of three or more and there is an edge (v, u) that is a leaf, then we have blocked (v, u) from raking onto (v, w) making u internal. Otherwise if v has degree two, then we have blocked compressing (v, u) with (v, w) also making v internal. However, if u has degree one, then we have also blocked raking (v, u) onto v, w. By symmetry with the other side of w we have blocked at most four vertices of degree one or two from becoming internal. However, since we assumed $n > 8$, we cannot have both u and y being leaves. Thus, it is actually only at most three degree one or two vertices that are prevented from becoming internal.

The three worst cases are illustrated in figure 8. For a more careful analysis of the cases, see either [5, 1]. What this means is that we are able to perform at least $\frac{n}{2} \cdot \frac{1}{4} = \frac{n}{8}$ merging operations for any tree resulting in T_{i+1} having size at most $\frac{7}{8}|T_i|$. □

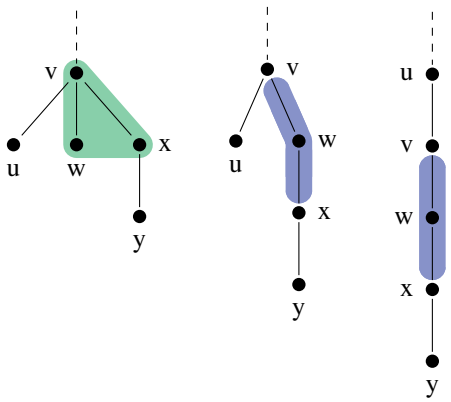

Figure 8: The three worst cases for cluster merging in which one rake or compress operation blocks three others.

A consequence of Lemma 2 is that at every level of the top tree, the number of clusters decreases by at most a constant fraction. Thus the total height of the tree is at most $\log_{\frac{8}{7}} n = 5.19 \log_2 n = O(\log n)$. In fact is is possible to prove a tight upper bound $|T_{i+1}| < \frac{5}{6}|T_i|$ which results in a height of at most $3.802 \log_2 n$ [5].

5 Top tree operations

5.1 Update Operations

The top tree mainly supports three dynamic operations:

- `Link`(v, w), which inserts an edge (v, w) into the underlying forest;
- `Cut`(v, w), which removes the edge (v, w) from the underlying forest;
- `Expose`(v, w), which restructures the top tree so that one of the root clusters (corresponding to the tree v and w are in) of the top tree has v and w as external boundary vertices.

The `Link` and `Cut` operations allow for dynamically updating the underlying forest and `Expose` allows us to compute properties between two specific vertices such as path length. We will explore using each of these operations further in section 6.1

It is possible to prove that all three of these operations can be done in $O(\log n)$ time. The importance of this $O(\log n)$ run-time complexity is that it allows us to query information along two clusters in the original tree in the time of $O(\log n)$ even if the actual tree-walk distance of the two clusters are larger than $O(\log n)$. Here we give a brief introduction on how these operations are implemented; however, we have omitted a proof of the runtimes as it is mainly a long series of case work and does not provide much in the way of intuition. The reader can refer to the proof from [1].

5.2 Supporting Link and Cut

After we either link or cut an edge on the underlying tree, we need to ensure that the the top tree structure both captures the new structure of the underlying tree and maintains logarithmic height. The key insight is that when inserting or removing an edge, only the clusters very close to the effected edge at each level of the top tree might need to be changed. For example, in Figure 9 by adding the edge (A, F), we can no longer compress (B, A) with (A, C) but we can still compress (E, D) with (D, B) which is further away from the added edge.

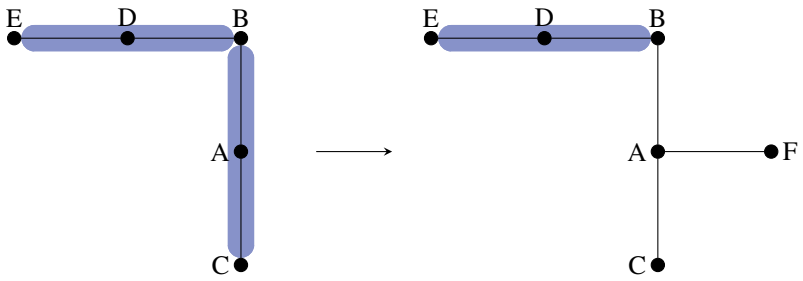

Figure 9: Effects of inserting an edge (A, F).

These changes will need to be propagated up the top tree since if a given cluster is added, removed, or altered, then the parent cluster will need to be changed as well or created if the given cluster is new. We take the very aggressive approach of completely deleting effected parent clusters and recomputing them from scratch. It turns out that since we can delete and create clusters in constant time this approach will result in logarithmic runtimes. However, we leave out the runtime analysis of these operations as they are very lengthy.

We implement this approach by updating each level of the top tree iteratively, starting at the lowest layer T_0 and working our way up repairing the damage that the link or cut caused. At each layer T_i we are going to be removing some clusters D_i that are no longer valid clusters (i.e. the compress or rake operation requirements are not longer satisfied or one of their children was either removed or updated) and then inserting some clusters I_i to replace the clusters we removed in order to fix the top tree structure. At the base level if we are performing a link, I_0 contains just the base cluster corresponding the the edge we just added and if we are performing a cut, D_0 contains just the base cluster corresponding to the edge we removed.

At each layer, we compute I_i and D_i as follows.

1. Add the parents of all clusters in D_{i-1} to D_i
2. Add all clusters in T_i to D_i whose compress or rake merges are no longer allowed by replacing D_{i-1} with I_{i-1}
3. Remove all clusters in D_i from T_i
4. Compute clusters I_i from the children of D_i with a greedy approach
5. Add all clusters in I_i to T_i

In step 1, we must remove the parents of deleted clusters as one of their child clusters no longer exists. In step 2, we can effectively compute this by only looking at clusters that are neighboring the effected areas (since as we saw in Figure 9 inserting and deleting clusters can effect neighboring clusters). More formally, we only need to check the parents of clusters that share a vertex with altered clusters since other clusters will be too far away to be effected by the change at layer i. We leave out the details of this step for brevity, but for a more detailed description, see [1]. In step 4 we employ the same greedy pairing approach that we used to construct a top tree from scratch, i.e. perform rake and compress merges on the unmatched children of D_i. Figure 10 illustrates how a cut operation's changes propagate up a top tree.

