Efficient maximum subsequence queries and updates for dynamic forests *

Bishnu Bhattacharyya † Frank Dehne ‡

Abstract

The problem of efficiently maintaining attributes of a dynamic forest modified by edge insertions and deletions is one that has been extensively studied. In this paper, we present a novel dynamic data structure that supports edge insertions and deletions while maintaining non-local properties in $O(\log n)$ time. We use this data structure to solve an important open problem: a dynamic data structure with $O(\log n)$ query, insertion, and deletion time for computing maximum subsequences between pairs of query nodes in dynamic forest.

1 Introduction

In *dynamic trees problems*, attributes (eg, diameter) of a forest of trees are maintained as it changes over time via edge insertions and deletions. An edge *insertion* connects the leaf of one tree to the root of another; an edge *deletion* splits one tree into two by removing an edge.

Both these operations can be naïvely implemented in $O(1)$ time. However, maintaining the attribute may become costly. For example, the tree diameter must now be repeatedly re-computed, leading to a query cost of $O(n)$. Consequently, it is desirable to maintain an additional data structure, encapsulating this information, which can be efficiently updated. This has motivated significant research in the area of dynamic tree algorithms.

In [1], Alstrup et al refine the work of Frederickson [10] and Sleator et al [16] and present *top trees* as one such data structure. Top trees implement queries for tree diameter in $O(\log n)$ time and support edge insertion and deletion in $O(\log n)$ time. Given a fixed-degree tree T and and a set δT of *boundary vertices* containing one or two nodes in T, a top tree of T is a binary tree whose root represents T, whose leaves represent the edges of T, and sibling nodes represent subtrees of T who intersect in a single vertex and whose union forms their parent.

However, top trees distinguish between *local* and *non-local* properties of trees. If an edge or vertex of a tree T exhibits some local property p, then all subtrees of T containing that vertex/edge also exhibit p. Top trees naturally lend themselves to computing local properties. In [1] the authors also present a modification to top trees that maintain tree center and median, again supporting $O(\log n)$ time queries, but it is cumbersome and does not extend to other problems easily. Furthermore, by allowing any vertices in T to be set as a boundary, any algorithm using top trees must account for many possible representations of the same forest. Additionally, queries of top trees can result in the data structure being modified. In [17], Tarjan et al extend top trees to include trees of arbitrary degree. However, the cost for edge insertion and deletion is now *amortized* $O(\log n)$.

Henzinger et al [13] outline *ET-trees*, a binary tree built on top of an Euler tour of a graph. This is used to maintain a number of different dynamic attributes: connectivity with $O(\log^3 n)$ update time and $O(\frac{\log n}{\log \log n})$ query time; bipartiteness with $O(\log^3 n)$ update time and $O(1)$ query time; a $1+\epsilon$ approximation

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

†School of Computer Science, Carleton University, Ottawa, Canada, bbhattac@connect.carleton.ca.

‡School of Computer Science, Carleton University, Ottawa, Canada, frank@dehne.net, http://www.dehne.net.

of a minimum spanning tree is maintained in time $O(\frac{\log^3 n \log U}{\epsilon})$ per update, where U is the weight of the heaviest edge in the graph. ET-trees have a relatively simple implementation but are only used when a value is to be maintained over a collection of edges and/or vertices. The computation of this value must be associative and commutative. This restricts the type of attribute ET-Trees can dynamically update.

In this paper, we present a dynamic data structure with time complexity $O(\log n)$ per edge deletion and insertion that supports $O(\log n)$-time queries of the *maximum subsequence with respect to u and v* in a tree, which is currently an open problem for dynamic trees. The query algorithm does not alter the data structure. Given a sequence of real numbers S, the subsequence with the highest sum is the *maximum subsequence*, and the problem of finding this subsequence is the *maximum subsequence problem* [5]. In the field of bioinformatics, this problem arises frequently in the analysis of DNA and protein sequences [14], homology modeling [12], ontology matching [11], and microarray design [6]. The maximum subsequence is also used when ranking k maximum sums [3] and computing the longest and shortest sub-arrays satisfying a sum or average constraint [8]. In [15], Ruzzo et al present a $O(n)$ time algorithm that computes the maximum subsequence. This problem is extended to trees as follows:

Definition 1. *Given a weighted tree T and nodes u, v, the* **maximum subsequence with respect to u and v** *is the maximum subsequence of the sequence formed by taking the edge weights on the path connecting u to v. This is denoted $MS(u, v)$.*

The goal is to perform repeated queries of the maximum subsequence between various vertices in a forest that evolves over time. A top tree-based solution is impractical as $MS(u, v)$ is a non-local property. With respect to ET-trees, the maximum subsequence of two concatenated sequences depends on the order in which they are concatenated, so commutativity does not hold. This makes ET-trees also unsuitable.

