ALGORITHMS FOR STATIC AND DYNAMIC PATH PROBLEMS IN TREES

by

Bishnu Bhattacharyya

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario

May, 2008

Your file Votre référence
ISBN: 978-0-494-40650-2
Our file Notre référence
ISBN: 978-0-494-40650-2

Table of Contents

List of Tables iv

List of Figures v

Abstract vii

Acknowledgements viii

Chapter 1 Introduction 1

1.1 Problems 1

1.2 Previous Work 1

1.3 Our contribution 2

Chapter 2 The Length-Constrained Heaviest Path for Trees 3

2.1 Problem Statement 3

2.2 Applications of LCHP 4

2.3 Previous Work 5

2.3.1 Hierarchical Decomposition of Trees 5

2.3.2 Review of Wu-LCHP 6

2.3.3 Review of Kim-LCHP 8

2.3.4 Review of the Kim-LNP 8

2.4 The Spine Decomposition of Trees 9

2.5 SLCHP: Our Novel Algorithm 11

2.5.1 *recurseLCHP* 12

2.5.2 *BSTNode* 13

2.5.3 Example 15

2.5.4 Analysis of SLCHP 18

Chapter 3 Fully Dynamic Trees **20**

3.1 Problem Statement 20

3.2 Previous Work 21

3.2.1 ST-trees 22

3.2.2 ET-trees 23

3.2.3 Top Trees 24

3.2.4 Self-Adjusting Top Trees 25

3.3 Applications 25

Chapter 4 DS-Trees: Our Solution for Fully Dynamic Trees **26**

4.1 Edge insertions 26

4.2 Edge deletion 32

4.3 Maximum subsequence queries in a dynamic forest 36

4.4 Other results 40

4.5 Conclusion 42

Chapter 5 Future Work **43**

Bibliography **44**

List of Tables

Table 2.1 The solution computed for each vertex of $SD(T)$ (Figure 2.6) by the algorithm SLCHP. 17

Table 4.1 A comparison of solutions to the fully dynamic forests problem 42

List of Figures

Figure 2.1 An instance of LCHP. Edges are labeled with $(weight, length)$ pairs and B is set to 0. 4

Figure 2.2 A tree (a) and the decomposition tree associated with its centroid decomposition (b). 6

Figure 2.3 The centroid decomposition tree associated with the instance of LCHP in Figure 2.1 7

Figure 2.4 Tree T. 10

Figure 2.5 The spine decomposition $SD(T)$ of the tree T in Figure 2.4. Black vertices and solid lines represent nodes and edges of T. White vertices and dashed edges represent the binary search trees. From this diagram, we see that all nodes in T are also in $SD(T)$. 11

Figure 2.6 The spine decomposition of the example tree in Figure 2.1. . . 15

Figure 2.7 A dependency tree for SLCHP for the nodes of $SD(T)$ in Figure 2.6. 17

Figure 3.1 Edge deletions (above) and insertions (below) in trees 20

Figure 3.2 An example of a rake operation. 21

Figure 3.3 An example of a compress operation. 21

Figure 3.4 An example of a solid path in a tree. 23

Figure 4.1 Splitting the spine at edge (v_4, v_5) requires that search tree nodes A, B, and C are deleted 27

Figure 4.2 The various cases of *mergeTree* input. 28

Figure 4.3 Edge insertion: Trees T_1 and T_2 are joined by edge *new*. . . . 31

Figure 4.4 Edge removal: Tree T is split into T_1 and T_2 after edge (u, v) is removed. Vertex A is a breakpoint of T_1; the spine must be split at this point, as the child spine has more leaves than the rest of the topmost spine. 32

Figure 4.5 When (u, v) is removed, if there is a spine S_2 below u it must be merged with the segment of S_1 that is in T_1. 34
Figure 4.6 Since P_{new} passes through the root of the search tree, for every b_i there exists a v_i as illustrated in the diagram. 35
Figure 4.7 If b_i is split before b_j, we ensure that when splitting b_j, the root of the search tree is to the right of P_{new}. Therefore, Y_j cannot include any of the bolded section of P_{new}. 37
Figure 4.8 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. 39
Figure 4.9 If v is not selected by our algorithm, then vertices v_1 and v_2 are. 39

Abstract

This thesis is an investigation into two separate problems for trees.