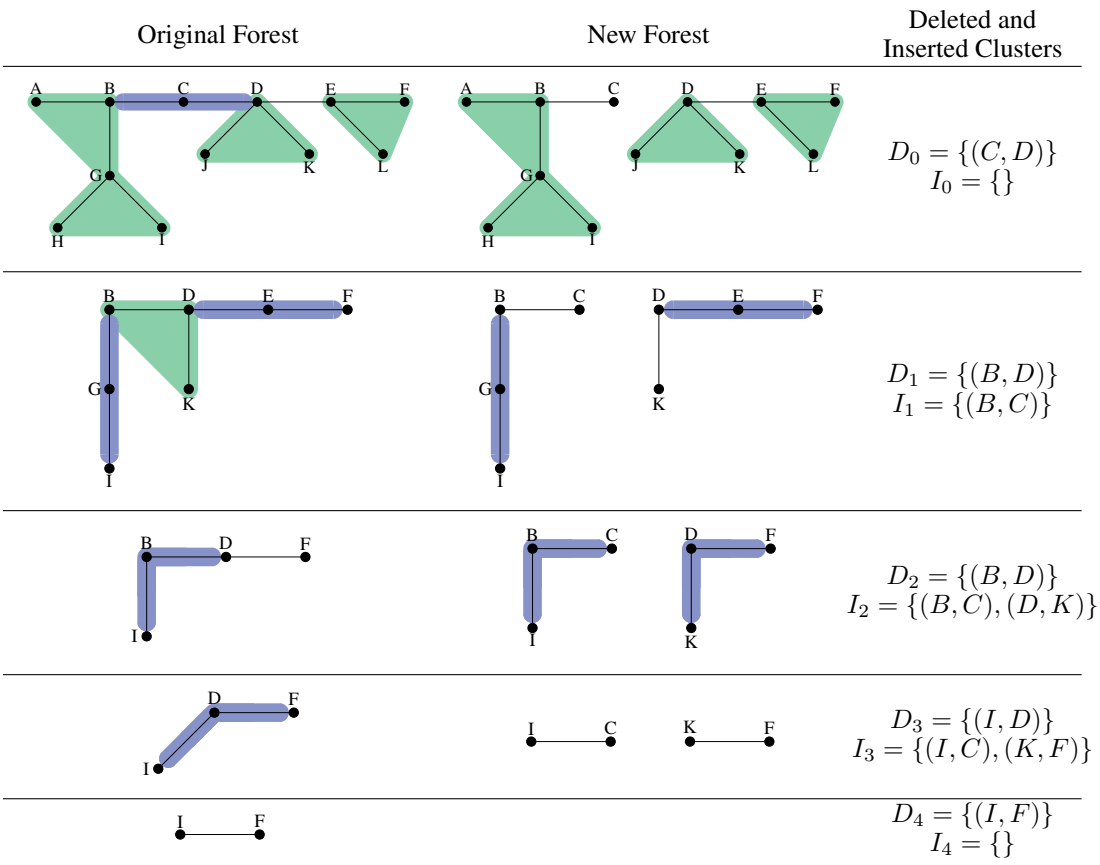

Figure 10: Process of updating the top tree when removing the edge (C, D). On the left is the original top tree from Figure 7 displayed using the tree representation of each layer. In the middle are the new tree representations of each layer after (C, D) is removed. On the right are the corresponding changes we made at each layer according to the update algorithm given above.

Now that we understand how to update the top tree under links and cuts we can see that our current definition of a rake merge has a serious flaw. It turns out the if we allow a leaf cluster to be paired with any other cluster with a shared vertex, the runtime of link and cut can degrade to $O(\log^2 n)$ even if we can perform each rake merge in constant time (including the time to find a potential pairing among the potential $O(n)$ other clusters). Consider a central vertex c with $2^k + 1$ leaves for $k > 1$. At every level of the top tree there are an odd number of clusters so there will always be one cluster left unpaired. If each of these unpaired clusters are distinct (that is no unpaired cluster is the descendent of another unpaired cluster in the top tree), then adding a new edge to the underling tree will result in each of the unpaired clusters at each level to get paired. We then need to update each of the parents of the formerly unpaired clusters. It turns out that each of these $O(\log n)$ clusters have $O(\log n - i)$ ancestors where i is the layer of the cluster in the top tree. Thus, after a careful analysis [1], the total number of clusters we need to fix is $O(\log^2 n)$.

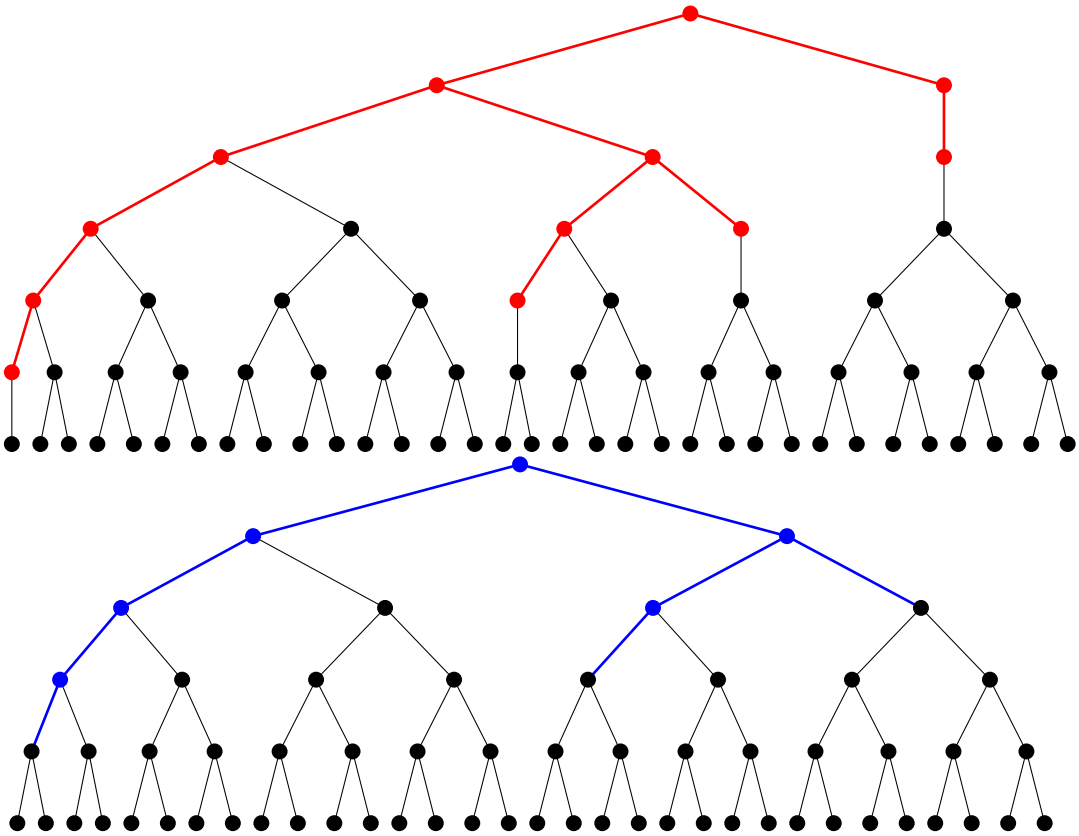

Figure 11: Why pairing arbitrary edges does not work. Above is the top tree representation of a star (One central vertex with multiple leaves attached). (upper) the original top tree of a star with $2^5 + 1$ leaves. (lower) The new top tree after the unpaired edge gets removed. Red vertices in the upper tree highlight the cluster nodes that will be removed by the update due to one of their children being removed. The blue vertices in the lower tree correspond to the clusters that were inserted into the top tree to replace the ones deleted.

To rectify this issue we would like to reduce the number of possible clusters each leaf cluster can be raked with; specifically we would like this to be bounded by a constant. One way to achieve this is by inducing a cyclical ordering on the edges around each vertex. A leaf cluster can then then only be raked onto its successor in the cyclic ordering. In this way we force each leaf cluster to only have one potential pairing (Or two if its predecessor is a leaf cluster as well). It turns out that this constraint still maintains a top tree height that is and the proof looks very similar to Lemma 2.

5.3 Expose

The purpose of `Expose`(v, w) is to ensure that the root cluster of the tree containing v and w has v and w as its external boundary vertices. What this means in practice is that we need to make sure v and w do not disappear in the process of building a top tree. By disappear we mean that they are no longer boundary vertices, instead they become *internal* vertices. If you go back to Figure 7 where we built a top tree from scratch, you will notice that each external boundary vertex is present as the endpoint of at least one cluster at every level. Every other vertex at some point becomes an internal vertex due to a merge and disappears from the tree representations of all layers above. Therefore, in order to implement expose, we need to ensure that v and w do not become internal vertices at any point in the top tree.

Notice that once a vertex becomes and internal vertex of a given cluster, it will remain an internal vertex of all of the ancestors of that cluster. Moreover, each vertex can only be an internal vertex of at

most one cluster per layer. Thus, we can trace a path from the root cluster to the first cluster where a given vertex became internal and be confident we passed through all clusters where that vertex was internal. Thus a simple algorithm for implementing expose is to remove each of the clusters for which v or w are internal.

The result of removing all of these clusters is that we will be left with a collection of all of the children of the clusters we deleted. This results in a collection of $O(\log n)$ clusters. Since it is impossible for us to delete a base cluster in this process (as they do not have an internal vertex), the remaining collection of clusters form a partition of the original underlying tree.