We present a dynamic data structure with time complexity $O(\log n)$ per edge deletion and insertion that supports $O(\log n)$-time queries. Our data structure is motivated by the spine decomposition of a tree [4], which is a static data structure on a tree. We review the static spine decomposition in Section 2. In Section 3, we present our method for handling edge insertion and deletion in a dynamic forest. In Section 4, we describe our query algorithm for evaluating $MS(u, v)$.

2 Review: Spine decomposition of a tree

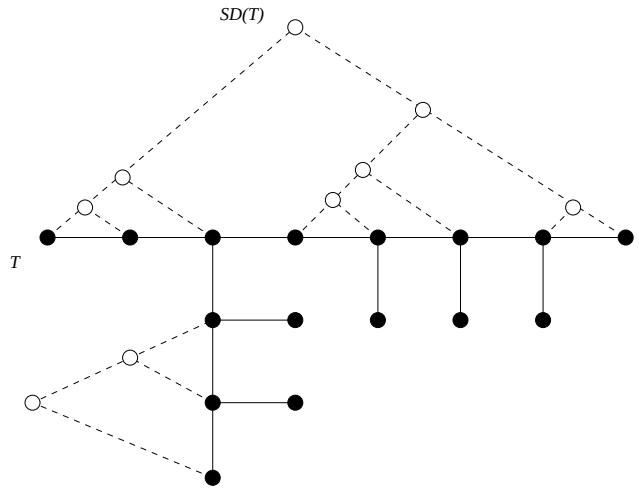

Figure 1: The spine decomposition $SD(T)$ of tree$T$. Black vertices and solid lines denote T; white vertices and dashed lines denote the search trees of $SD(T)$.

The *spine decomposition* of a tree has been used to implement efficient algorithms for computing the k-median [4] and length-constrained heaviest path [7] on trees. The decomposition of a tree T, denoted $SD(T)$, is specified as follows. The *weight* of node v in T, denoted $w(v)$ is the number of leaf nodes in the subtree of T rooted at v. If v is a leaf, its weight is 1. The weight of a tree T, denoted w_T, is defined as the weight of its root. Starting with the root, we repeatedly select the child with the greatest weight until we end up at a leaf. This root-to-leaf path is the first *spine*. We refer to this spine as the *top spine*. We now recursively compute the spine decompositions for all subtrees of T rooted at nodes incident to the first spine. Once this is complete, we construct a binary search tree for each spine whose leaves are the individual nodes on the spine. This tree is balanced with respect to node weight. This can be observed in Figure 1. From [4] we have that a spine decomposition can be constructed in $O(n)$ time, and that the depth of any vertex v in search tree S is $O(\log \frac{w_S}{w(v)})$, where w_s denotes the weight of the tree S. A corollary of this is the distance between nodes in a spine decomposition is $O(\log n)$ regardless of the initial diameter of T.

3 Dynamic trees for maintaining non-local properties

We now present our novel data structure for maintaining non-local properties in dynamic forests.

3.1 Edge insertions

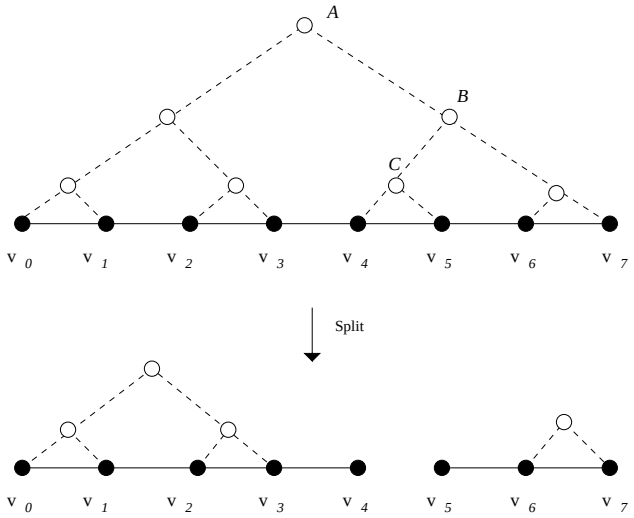

Figure 2: Splitting the spine at edge (v_4, v_5) requires that search tree nodes A, B, and C are deleted