The first is the length-constrained heaviest path problem for trees (LCHP). Given a tree T with weight function w, length function l, and threshold B, we seek the path of maximum total weight whose total length is bounded by B. We review the solutions of Wu et al (which runs in $O(n \log^2 n)$ time) and Kim (which runs in $O(n \log n \log \log n)$ time) before presenting algorithm $SLCHP$, which solves LCHP in $O(n \log n)$ time. This also compares favorably to Kim's solution for the longest nonnegative path (LNP). This is an instance of LCHP where $w(e) = 1$ for all edges e and $B = 0$. Kim provides a $O(n \log n)$ time algorithm that solves LNP on fixed-degree trees; SLCHP executes on trees of arbitrary degree.

The second problem is that of maintaining tree attributes through dynamic edge insertions and deletions. This is known as a the fully dynamic trees problem. Typical tree attributes include tree diameter, maximum subsequence between vertices, and the minimum vertex on a path. Another common operation on dynamic trees is adding a constant value to the weight of all edges on a given path. We present DS-trees, which are able to perform all these operations, with worst-case $O(\log n)$ time edge insertion and deletion. This is comparable to previous solutions to the dynamic trees such as ST-trees, ET-trees, and top trees.

Acknowledgements

First and foremost, I'd like to thank my supervisor Dr. Frank Dehne for always being available to answer questions and for being patient as I jumped from topic to topic before finally settling on one.

I am also grateful to Dr. Pat Morin and Dr. Amiya Nayak for agreeing to sit on my defense committee, and to Dr. Doron Nussbaum for chairing.

I would like to thank Dr. Evangelos Kranakis for so wonderfully teaching Wireless Networks and Mobile Computing, the class where I was first introduced to the problem of maintaining data in dynamic graphs.

I greatly appreciated the funding I received from the Faculty of Graduate Studies over the course of my studies.

Not many sons are able to cite their fathers in their thesis, but even beyond that, without the love and support shown by both my parents, this thesis would not have been written. I also want to thank Nihar for being so hospitable whenever I visited her in Montreal during the sleepy summer of 2007.

Finally, I'd like to thank every one of my friends, colleagues, the administrative staff, and the faculty at Carleton University for making these last 2 years so enjoyable for me, and all my friends back in Vancouver for making me feel so welcome whenever I returned home.

Chapter 1

Introduction

1.1 Problems

This thesis investigates two general problems. The first is the length-constrained heaviest path problem for trees (LCHP), and the second is the problem of maintaining data in fully dynamic trees.

The length-constrained heaviest path problem (first discussed in [40]) accepts as input a tree T, edge weight and length functions w and l, and threshold B, and returns the path of maximum total weight such that the total length is constrained by B. There have been numerous proposed solutions to LCHP [40, 25], including ones for specific sub-problems of LCHP [26].

In the dynamic trees problem, a forest of trees is maintained over edge insertions and deletions. Because we allow edge deletions, we say the forest is *fully* dynamic. Throughout these operations, certain values are computed and updated in each dynamic tree (for example, tree diameter). Alternatively, some applications of dynamic trees require values to be combined with the tree (for example, adding a constant value to all edges in a path). Dynamic trees are used, for example, by solutions to the maximum flow problem [22, 37] and by dynamic graph algorithms [5, 16, 39, 23].

1.2 Previous Work

The first solution to LCHP was given by Wu et al in 1999 [40], with time complexity $O(n \log^2 n)$. In [25] Kim refines the solution to run in $O(n \log n \log\log n)$ time. Additionally, Kim developed an $O(n \log n)$-time algorithm for the special case of finding a longest nonnegative path in a constant degree tree.