We can then use the edge interpretation of these clusters. In other words, this collection of clusters forms a tree of size $O(\log n)$. We can then build a top tree from scratch from this tree with the restriction that we cannot perform an merge that would make v or w an internal vertex. For the example in figure 12, we find the clusters with G and J as internal vertices (in grey) and remove them, resulting in a partition the underlying tree into the clusters AB, GI, GB, DB, JD, DK, and FD. Below is the tree formed from this collection of clusters. Since there are $O(\log n)$ clusters, we can build this top tree in $O(\log n)$ time.

An important note is that after we are finished working with the exposed top tree, we revert back to the original top tree before performing any link and cut operations.

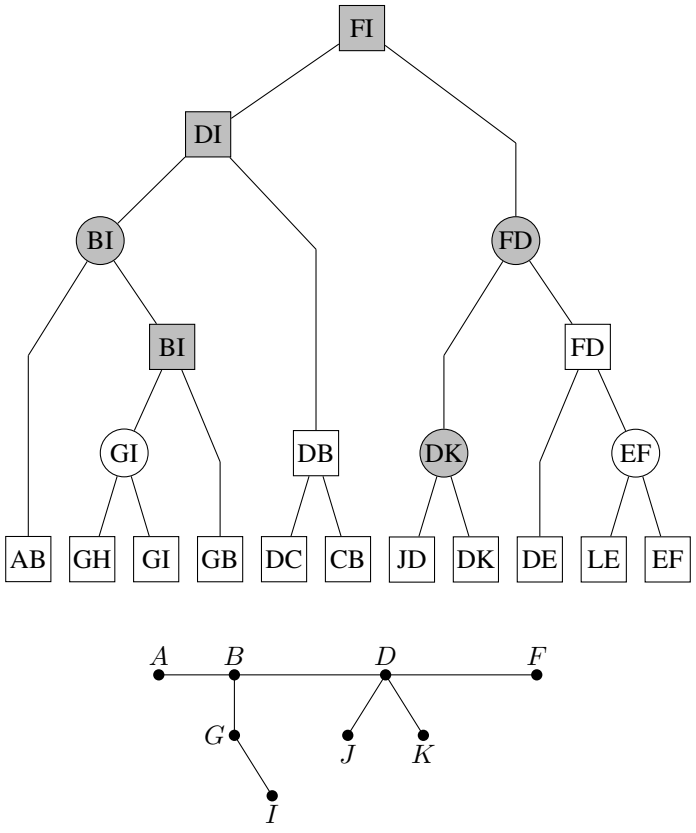

Figure 12: Above is the Top tree from Figure 7 corresponding to `Expose`(G, J). Clusters containing G or J as internal vertices that will be removed are shaded grey. Below is the new tree representation of the edge partition from which we will build a new top tree.

6 Applications of Top Trees

6.1 The Top Tree Interface

The Top Tree supports the three update operations (`Link`, `Cut`, and `Expose`) described in the previous section. As the top tree handles those update operations, the various changes made to the top tree are decomposed into a sequence of the following four internal operations[5]. `Info`(C) is user defined information associated with the cluster C, an augmentation of the top tree with auxilliary data (in the BST analogy, this would be the augmentation at each BST node). The user must define how the `Info` of each cluster is updated or recomputed as each of the following internal operations occur. In essence, the user of the top tree will call the update operations `Link`, `Cut`, and `Expose` as desired, and the top tree will "callback" with a sequence of these internal operations that accomplishes the desired update operation[1]. The user must define the following four operations:

- `Merge`(A, B). Whenever we merge two clusters A and B, we create a new larger parent cluster C. The purpose of this operation is to tell the top tree how to set the auxiliary data for the new cluster C. Specifically, `Merge`(A, B) computes `Info`(C) from `Info`(A) and `Info`(B), and returns new cluster C, where the boundary vertices of C are the boundary vertices of the union of A and B.

- `Split`(C). Whenever we remove a cluster C, we may want to update the information of its children A and B. The `Split` operation computes `Info`(A) and `Info`(B) from `Info`(C), and cluster C is removed from the tree. However, in many use cases, we will just leave A and B unchanged during a split operation.

- `Create`$((u, v))$. Whenever we add a new edge, we want to create an associated base cluster with auxiliary data. The `Create` operation creates a new cluster C from the edge (u, v), and `Info`(C) is computed from scratch.

- `Eradicate`(C). When we cut an edge in the underlying forest, we will end up removing the corresponding base cluster. However, unlike other clusters in the tree, base clusters do not have any children. Thus the `Eradicate` operation is called when a base cluster cluster C is deleted from the tree, but it still might store `Info`(C) in some manner so that this stored information may later be used when cluster C is re-created with the `Create` operation. This will be motivated when we discuss applications of `Split`.

Regarding time complexity of the above interface, the same paper[5] proves a theorem that a forest of top trees of height $O(\log n)$ can be maintained with sequence of at most $O(\log n)$ `Merge` and `Split` and $O(1)$ `Create` and `Eradicate` per operation, which can be computed in $O(\log n)$ time. The overall runtime then, is the $O(\log n)$ time of computing the sequence, combined with runtime of the user-defined `Info` function.

To further explain the interaction of top trees with data structures and algorithms that use them, we will introduce the internal operations and their complications one at a time and present some applications of the internal operations.

6.2 Applications of `Create` and naive `Merge` in Minimum Weight Query

A highly simplified example of a use case for top trees is querying the lowest cost edge in a tree[6]. The way we will go about achieving this is very similar to how we made the tournament heap that let us efficiently compute the minimum element in a set. We build up a binary tree on on top of the underlying tree where each cluster points to whichever child contains the edge with the smaller weight. For the `Create`$((u, v))$ operation, we look up the weight of the edge from u to v, and store it as `Info`(C). In this case, the cluster C is simply the edge from v to w. Then, for the `Merge`(A, B) operation, we ignore all metadata and context about A and B, and simply compute and store `Info`$(C) \leftarrow \min($`Info`$(A),$ `Info`$(B))$. We will leave the other operations `Split` and `Eradicate` as no-ops.

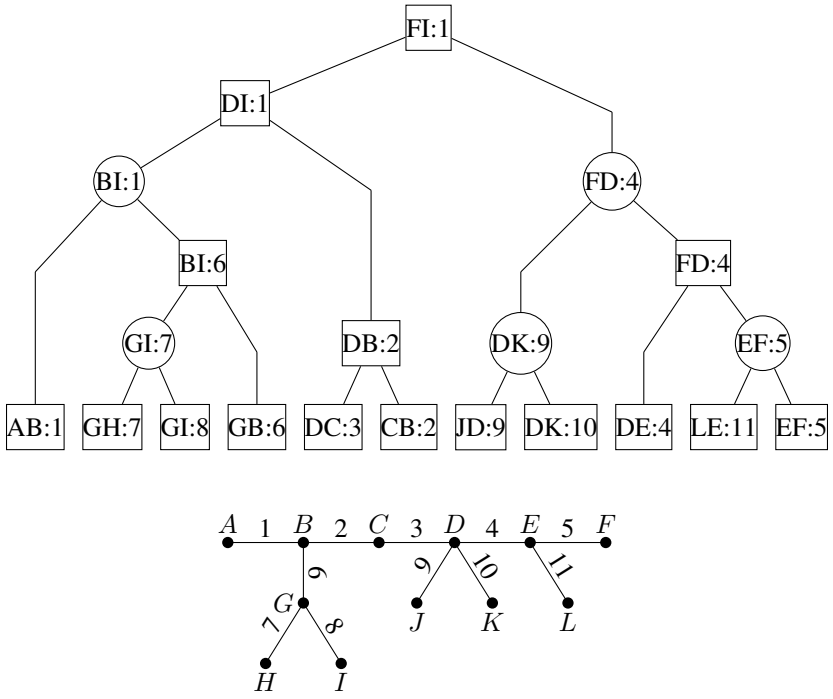

Figure 13: An example of top tree used for lowest cost edge query. The same top tree is used in example for construction of top trees in Figure 7. For each cluster, `Info`(C) stores the minimum cost of the edge contained in the cluster. With each merge operation, `Info`(C) is updated. Eventually, the topmost cluster gives the lowest cost and the query of minimum weight takes constant time.