We first present our method for handling edge insertions. Consider trees T_1 and T_2, with edge $e = (u, v)$ connecting some vertex in T_1 to the root v of T_2. Note that, without loss of generality, all trees in the forest are rooted binary trees, so vertex $u \in T_1$ must be of degree 2 or less. Let $T = T_1 \cup T_2$. Once e is inserted, $w(u)$ increases. This may alter the spine configuration of $SD(T)$. We can check if it does so by traversing the path from u to the root, making changes as necessary. Consider the case where the spine configuration is changed. Consider a spine $S = \{v_0, \ldots, v_k\}$ that has been disconnected at edge (v_i, v_{i+1}), with segment $S_1 = \{v_0, \ldots, v_i\}$ being appended to some other spine P, and the remainder $S_2 = \{v_{i+1}, \ldots, v_k\}$ being formed into a new, shorter spine (see Figure 2). To construct the search tree over these new spines, we use the subtrees of the search tree covering the vertices in S_1. We can identify them by tracing the path from v_{i+1} to the root. When we reach the first vertex t that has some $v_j (j \leq i)$ as a descendant, we delete all vertices from t to the root (Figure 2). Trees on the left side of the deleted vertex belong to T_1, and those on the right side belong to T_2.

We now present an algorithm to merge this collection of search trees while maintaining the depth property stipulated by the spine decomposition. We first present our method *mergeTree* (Algorithm 1) to merge two neighboring search trees U_1 and U_2 such that the depth of any node $u \in U = U_1 \cup U_2$ is $O(\log \frac{w_U}{w(u)})$. When we merge U_1 and U_2, if $w_{U_2} \ll w_{U_1}$ we connect U_2 to the root of U_1, resulting in a 3-ary tree U. Suppose now we merge U with a third tree U_3 that lies on the opposite side of U_1 from U_2. In this case, we simply ignore U_1 and merge as if it is not there. If $w_{U_3} \ll w_U$, the merged tree is 4-ary. However, we show that the degree of a 4-ary tree can never be increased via a merge operation.

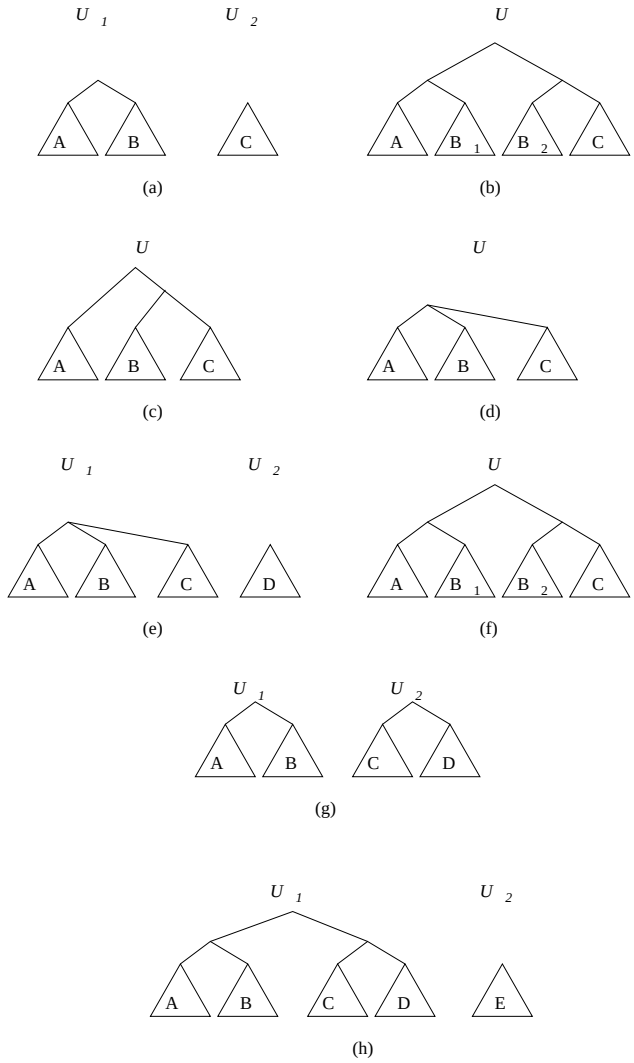

Figure 3: The various cases of *mergeTree* input.