Solutions to the dynamic trees problems include ET-trees [23], ST-trees [32, 33], and top trees [5]. ET-trees are relatively simple data structures, but they are only

suitable for maintaining subtree-based attributes of dynamic trees. Top trees and ST-trees are more robust in that they can also handle path-based attributes. However, queries on these data structures cause them to modify themselves. This does not allow for parallel queries to be efficiently executed, or for users that are restricted to read privileges to run queries. SinceAdditionally, it can be cumbersome to design algorithms for top trees and ST-trees since they can represent the same tree in many different ways.

1.3 Our contribution

In Chapter 2, we present an algorithm $SLCHP$ that solves the LCHP for trees in $O(n \log n)$ time, a factor $\log \log n$ improvement over the Kim algorithm. Our method also improves the Kim algorithm for the longest nonnegative path in that we can handle trees of arbitrary degree within the same time bounds.

In Chapter 4 we present *DS-trees*, our own data structure that supports edge insertions and deletions in $O(\log n)$ time and provides efficient methods for maintaining tree diameter, maximum subsequence, the minimal edge on a given path, as well as adding a constant value to all edges on a given path. DS-trees decompose their input trees into individual paths, but only alter their internal structure on edge insertions and deletions. DS-trees also unambiguously partition their underlying trees into a set of spines, and therefore retain the structure of their underlying tree to some degree; this makes designing novel query algorithms for DS-trees a more intuitive process.

Chapter 2

The Length-Constrained Heaviest Path for Trees

2.1 Problem Statement

Consider an undirected tree $T = (V, E)$, and define functions $w(e)$ and $l(e)$ to be the *weight* and *length* of each edge $e \in E$, respectively. For any path $path(u, v)$ between vertices u and v, we define the path weight $w(path(u, v)) = \sum_{e \in path(u,v)} w(e)$ and path length $l(path(u, v)) = \sum_{e \in path(u,v)} l(e)$. The *length-constrained heaviest path* for T is then defined as follows [40]:

Definition 1. *Given a tree $T = (V, E)$ with edge weights $w(e)$ and edge lengths $l(e)$, and a real number B, then the* **length-constrained heaviest path** *(LCHP) for T is the path P such that*

$$w(P) = \max_{u,v \in V} \{w(path(u, v)) | l(path(u, v)) \leq B\}$$

and $hw(T, w, l, B)$ denotes the weight of the length-constrained heaviest path for T.

LCHP can be used to solve network design problems on tree networks, where the edge weights represent bandwidth and the lengths represent link costs [40]. A special case of LCHP, called the *longest nonnegative path* (LNP), has applications in computational molecular biology and bioinformatics [4]. An example of LCHP with $B = 0$ is shown in Figure 2.1.

Definition 2. *Given a tree $T = (V, E)$ with arbitrary edge weights $w(e)$, the* **longest nonnegative path** *(LNP) for T is the path P with the greatest number of edges such that $w(P) \geq 0$.*

The first solution to LCHP was presented by Wu et al in [40]. Their algorithm had time complexity $O(n \log^2 n)$. Since then, Kim has presented two refinements to their algorithm. The first solves LNP for trees with fixed degree vertices in $O(n \log n)$ time

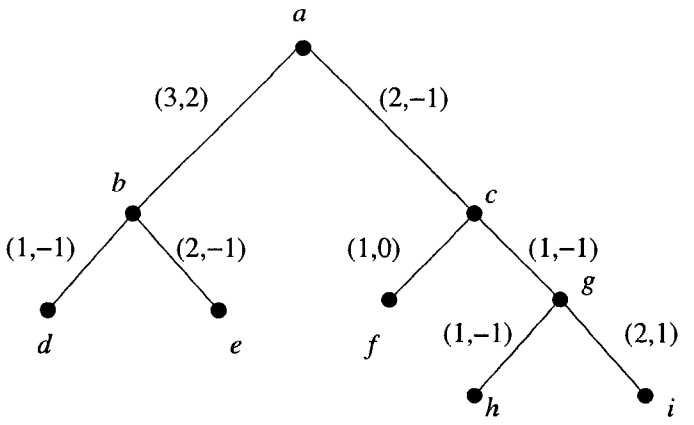

Figure 2.1: An instance of LCHP. Edges are labeled with $(weight, length)$ pairs and B is set to 0.