This is a simple example of augmenting the top tree with user-defined metadata, in this case simply the lowest weighted edge of the cluster. By querying the augmentation of the root cluster (namely, `Info`$(root)$), we can learn the minimum weight of our tree in $O(1)$ time. Figure 13 shows an example of the minimum weight query using a top tree.

6.3 Merge based on `rake` versus `compress`

For querying simple information about the entire tree, a naive `Merge` implementation that ignores how the clusters beneath it were merged will suffice. However, clusters are merged in two different ways, either a `rake` or a `compress`. We can define `Merge-rake`(A, B) to handle the cases where a leaf cluster A is raked onto a cluster B, in which the boundary vertices of the combined cluster C are just those from B (as A was raked into B), and we'll define `Merge-compress`(A, B) to take the cases where A and B are being compressed, in which the external boundary vertices of the combined cluster C are taken one each from A and B[6]. In fact there are really five different ways to merge two clusters together as illustrated in Figure 14.

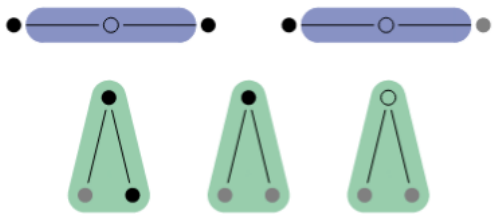

Figure 14: The five distinct ways to merge together two clusters (up to symmetry). The black vertices refer to boundary vertices that will remain boundary vertices in the parent cluster. The hollow vertices refer to boundary vertices that will become internal vertices in the parent cluster. The grey vertices refer to the remaining endpoints in the children clusters. The clusters connected to a grey vertex are leaf clusters.

The intuition to motivate why we might like to treat these two kinds of `compress` differently comes from the unique feature of the top tree that sets it apart from simpler data structures (such as ST-trees or Euler Tour forests), namely the `Expose` operation.

When we call `Expose`(u, v), the top tree rearranges itself to expose these two vertices as the top level external boundary vertices of the tree. This is conceptually similar to the `evert` operation on an ST-tree, or the `splay` operation on a splay tree, in which an arbitrary node is hoisted up to the root in order to facilitate some operation on it. Simply, the effect of `Expose`(u, v) is to cause the root of the tree to have external boundary vertices of u and v, in other words, the root of the tree is a cluster representing an edge between u and v (note that this is only after clustering, u and v need not actually be direct neighbors in the actual tree). This pair of vertices, representing the final remaining edge in the hierarchical processing of the tree, are called the external boundary vertices.

This operation is used to analyze the path between u and v. By `Expose`-ing them, the top tree recontorts itself such that the path between them is expressed as a sequence of `compress` operations. The intuition for this is that in order for u and v to be the endpoints of the root cluster, all parts of the tree that aren't on the path from u to v must have been `raked` on to the path between them at some point. The clusters along the path from u to v can never be raked onto another cluster since they will always have degree two or higher (since they will always have a path connecting to u and a path to v). Thus, since the path from u to v is unique, the only possible way to process the tree all the way down to a singular root cluster of u and v is if the clusters along the way from u and v were all compressed (in some order), see figure 20. The intuition for this can be found by looking at the outcomes of `rake` and `compress`: if the boundary vertices of A are (u, v) and B are (u, w), `rake`(A, B) will have boundary vertices (u, w) (preserving B but discarding A), whereas `compress`(A, B) will have boundary vertices (v, w) (combining A and B). Everything extraneous to the path from u to v must at some point be `raked` in, whereas everything on the path from u to v can only be `compressed` and never `raked`.

The upshot is that by calling `Expose`(u, v), then looking at the `compress` operations that contribute to the augmentation of the root node of the top tree, we can analyze the path from u to v and ignore the rest of the tree (in $O(\log n)$ time).

To make this concrete, a simple example is path cost queries. Consider the problem of wanting to answer queries of "What is the total weight of the path from u to v" (presumably while the tree in question is also being updated with `Link` and `Cut` operations).

We initialize the augmentation of each edge to its weight, as in the previous section. Upon the `Create`$((u, v))$ operation, we simply set `Info`$(C) \leftarrow weight(u, v)$. On a `Merge-compress`$(A, B)$ operation, we set `Info`$(C) \leftarrow$ `Info`$(A) +$ `Info`(B). The intuition is that the total weight of a path is equal to the sum of the total weight of the sections that comprise it. However, we do something different for `rake`. On a `Merge-rake`(A, B) operation, we set `Info`$(C) \leftarrow$ `Info`(B) (recall that A is the leaf cluster being raked onto B). The intuition is that we want to discard any information from other parts of the tree, we only care about the path currently being queried. (recall that we defined the first argument to `rake` to be the cluster being raked, while the second argument B is the cluster into which A is being raked). By calling `Expose`(u, v), we see that `Info`$(root)$ will hold the total weight

of the path from u to v. Figure 15 shows a simple example of using the `Expose`(u, v) operation to acquire sum of edge weights along a particular path.

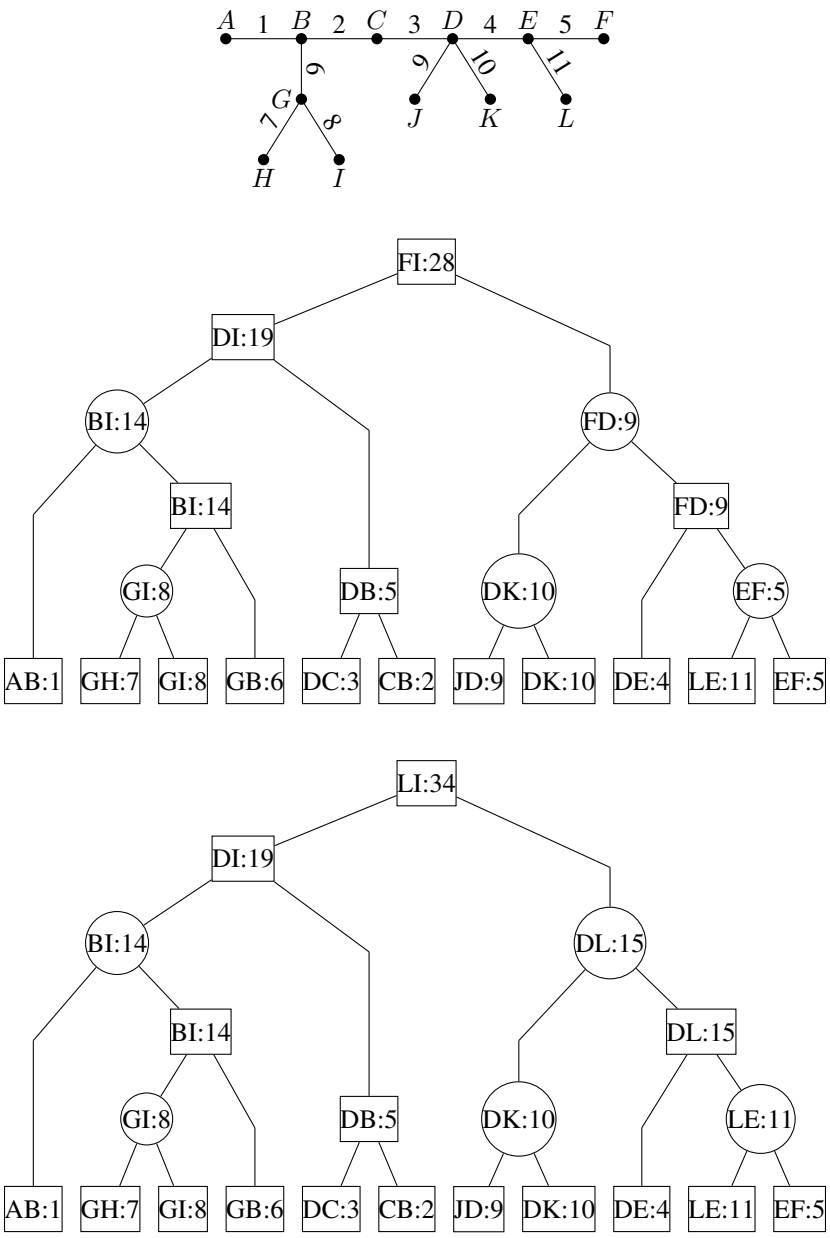

Figure 15: This figure shows an example of application in finding sum of edge weights along a path through `Merge-rake` and `Merge-compress`. The topmost figure shows the same tree as in Figure 13. The first top tree shows the initial constructed top tree, in which the root node stores the sum of edge weight along the path from F to I. The second top tree shows a top tree reconstructed from the operation `Expose`(L, I) to acquire the sum of edge weights along the path from L to I.