Lemma 1. *Algorithm mergeTree results in a tree U such that for any vertex $u \in U$, the depth of u, denoted $d_U(u)$, is $O(\log \frac{w_U}{w(u)})$.*

Proof. In all cases, the depth of U_2 in U is at most 3. Hence, $d_U(U_2) = 3 \leq O(\log \frac{w_U}{w_{U_2}})$.

If the depth of a subtree T does not change during the merge operation, the depth condition is still satisfied. Since $w_{new} > w_{old}, O(\log \frac{w_{old}}{w_T}) = O(\log \frac{w_{new}}{w_t})$.

Consider the case where U_1 has root of degree 2 and $w_A \leq w_C$ (line 9). In this case (Figure 3b), $d_U(B_i) = d_{U_1}(B_i) = O(\log \frac{w_U}{w_{B_i}})$ holds for $i \in \{1, 2\}$, and $d_U(A) = d_U(C) = k \log(\frac{w_U}{w_C}) \leq k \log(\frac{w_U}{w_A}) = O(\log \frac{w_U}{w_A})$.

Consider the case where $w_A > w_C$ and $w_B \leq w_C$ (line 11). In this case (Figure 3c), $d_U(A) = d_{U_1}(A) = O(\log \frac{w_U}{w_A})$. For $B, d_U(B) = d_U(C) = k \log(\frac{w_U}{w_C}) \leq k \log(\frac{w_U}{w_B}) = O(\log \frac{w_U}{w_B})$.

For the case where w_C is small and is connected to the root of U_1 (Figure 3d), $d_U(A) = d_{U_1}(A) = O(\log \frac{w_U}{w_A}$ and $d_U(B) = d_{U_1}(B) = O(\log \frac{w_U}{w_B})$.

We now consider the case where the root of U_1 is of degree 3 (Figure 3e). If C is the "small" subtree and $w_C + w_D \geq w_A + w_B$ (line 20), since $w_U \geq 2(w_A + w_B)$ (Figure 3f), $d_U(A) = d_{U_1}(A) + 1 = k \log(\frac{w_{U_1}}{w_A}) + 1 = k \log(\frac{w_U}{w_A}) = O(\log \frac{w_U}{w_A})$.

A similar argument can be used for B. From the degree-2 case, we have that $w_C \leq w_A + w_B$. Therefore, $d(C) = 2 \leq O(\log \frac{w_U}{w_C})$.

If $w_C + w_D < w_A + w_B$, we construct T_1 and T_2 as in Figure 3g (lines 22-23). T_1 has a root of degree 2, so we have shown that the recursive call to *mergeTree* balances A and B correctly. The depth of C is at most 3, so $d(C) = 3 \leq O(\log \frac{w_U}{w_C})$.

When A is the "small" subtree (line 29), its depth does not change, and D is added to $B \cup C$ as normal. Likewise, when U_1 is a 4-ary tree (line 32), we ignore subtree A (Figure 3h) and merge as if it is a 3-ary tree (the depth of A is unchanged). □

Lemma 2. *Algorithm mergeTree results in a tree U that is 4-ary.*

Proof. *mergeTree* only alters the degree of the root of U. If U_1 has a root vertex of degree 2 or 3, at most one child is added by *mergeTree* (line 13). If U_1 has root of degree 4, we construct a special degree-3 case where subtree C (see Figure 3e) always has the least weight. Therefore, the case where D is appended to the root of U_1 (line 23) is never entered, and the degree of the root of U is not increased. □

To construct the binary search tree for the new spine we iteratively apply *mergeTree* to all tree fragments.

Lemma 3. *When an edge $e = (u, v)$ connecting T_1 and T_2 is inserted into the forest, the spine configuration of the new tree $T = T_1 \cup T_2$ can be updated $O(\log n)$ time.*

Proof. To check whether a change is necessary, all spines in the traversal from the insertion point to the root must be checked. This is easily done in $O(\log n)$ time by traversing the path P from v to the root of T_1. We now have k binary search trees to merge together. Note that every vertex on path P represents at most 2 search tree fragments; one for the spine segment that is to be concatenated with others, and one for the remainder (In Figure 2, $P = \{A, B, C, v_5\}$). Hence $k = O(\log n)$. *mergeTree* runs in $O(1)$ time. Iteratively applying it to all tree fragments takes $O(\log n)$ time. □

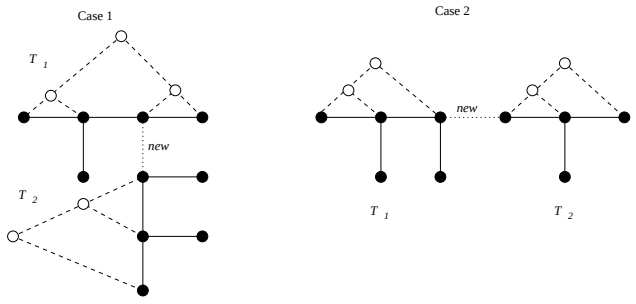

Figure 4: Edge insertion: Trees T_1 and T_2 are joined by edge *new*.

Now we consider the scenario where joining T_1 and T_2 with edge $e = (u, v)$ does *not* result in a change in spines. There are two cases (Figure 4) . In the first case, T_2 is connected to some internal vertex of u of T_1. Since $w(u)$ increases, we re-balance the search tree. Let u_L and u_R denote the spine vertices lying to the

Algorithm 1 *mergeTree*

Input: Search tree fragments U_1 and U_2. We assume without loss of generality that $w_{U_2} < w_{U_1}$ and U_2 lies to the right of U_1.
Output: Merged tree U

if $U_1.root$ is of degree 2 **then**
  Consider trees A, B, C as in Figure 3a.
  if $w_A \leq w_C$ **then**
    $B_1 \leftarrow$ the left subtree of B
    $B_2 \leftarrow$ the right subtree of B
    Join A with B_1 and B_2 with C as shown in Figure 3b and return.
  else if $w_A > w_C$ **then**
    if $w_B \leq w_C$ **then**
      Arrange A, B, C as in Figure 3c and return.
    else if $w_A > w_C$ and $w_B > w_C$ **then**
      Connect C to the root of U_1 (as in Figure 3d) and return.
    end if
  end if
else if $U_1.root$ is of degree 3 **then**
  Consider trees A, B, C, D as in Figure 3e.
  if $w_C \leq w_A$ **then**
    {The smallest subtree is on the side being merged}
    if $w_C + w_D \geq w_A + w_B$ **then**
      Arrange A, B, C, D as in Figure 3f and return.
    else if $w_C + w_D < w_A + w_B$ **then**
      Join A and B as in Figure 3g and denote this joined tree T_1
      Join C and D as in Figure 3g and denote this joined tree T_2
      Recurse: $mergeTree(T_1, T_2)$
    end if
  else if $w_C > w_A$ **then**
    {The smallest subtree is on the opposite side}
    Ignore A and merge as if U_1 is B joined with C (as in Figure 3e)
    Connect A to the root of the resulting tree and return
  end if
else if $U_1.root$ is of degree 4 **then**
  Consider trees A, B, C, D, E as in Figure 3h
  We know that w_A and w_D are small relatove to w_B and w_C
  Merge $U_1 = B \cup C \cup D$ with E
  Connect A to the root of the resulting tree and return
end if

left and right of u, respectively. We split spine $S = \{u_0, \ldots, u_L, u, u_R, \ldots, u_k\}$ at edges (u_L, u) and $u, u_R)$, and then re-join all the tree fragments via $mergeTree$. The vertex u is one such fragment, and if its weight has increased sufficiently the $mergeTree$ operations will place it closer to the root. The weight of the vertex that is connected to S is increased as well, so we repeat this operation for all spines all the way up to the top spine. The total number of search trees to merge is linear in the number of vertices on the path from u to the root of T_1. Since this length is $O(\log n)$, we update the search trees in $O(\log n)$ time.