6.4 Application to dynamic minimum spanning tree

This approach can help us maintain a minimum spanning tree in the face of dynamic edge additions[6]. While the minimum spanning tree problem takes as input a graph and outputs a tree, we can

nevertheless use top trees if we formulate the problem as a sequence of edges being considered and potentially discarded, allowing our top tree to remain as a minimum spanning forest at every stage. While edges can arrive in any order thus allowing the internal state to temporarily be a forest, if the incoming edges eventually fully connect into a single graph, the outcome will be a single tree. Note that edge deletions are not covered by this approach. Leave `Create` and `Merge-rake` as before, but set `Merge-compress`(A, B) as `Info`$(C) \leftarrow max($`Info`$(A),$ `Info`$(B))$. We are now augmenting the top tree with the maximum weight along the exposed path, rather than the sum of said weights. Now, whenever a new edge (u, v) arrives, we test whether it connects different trees, and `Link` it if so. If not, we must test this edge to see if it ought to be part of the minimum spanning tree, or discarded. We leverage the top tree and call `Expose`(u, v). The exposed path of the top tree is now the alternate existing route from u to v. We query the augmentation of the root, and learn the maximum edge weight along the alternate route from u to v. If the new edge has less weight, we `Cut` the other highest-weight edge and `Link` this new one, otherwise discard this edge as not an improvement.

6.5 Applications of Split(C)

In simple tree augmentations, the augmentation at each node is a pure function computed from its child nodes. For example, in the simple case of augmenting a red-black tree with the subtree size at each node, we can always accurately say that the augmentation at each node will be the sum of the augmentation of its children. In top trees however, the augmentation is stateful; meaning that the user-defined information attached to each node is **not** always necessarily equal to what would be arrived at by recomputing all augmentations from scratch. [1] presents applications of "split" and provides example of solving the maximum flow problem summarized below. The intuition is that while simpler augmentations are strictly "bottom-to-top", top trees allow for "top-to-bottom" modifications. In the same sense that an augmented BST allows data to be constantly collated from $O(n)$ sources and updated in $O(\log n)$ time, this opposite direction allows data to be lazily distributed to $O(n)$ destinations and updated/accessed in $O(\log n)$ time.

As a motivating example, consider the maximum flow problem. Top trees can helpfully answer queries of "Locate the lowest-weight edge along the path from u to v" in $O(\log n)$ time, but to be helpful for maximum flow, we also would like some way to do "Decrease all weights along the path from u to v by some Δw". In lecture we covered several ways of implementing the decrease-key operation for entire trees in a heap. The main idea was to lazily propagate the total change to the keys by storing the value of the change in the root node and slowly propagating the changes down as the structure of the heap changes. It turns out we can use a similar approach with top trees. In fact top trees can do this in $O(\log n)$ time! We add to each augmentation a new field Δw. After exposing the path from u to v, we simply add our desired Δw to the augmentation of the root cluster. Upon the `Split-compress`(C) operation, we apply `Info`$_{\Delta w}(A) \leftarrow$ `Info`$_{\Delta w}(A) +$ `Info`$_{\Delta w}(C)$ and `Info`$_{\Delta w}(B) \leftarrow$ `Info`$_{\Delta w}(B) +$ `Info`$_{\Delta w}(C)$, whereas for `Split-rake`(C) we only apply that adjustment to B. Simply, since C is being destroyed, we must preserve the weight adjustments applied to it, by pushing it down into its child clusters. If the child clusters were the result of a `compress`, our weight adjustment rightly applies to both, but if we are a `rake` cluster, only the right child ought be affected by this adjustment, since only that side of the tree comprised the exposed path. Queries to each cluster are appropriately modified by its Δw. When we want to get the weight of an edge (u, v), we first expose u and v, so that the root cluster corresponds to the edge between them and then return the weight of that edge plus the Δw stored at the root cluster.

If we still want to support finding the minimum weight edge, we need to slightly change how we update the minimum edge auxiliary data. Specifically, for a `Merge-compress`(A, B), we set `Info`$_w(C) = \min($`Info`$_w(A) +$ `Info`$_{\Delta w}(A),$ `Info`$_w(B) +$ `Info`$_{\Delta w}(B))$.

7 Interesting Component

We will present a deep dive into precisely how top trees fit into maximum flow, as described in the previous section but in more depth.

7.1 Introduction to Maximum Flow

The below definition of the maximum flow problem is adapted from [7]. Maximum flow is an optimization problem that looks for the maximum flow rate in a flow network, where each edge has a designated flow capacity indicating the maximum amount of flow allowed to pass through the edge. Figure 16 shows an example of a flow network with a source to terminal maximum flow of 9.

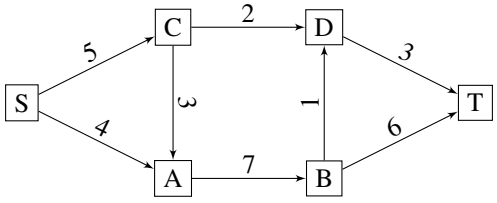

Figure 16: A sample flow network with maximum s-t flow of 9.

The next section presents a formalized definition of the maximum flow problem.

7.2 Formalizing Maximum Flow

This section provides a formalized definition of the maximum flow problem similar to one provided in [7]. Assume a directed weighted graph $G = (V, A)$, with V as the vertices and A as the arcs. An "arc" shall refer to an ordered pair of vertices in a directed graph, while an "edge" shall refer to an unordered pair of vertices. Two distinct vertices in V are designated the "source," s and the "sink" t. The weight of each edge is interpreted as a nonnegative "capacity", denoted c_{ij} for arc (i, j). For simplicity, arcs that do not exist are treated as arcs with $c_{ij} = 0$, and vice versa. A "flow" on the graph assigns a flow value f_{ij} to each arc, satisfying properties $0 \leq f_{ij} \leq c_{ij}$ (no arc can flow more than its capacity) and for all i other than the source and sink, $\sum_j f_{ji} = \sum_k f_{ik}$ (sum of incoming flow equals sum of outgoing flow). The "value" of a flow is the sum of flow exiting the source, or entering the sink ($value = \sum_i f_{si} = \sum_i f_{it}$). The maximum flow algorithm shall determine a flow (assignments to f_{ij}) that maximize this value.

7.3 Residual capacity and Residual network

A very simple algorithm for solving max flow that we would like to work (but does not) would be to simply find paths from s to t such that there is space to increase the flow along the path ($c_{ij} - f_{ij} > 0$ for each (i, j) along the path) and then increase the flow along that path. We would repeat this process until we can't find such a path any more. Unfortunately this does not work. For example in figure 17, if the first path we choose is $s \to A \to B \to t$ and we set the flow along that path to be 10, then there are no more possible paths from s to t that we can increase the flow along. However, it is clear that we can get a total flow of 20 by sending 10 units of flow along the paths $s \to A \to t$ and $s \to B \to t$ respectively.