In the second case, T_2 is connected to a leaf of T_1. This "extends" a spine of T_1 to include the top spine of T_2. We merge the two search trees via $mergeTree$. The weight of the vertex this spine is connected to is also increased. We rebalance the search trees by the method described for the first case. Again, this takes $O(\log n)$ time.

Corollary 1. *Edge insertion in a dynamic forest of trees with spine decompositions has time complexity* $O(\log n)$.

3.2 Edge deletion

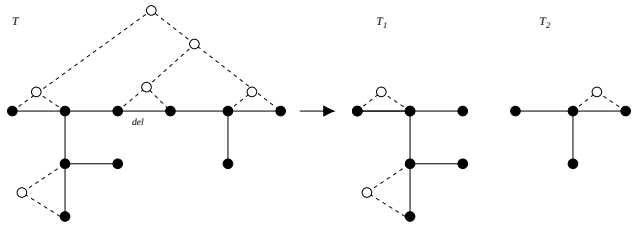

Figure 5: Edge removal: Tree T is split into T_1 and T_2 after edge *del* is removed.

Edge deletion in a dynamic forest of trees with spine decompositions is analogous to insertion (Figure 5). Therefore, we are able to use results from the previous section.

Lemma 4. *Edge deletion in a dynamic forest of trees with spine decompositions has time complexity* $O(\log n)$.

Proof. Suppose tree T is split into trees T_1 and T_2 via edge deletion. Without loss of generality, suppose that T_1 and T share the same root. Splitting the spine at the deletion point is analogous to splitting a spine during an edge insertion (Figure 5). Following this the spine configuration of T_1 may change since w_{T_2} is subtracted from the vertex on the cut-edge in T_1. From Corollary 1, we have that updating the spine decomposition takes $O(\log n)$ time. For T_2, we must re-assemble the search trees over the top spine. This also takes $O(\log n)$ time. □

For a given tree T in a dynamic forest, let $SD(T)$ denote our data structure for T outlined in this section.

4 Maximum subsequence queries in a dynamic forest

We now use the data structure $SD(T)$ presented in the previous section to solve the maximum subsequence problem for dynamic forests. Given a sequence S of real numbers, T_S denotes the sum of all elements in S and $|S|$ denotes the number of elements of A. Given sequences S_1 and S_2, the seqeunce $S_1.S_2$ denotes the concatenation of S_1 and S_2. Note that in this section, *weight* refers to the weight of an edge in the tree, and not the weight of a vertex in the spine decomposition.

Consider a sequence $S = \{a_0, \ldots, a_{n-1}\}$. S can be partitioned into 5 subsequences B, N_1, M, N_2, L, where B and L are the maximum prefix and suffix, respectively; M is the maximum subsequence; N_1 and N_2 are the intervals between B and M, and M and L, respectively. If the entire sequence S is the maximum subsequence, $M = S$ and all other subsequences are empty. If no maximum subsequence exists (this is

the case when all elements are negative), $N_1 = S$. If $S = B.N_1.M.N_2.F$, let P_S denote the sequence $\{T_B, T_{N_1}, T_M, T_{N_2}, T_F\}$. In [2], the authors demonstrate that given sequences S_1 and S_2, the sum of the maximum subsequences of $S_1.S_2$ and $P_{S_1}.P_{S_2}$ are identical.

In [15], Ruzzo et al present an $O(n)$-time algorithm to compute the maximum subsequence. For our $O(\log n)$ time query algorithm, we execute the Ruzzo algorithm on a sequence M of length $O(\log n)$. When computing $MS(u, v)$, the distance between u and v is $O(n)$ in T, but $O(\log n)$ in $SD(T)$. We construct M from the path through the spine decomposition. However, we first specify some notation. Suppose v is a search tree vertex in $SD(T)$. Let $v.leftmost$ define the spine node found by repeatedly traversing the leftmost node from v. We analogously define $v.rightmost$. The *cover* of vertex v is the spine segment from $v.leftmost$ to $v.rightmost$.

To compute $MS(u, v)$ in a dynamic forest, at each search node vertex v we store the sequence $S_v = \{T_B, T_{N_1}, T_M, T_{N_2}, T_F\}$ corresponding to the maximum subsequence of the edge weights taken from the path along the spine connecting $v.leftmost$ and $v.rightmost$.

Lemma 5. *Maintaining S_v for every search tree vertex v in a dynamic forest adds $O(1)$ overhead to $mergeTree$.*

Proof. $mergeTree$ modifies a search tree by either creating a new vertex and assigning it children or connecting a subtree . In the first case, when two search tree vertices v_1 and v_2 are joined at a new root v, we compute S_v by executing the algorithm of [15] on $S_{v_1}.S_{v_2}$ and obtaining $P_{S_{v_1}.S_{v_2}}$. Since $|S_{v_1}.S_{v_2}| \leq 10$, this takes $O(1)$ time. In the second case, if vertex v_1 is attached to v, we replace let $S_{new} = S_v.S_{v_1}$ and replace S_v with $P_{S_{new}}$. This also takes $O(1)$ time. □

If a vertex v is deleted during a spine splitting, its associated sequence information is discarded.

Corollary 2. *Edge insertion and deletion in a dynamic forest while maintaining S_v at every search tree vertex v takes $O(\log n)$ time.*

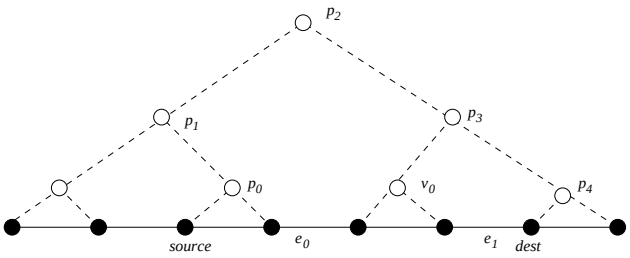

Figure 6: Path $P' = \{source, p_0, p_1, p_2, p_3, p_4, dest\}$ connects *source* and *dest*. Vertices $source, p_0, v_0$, and *dest* are chosen by our algorithm. Their covers are connected by edges e_0 and e_1.

We now present our query algorithm for $MS(u, v)$. Consider the path P of length $O(n)$ connecting u and v in T, and path P' of length $O(\log n)$ in $SD(T)$. To construct a sequence M, we choose search tree vertices in or adjacent to P' such that their covers include all vertices of P. We then examine consecutive vertices in this collection and insert between them the edge that connects their covers (Figure 6).

We choose these vertices as follows. The path P' traverses one or more search trees in $SD(T)$. Within each search tree we have a path $Q \subseteq P'$ connecting spine vertices *source* and *dest*. Assume without loss of generality that *source* is to the left of *dest*. We add a vertex $v \in Q$ if $v.leftmost$ is *source* or $v.rightmost$ is *dest*. Whenever such a vertex v is added, we remove all descendants of v that we have previously added. This is easy to do; we track the vertex $h \in Q$ of least depth. If v occurs before h in Q, we delete all vertices chosen so far. If v occurs after h, we delete all vertices chosen since h was visited. Once a vertex v with $v.rightmost = dest$ is added, we stop.

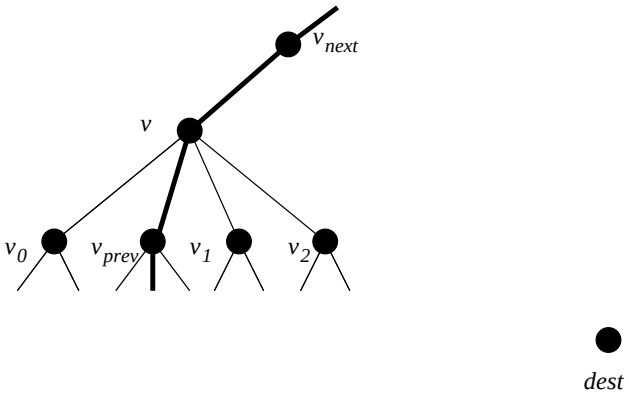

Figure 7: If v is not selected by our algorithm, then vertices v_1 and v_2 are.

If vertex v is *not* added, we examine adjacent vertices v_{prev} and v_{next} in Q. Assume v_{prev} is a child of v. By $mergeTree$, v can have up to 4 children. We choose the children of v that descend towards the final vertex in Q (Figure 7) and add them to our collection.