The problem is that we forced 10 units of flow to go across the edge (A, B). We would like to be able to move this flow along a different edge where we might be able to increase the flow elsewhere, in essence undo the choice of path we made before. To do this we need some way of being able to reroute already assigned flows which we will do using something called a *residual graph*[7]. A residual graph R of a network G has the same vertices as G, but for each edge (u, v) in G, R has the arcs

1. A forward arc (u, v) with residual capacity $c_{uv} - f_{uv}$
2. A backward arc (v, u) with residual capacity f_{uv}

If we look back at the example in Figure 17, if we assign 10 units of flow across the path $s \to A \to B \to t$. In the residual graph there will then be an arc $B \to A$ with residual capacity 10 (and the arc

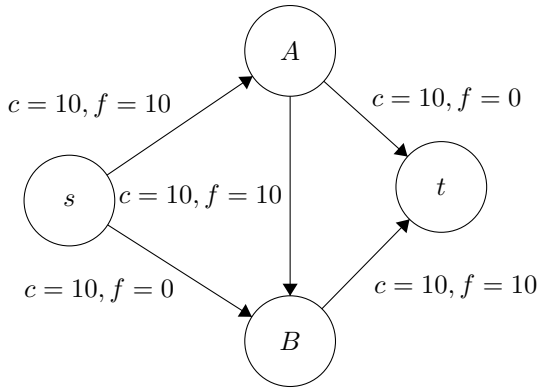

Figure 17: This flow is not maximal, it only admits a flow of 10 while a flow of 20 is trivially possible. Without the reverse arc exception, our residual network would only include (s, B) and (A, t) (so, no path from s to t in G_f), which would be a problem because the algorithm would appear to have reached a dead end, having painted itself into a corner by choosing to augment $s \to A \to B \to t$. But, with the residual graph, the action of assigning flow 10 to (A, B) also created a backward arc from B to A with a residual capacity of 10. Now, our residual network includes (B, A). The next augmenting path will be $s \to B \to A \to t$, even though (B, A) was not in the original graph.

$A \to B$ has residual capacity 0). Thus in the residual graph there is still a path from s to t that we can send flow across. However, if we send flow across this path $s \to B \to A \to t$, we will be adding 10 units of flow to the forward edges $s \to B$ and $A \to t$ and removing it from the backward edges $B \to A$. We call a path in the residual graph where all the arcs have positive residual capacity an *augmenting path.*

For a flow f, it can be proven that if there does not exist a path from s to t in the residual graph G_f, then f is a maximal flow. For a flow f, it can also be proven that if there exists a path from s to t in the residual graph of f, then there exists a augmenting path. This formulation leads to a potential solution to maximum flow. Consider an algorithm that repeatedly finds an augmenting path, finds its capacity (the minimum residual capacity among its arcs), and increases the flow among all such arcs by that amount. Sadly, poor choice of augmenting paths can lead to super-polynomial running time, and, interestingly, in the case of irrational arc weights, it may not ever converge (per [8]). However, it turns out that selecting augmenting paths with a BFS works very well. The resulting algorithm is called the Edmonds-Karp algorithm and runs in time $O(m^2n)$ runtime since each BFS to find an augmenting path takes time $O(m)$ and we may have to augment as many as $O(mn)$ times.

However, as we can see in Figure 18, these BFS for augmenting paths can repeat a lot of the same computations. Ideally we would like to find some way of reusing the work done from previous searches for augmenting paths.

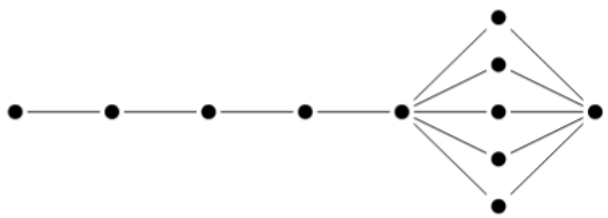

Figure 18: A graph where a BFS will have to search through the same tail on the left for every path.

7.3.1 Intuition for choosing augmenting paths using distance labels

It happens that simple heuristic of choosing augmenting paths is able to achieve $O(n)$ augmentations. The following method comes from [1]. For each node v, define $dist(v)$ as the length of the shortest path from v to the sink t, across the residual network G_f. If there is no such path, default to ∞. This distance ignores capacity, it is a simple quantity count of arcs. Notice that the shortest path from any vertex v to t, the value of $dist$ decreases by one from v to t. Thus if we want to find short paths on G_f, we only need to look for these sequences of decreasing distances. First, note that as long as there exists any augmenting path, $dist(s)$ will exist. We define the *admissible network* G_L as a subgraph of G_f, including only arcs (u, v) that satisfy $dist(u) = dist(v) + 1$. Intuitively, G_L includes precisely the union of all arcs that could form an augmenting path of minimum length.

Now, assume we choose augmenting paths exclusively from G_L. After saturating at least one arc along each such path (which is the effect of increasing its flow by its minimum capacity, as earlier), it is possible to prove (although we omit the proof) that the resulting residual network G_f will have a strictly increased $dist(s)$. Overall, $dist(s)$ can only range from 0 to n, thus demonstrating that this algorithm must terminate after augmenting n paths. However, we must still deal with finding out how long it takes to find and augment each of these paths.

7.3.2 How to Choose Augmenting Paths

It turns out that the actual algorithm to find augmenting paths along the admissible path can be quite simple. Also adopting a method called Dinitz's Algorithm as described in [1], we consider a simple depth first search, with recursive backtracking. We start at the source s and expand our path iteratively. At every iteration, we locate admissible arcs (an arc in the admissible network) originating from our current vertex v. If there is no such arc, then we have reached a dead-end (which must have been caused by a previous iteration saturating an arc). In this case, we recompute the correct $dist(v)$ by finding the unsaturated successor of v that with the smallest distance and setting $dist(v)$ to be one more than that. We then set v to be this successor. If there are no successors of v then we immediately "retreat" (backtrack, returning to the previous vertex before v). If the new vertex is the sink, we look back along our current recursive path from the source, compute the minimum residual capacity, and then increase the flow along all arcs in the path by this residual capacity. This is guaranteed to saturate at least one of them (whichever had the minimum residual capacity). Intuitively, every time this algorithm reaches the sink t, the fact that it saturates at least one arc along its path means that it "burns" at least one distinct way to get to the sink. By the time it "retreats" backtracks all the way to the source and completes, all routes to the sink in the current admissible network have been saturated at at least one arc (and the `dist` values were all recomputed wherever needed).

By using this process we end up reusing much of the same computation when finding augmenting paths rather than having to recompute everything from scratch. However, we still have one major bottleneck which is that updating the weights of ever arc along the augmenting path can still take $O(n)$ time in the worst case (which is still better than $O(m)$ for a BFS). Thus the total runtime is $O(mn^2)$. However as we saw previously, top trees provide a very easy way of changing the weights along an entire path in logarithmic time. If we can utilize a to tree to perform this operation, we could improve performance to $O(mn \log n)$.

7.3.3 Dinitz's Algorithm via Dynamic Forest Interface

In this section, we will start from how a dynamic forest interface can implement the Dinitz's Algorithm to calculate the maximum flow in a directed graph. This discussion will allow us to further explore how the top tree intuitively implements the dynamic tree interface and therefore solve the problem of maximum flow.

We've discussed that the major inefficiency in Dinitzs algorithm is that successive searches starting from the source will potentially re-discover paths with positive residual capacity that are already found in previous searches. It would be more efficient if there is a data structure for us to remember which portion of the graphs we have explored, and in particular for the maximum flow problem, which portions of the graph that still have positive residual capacity. Then instead of searching for an augmenting path from the source, we can try to to combine these segments we have already explored to a complete augmenting path.

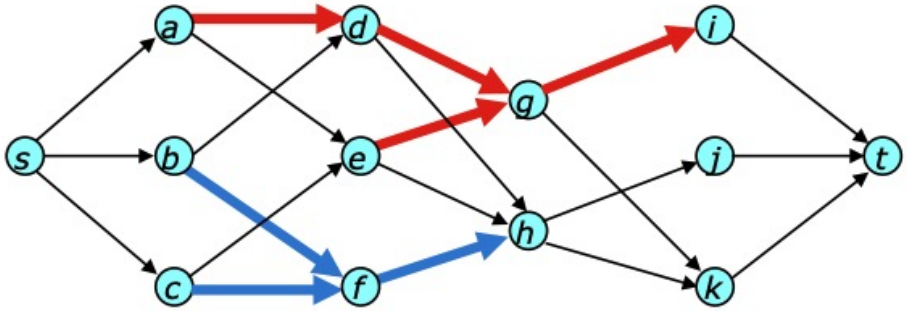

Figure 19: Using a forest to record augmenting paths [9]. The colored subtrees consist of edges (u, v) where the residual capacity is positive and $dist(u) = dist(v) + 1$.