Lemma 6. *The aforementioned method allows us to construct a sequence M of length $O(\log n)$ whose maximum subsequence has the same sum as $MS(u, v)$.*

Proof. By our aforementioned method, we build a collection of vertices V ensuring that for all $v \in V$, v does not have an ancestor also in V. Hence, each spine vertex is only covered at most once.
To show that every spine vertex is covered, note that every vertex between u and v has an ancestor on the path P' in $SD(T)$. If that ancestor is not added to V, then its immediate children are.
For each vertex in P', we add a constant number of vertices to V (at most 3). The spine segments covered by successive vertices in V are separated by at most one edge (Figure 6). We obtain M by concatenating the sequences associated with each search node vertex and the connecting edges. Both these sequences have constant length, hence the length of M is $O(\log n)$. □

Corollary 3. *When S_v is stored at all search nodes in $SD(T)$, we can compute $MS(u, v)$ in $O(\log n)$ time.*

5 Conclusion

In this paper, we have shown how to maintain a dynamic forest of spine decompositions in $O(\log n)$ time per edge insertion and deletion. This improves on the result of [17], where insertion and deletion was achieved in only *amortized* $O(\log n)$ time. We used this new method to design a dynamic data structure with $O(\log n)$ query, insertion and deletion time for computing maximum subsequences between pairs of query nodes in a dynamic forest, which has been an open problem.

References

[1] Alstrup S, Holm J, Lichtenberg K, Thorup M, "Maintaining diameter, center, and median of fully-dynamic trees with top trees," *ACM Transactions on Algorithms, 2005*, 1:243-264

[2] Alves C, Caceres E, Song S, "BSP/CGM Algorithms for Maximum Subseqence and Maximum Subarray," *European PVM/MPI User's Group Meeting, 2004*, 3241:139-146

[3] Bengtsson F, Chen J, "Ranking k maximum sums," *Theoretical Computer Science, 2007*, 377:229-237

[4] Benkoczi R, Bhattacharya B, Chrobak M, Larmore L, Rydder W, "Faster algorithms for k-median problems in trees," *Proc. 28th International Symposium on Mathematical Foundations of Computer Science, 2003*, 2747:218-227

[5] Bentley J, *Programming Pearls*, Addison-Wesley, 1986

[6] Berman P, Bertone P, Dasgupta B, Gerstein M, Kao M, Snyder M, "Fast optimal tiling with applications to microarray design and homology search", *Journal of Computational Biology, 2004*, 11(4):766-85

[7] Bhattacharyya B, Dehne F, "Using spine decompositions to efficiently solve the length-constrained heaviest path problem on a tree," *Information Processing Letters, 2007*, to appear

[8] Chen K, Chao K, "Optimal algorithms for locating the longest and shortest segments satisfying a sum or an average constraint", *Information Processing Letters, 2005*, 96:197-201

[9] Cole R, Vishkin U, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time," *Algorithmica, 1988*, 3:329-346

[10] Frederickson G, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees," *SICOMP, 1997*, 26:484-538

[11] Gal A, Modica G, Jamil H, Eyal A, "Automatic ontology matching using application semantics," *AI Magazine, 2005*, 26:21-31

[12] Ginzinger S, Graupl T, Heun V, "SimShiftDB: Chemical-Shift-Based Homology Modeling", *Bioinformatics Research and Development, 2007*, 357-370.

[13] Henzinger M, King V, "Randomized dynamic graph algorithms with polylogarithmic time per operation", *Proc. 27th Symposium on Theory of Computing, 1995*, 519-527

[14] Kucherov G, Noe L, Ponty Y, "Estimating seed sensitivity on homogeneous alignments", *Proc. 4th IEEE Symposium on Bioinformatics and Bioengineering, 2004*, 387-394

[15] Ruzzo W, Tompa M, "A linear time algorithm for finding all maximal scoring subsequences," *Proc. 7th International Conference on Intelligent Systems for Molecular Biology, 1999*, 234-241

[16] Sleator D, Tarjan R, "A data structure for dynamic trees," *Journal of Computer and System Sciences, 1979*, 18:110-127

[17] Tarjan R, Werneck R, "Self-Adjusting Top Trees", *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms, 2005*, 813:822