By using the forest to record the partial path from the DFS of exploring augmenting paths, we present the following overview of how to use dynamic trees to implement the Dinitz's algorithm. Initially, we can think of all forests as isolated points in the graph with no edges in between. Note that along each path we explore, we always consider the root of a tree to be the last point in the partial path that ends with it (see Figure 19). Denote s as the source of the graph (or its representing node in the forest), and (a, b) as an directed edge $a \to b$, then

1. Start with each node in its own tree in the forest.
2. Define a "current node" v, which will always be at the end of the path we are searching.
3. Set $v = \texttt{root}(s)$.
4. If $v \neq t$, then we want to move forward one edge in the DFS (intuitively, this urges the network to flow in the direction to the sink in each step). Denote the edge as (v, w), then we perform link(v, w) in the dynamic tree and assign the weight (i.e. cost) of the edge in the tree to be the weight of (v, w) in the graph, and start again from step 3.
5. If there are no edges coming out from v, then there is no way to move flow away from v, so we want to delete it from the forest. Hence, we delete all incoming edges (u, v). If any of these edge are also an edge in the dynamic forest, perform $\texttt{cut}(u, v)$ to avoid information sharing of its corresponding tree.
6. If $v = t$ then we've arrived at the sink. We can perform the augmentation by calling $\texttt{min-weight}(v)$ and augment our dynamic tree by subtracting this minimum weight along the path, then delete all edges with resulting weight 0. Similarly when we delete these edges from the dynamic forest, we perform a cut as in Step 5.
7. The algorithm terminates when there is no outward pointing edges from s.

Looking at the above algorithm, we want our dynamic forest structure that supports the following operations efficiently:

- $\texttt{root}(v)$: given a node v, find the root of tree that contains v;
- $\texttt{link}(u, v)$: merge the tree containing u and v by making the tree rooted at u the child of node v (assuming that u is a root and that v is in a different tree than the tree rooted at u);
- $\texttt{cut}(v)$, given a node v, cut the subtree rooted at v off from the parent of v;
- $\texttt{min-weight}(v)$: given a node v, return the minimum weight edge on the path from v to $\texttt{root}(v)$;
- $\texttt{update}(v, \Delta)$: adds Δ to all edges on the path from v up to $\texttt{root}(v)$

However, we have already shown that we can support each of these!In fact, if n is the number of vertices in the graph, the top tree is a good candidate for a dynamic forest that support each of these operation in amortized $O(\log n)$ time.

Assume, as previously, that our top tree is augmented with the minimum residual capacity along each path. Whenever we reach the sink, we saturate the path that we took. The top tree can do this

in $O(\log n)$ time. Then, we need to locate all arcs that are now saturated. While there can be more than one, the number of saturated arcs removed is amortized $O(1)$, because every arrival at the sink will permanently remove at least one arc, and there are only m arcs in total, across all n outer loop iterations. Again, the top tree can locate an arc that is now zeroed (saturated) in $O(\log n)$ time, then `Cut` it in $O(\log n)$ time. In this case, we optimize the run time by adding another augmentation to `Info`, which mainly supports the `min-weight` operation above as described in the next section.

There are n outer loops in the above algorithm as a loop over each node to start with, and $m \cdot O(\log n)$ operations within each loop since the number of "advancing" to the next node with possibly a link and "retreating" with possibly a cut is asymptotically the number of arcs in the graph, $O(m)$. As a result, this reduces the runtime of the maximum flow algorithm to $O(mn \log n)$, a large improvement over the naive $O(n^2 m)$ algorithm which didn't use the dynamic forest!

8 Top Tree Generalization

A natural question to pursue is whether or not it is possible to generalize the idea of a top tree further. After all, it is already a generalization of an augmented BST. Unfortunately, being able to perform operations of general graphs efficiently was out of reach as the possible complicated structures of edges we found too challenging to deal with. However, we did come up with a few ideas for directed acyclic graphs (DAGs), which, while being a subset of graphs, are much more general than forests.

The idea came to us while thinking about the transitive closure problem – being able to answer questions of the form "can you reach vertex b starting from vertex a?" We discovered after a bit of effort that solving this question for forests is actually not too hard once we have to top tree interface.

8.1 Transitive Closure for Directed Forests Using Top Trees

When two vertices are exposed, we reshape the structure of the top tree so that the root cluster corresponds to the path between them. What this results in is that every cluster on that path is always compressed, and any cluster that extends off of the path is at some point raked onto it. This is easier to visualize in Figure 20.

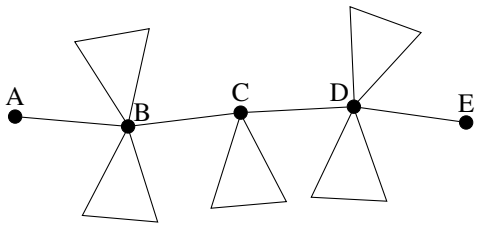

Figure 20: One way to interpret the structure of the top tree after exposing (A, E). The five clusters along the path will be compressed while everything else will end up as being part of one of the triangle clusters that will then be raked onto the path from A to E.

A consequence of this structure is that if we only want to know some property of a path, we can completely disregard rake merges and simply have clusters remain unchanged when a leaf is raked onto them. This leads us to a very simple algorithm for computing the transitive closure problem in forests. In each cluster with endpoints v and w we keep track of the information "can we reach v from w?" and "can we reach w from v?"

There are really only three possible combinations for the answers to these questions since in a tree there are no cycles so we can only ever travel one direction. Thus for brevity we record being able to get to v from w as $w \to v$ and being able to get to w from v as $v \to w$. If we can't go in either direction we record our answers to the questions as $\times$.

For the base clusters, the answer is given by the direction of the edge. That is given an edge (v, w), we record in the base cluster $v \to w$. When compressing two clusters (a, b) and (b, c) we can then calculate the information for the merged cluster as follows. If the recorded information of the children

point in the same direction, that is $a \to b$ and $b \to c$, we are able to get from a to c so we record $a \to c$. Similarly, if the children have values $b \to a$ and $c \to b$ then we record $c \to a$. Otherwise, if either of the children have the value $\times$ or point in opposite directions we record $\times$.

Thus to answer the question can we reach w from v in a forest, we simply expose v and w and then look at the value recorded in the root cluster.

8.2 DAGs

It is rare that you will need to answer the transitive closure question on a forest however. It is much more common that you will have a DAG instead.

The main idea behind this generalization came from the last lecture on Planar Point Location, where there was the concept of a purely functional red/black tree where no vertex is changed. The implementation of this data structure results in turning a tree into a DAG where (ignoring many details) vertices point to both past children and new changed children. The idea for generalizing Top Trees for DAGs is the opposite. Instead of turning a tree into a DAG we want to turn a DAG into a tree. We call this process *tree-ification*. If we are able to successfully achieve this we could then use the top tree data structure on this tree-ified version of the DAG.

Tree-ifying a DAG is not actually very complicated. For every edge going into a vertex, we make a copy of that vertex and everything that descends from it and change the edges to each point to a unique copy. The result of this process is a tree version of the DAG with the underling structures and relationships preserved. For example of tree-ification see Figure 21.

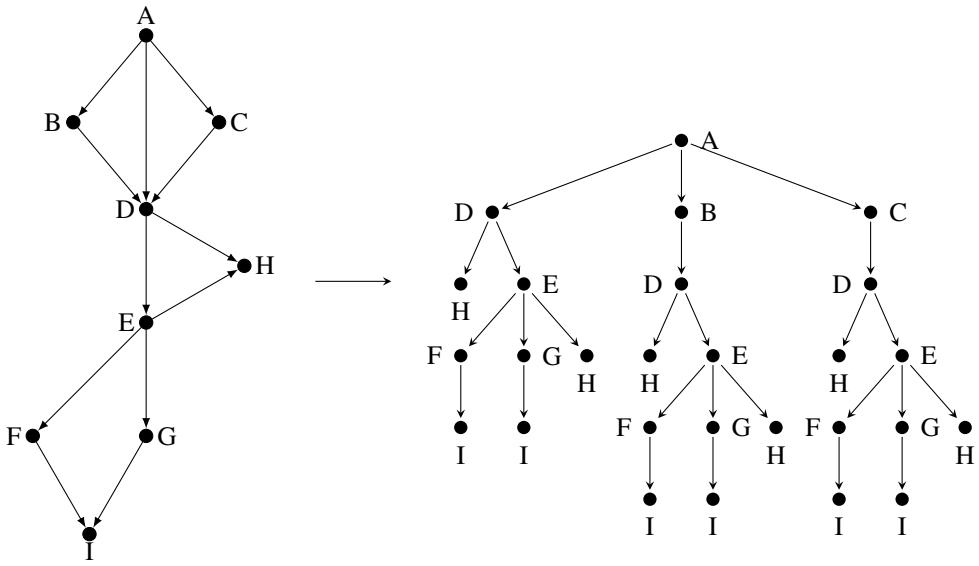

Figure 21: Tree-ifying a DAG.

This tree has some nice properties. For instance, it is possible to get from a to b in the DAG if and only if it is possible to get from a to some copy of b in the tree. Additionally, if Δ is the maximum degree in the DAG and $\mathcal{H}$ is the height of the DAG (the maximum path length), then it is not hard to see that the total number of vertices in the tree-fication is at most $O(\Delta^{\mathcal{H}})$. Thus, if we were to use a top tree on this tree-ification we would have a tree height of $O(\mathcal{H} \log \Delta)$. In the worst case the height and maximum degree are both linear in n so the total height is $O(n \log n)$. Thus, in general we are able to answer questions (by using expose) about the DAG in worst case $O(\mathcal{H} \log \Delta)$ time.

Unfortunately this tree has two major problems. The first is that the number of vertices in the tree can be as large as $O(\Delta^{\mathcal{H}}) = O(n^n)$. For some specific applications such as using DAGs for pairing algorithms where $\mathcal{H} = 2$ the size is not too bad only $O(n^2)$. However, this is an enormous tree in general to store in memory and not particularly feasible. The other is that dynamic operation do not

run as efficiently as expose does. To see this, imagine removing the edge (F, I) in Figure 21. We would have to remove three edges in the tree-ified version of the DAG. In the worst case, where we have a a single sink vertex and add an outgoing edge from that sink, we would have to add $O(n)$ edges from the tree-ified DAG resulting in a total worst case runtime of $O(n^2 \log n)$.

If you reexamine Figure 21, you will notice that there are a lot of duplicate subtree structures. For instance, The two subtrees rooted at the two D vertices are completely identical. When we built the top tree we expect that many of the parents of these two subtrees are going to be identical or at least very similar. Let us look closer at the vertex D in Figure 21 we can see that in the tree-ification there is a duplicate subtree rooted at a copy of D for every edge pointing to D. It would be nice if we could reuse these clusters when we build the tree.

A reasonable next question would be what do rake and compress merges on the tree-ification look like when compared to the edges in the DAG? Notice that a vertex becomes a leaf in the tree-ification if and only if it is a sink in the DAG. In other words it has an out-degree of zero. Similarly, a vertex has degree two in the tree-ification if and only if it has an out-degree of one in the DAG.

What if we redefine rake and compress using this notion of out degree instead? For rake merges, rather than looking for vertices of degree one, look for vertices of out-degree zero and perform a separate rake merge for each of the edges pointing into the vertex. Similarly instead of compressing a vertex with degree two, look for vertices with out-degree one and perform a separate compress merge with each of the edges pointing into the vertex.

In other words, we perform rake and compress merges pretending that we are operating on a tree. However, each contraction layer is still being represented in DAG form rather than the tree-ification. Thus we call the data structure a top DAG. Notice that this process results in $O(m)$ clusters at each layer where m is the number of edges. A crude bound on the total space usage is $O(m\mathcal{H} \log \Delta) = O(mn \log n)$ which is a major improvement from the exponential space usage.

However, we suspect that actually the total space usage might even be as good as something like $O(m \log m)$ or even $O(m)$. The reason for this is because in the tree-ification process, the further away the DAG gets from being a tree (in other words the more edges it has), the more branching will occur in the top tree and the more duplicate sub tree structures will occur. Empirically it seems like these repeating substructures contract at a faster rate than forests do in general. Just from looking at examples it appears possible that the maximum height of the top tree is only $O(n)$. For example, one pathological structure, the DAG generated by the less than predicate on the first n natural numbers $\{(i, j) : i, j \in \mathbb{N}, i < j \leq n\}$, results in $O(n^2)$ edges. However, a relatively straightforward inductive argument, showing that the tree representation's structures at each layer of the top tree converge to the same sequence for each n after two contraction rounds (although slightly messy as there are some edge cases to worry about), implies that the height of the top tree built on top of the tree-ification has height $n + 1$ and thus the top DAG also has height $O(n)$. Another consequence of this would be that expose also runs in time $O(n)$.

Another problem that needs to be answered is how fast the dynamic operations perform on this top DAG. The runtime of update operations was already by far the most complicated part of the analysis of a normal top tree, so proving anything on this more general data structure will likely be even more challenging. In a top DAG each edge can be a part of at most $O(\Delta)$ base clusters. Thus the best performance we could hope to get for a top DAG is $O(\Delta n)$ if the height of the top DAG is $O(n)$ or $O(\mathcal{H}\Delta \log \Delta)$ otherwise. However, there is a chance that a careful analysis of top DAGs would show that actually the number of clusters that need to be updated in each layer of the top DAG increases by a factor of about $O(\Delta)$. The reason why this could be the case is if each of the $O(\Delta)$ base clusters we need to add or remove each results in $O(\Delta)$ other clusters needing to be changed and so on. The growth rate of the number of effected clusters at each layer probably is not quite that bad because even in the top tree, each cluster can effect up to four other nearby clusters in the worst case, but the number of effected clusters at each layer is still at most a constant.

8.3 Transitive Closure Revisited

Because we built up the top DAG in reference to a tree, almost any augmentation we can use on an top tree we can use on the top DAG. Specifically for our case, we can use the transitive closure augmentation. Assuming that the top DAG is actually able to perform updates in time $O(\Delta n)$ then it could perform relatively well in comparison to other dynamic transitive closure data structures.

Query Time	Update Time	
1	n^2	Demetrescu and Italiano [10]
$n^{0.58}$	$n^{1.58}$	Sankowski [11]
$n^{1.407}$	$n^{1.407}$	van den Brand et al. [12]
n	$m + n \log n$	Roditty and Zwick [13]
n	Δn	Ours

Table 1: Table of query and update time of our top DAG compared to other dynamic transitive closure data structures.

References

[1] Renato F Werneck. *Design and analysis of data structures for dynamic trees*. Princeton University, 2006.

[2] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[3] Greg N Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

[4] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. 2004.

[5] Jacob Holm and Kristian de Lichtenberg. Top-trees and dynamic graph algorithms. 1998.

[6] Renato Werneck. Slides on design and analysis of data structures for dynamic trees. `https://www.cs.uoi.gr/~loukas/courses/grad/Data_Structures_and_Algorithms/index.files/Dyntrees_Werneck_Thesis.pdf`.

[7] Jessica Su. Cs161 lecture notes: Lecture16 - maxflow.

[8] Harvey J. Greenberg. Ford-fulkerson max flow labeling algorithm, 1998.

[9] Jonathan Turner. Dinics algorithm with dynamic trees, 2013. Available at `https://www.arl.wustl.edu/~jon.turner/cse/542/text/sec19.pdf`.

[10] Camil Demetrescu and Giuseppe F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure, 2001.

[11] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse: extended abstract. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 509–517, 2004.

[12] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds, 2019.

[13] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC '04*, 2004.