[26]. The second solves LCHP in $O(n \log n \log \log n)$ time [25]. We improve on all these in results in 2.5 with SLCHP, an $O(n \log n)$ time algorithm that solves LCHP for arbitrary trees. SLCHP is presented in Section 2.5.

2.2 Applications of LCHP

The length-constrained heaviest path problem was initially conceived as a problem in network design [40]. For instance, when choosing paths to construct in a potential network, we may want to maximize bandwidth while constraining the total cost.

More generally, the length-constrained heaviest segment is frequently computed in the area of bioinformatics and computational molecular biology. For instance, it is used to identify GC-rich regions of the DNA strand. In all organisms, GC base pairs compose between 25 and 75% of the DNA strand. Nekrutenko et al [29] showed that genes are predominantly found in these GC-rich regions. Hence, the identification of such regions is important in the area of genome annotation and comparative genomics. To measure the GC-richness of a region, Huang [24] used the expression $x - pl$, where x is the GC count of a region, l is the region's length, and p is a positive penalty ratio. Essentially, each G or C nucleotide is "rewarded" $(1 - p)$, and each A or T nucleotide is "penalized" by p. The length constraint is specified by a cutoff value L used to eliminate extremely short optimal regions.

In multiple sequence alignments, conserved regions (subsequences that occer in each sequence) are strong candidates for functional elements. Stojanovic et al [34, 35] present several methods for analyzing a previously computed multiple sequence alignment to find highly conserved regions. These methods are based around assigning a positive numerical score to each column of the alignment, and searching for sequences of columns with high cumulative scores. Since all scores are positive, to avoid reporting the entire alignment as a conserved region, we now constrain the *maximum* length of the conserved sequence.

In [28], Lin et al present a $O(n \log L)$ time algorithm for computing the length-constrained heaviest segment, where L is the minimum allowed length.

2.3 Previous Work

Because Wu's algorithm (Wu-LCHP) and SLCHP share some structural similarity, we begin with an outline of Wu's. For the sake of completeness we also briefly talk about the structure of Kim's algorithms for LCHP and LNP (Kim-LCHP and Kim-LNP, respectively). Before we start, however, we introduce the concept of tree decompositions, which are used by both Wu-LCHP algorithm and SLCHP.

2.3.1 Hierarchical Decomposition of Trees

Definition 3. *A general* **decomposition** *of tree T, denoted $D(T)$, is a collection of subtrees of T such that*

1. *$T \in D(T)$*
2. *For all $T_1, T_2 \in D(T)$ either T_1 and T_2 are disjoint, or one is strictly contained in the other.*

The **depth** *of a decomposition is the maximum cardinality of $H \subseteq D(T)$ such that*

$$H = \{T_1, T_2, \ldots, T_k | T_1 \subset T_2 \subset \ldots \subset T_k\}.$$

It is important to define the depth of a tree decomposition, since it directly influences the running time of our algorithm. A common tree decomposition that is used

is the *centroid decomposition* [14].

Definition 4. *A* **centroid** *of a tree T is a vertex x whose removal results in a set of subtrees $T_1, \ldots, T_k$ such that for all $1 \leq i \leq k, |T_i| \leq |T|/2$ (where $|T|$ denotes the number of vertices in T).*

Any tree T has at least one centroid [14]. Let $T(v)$ denote the set of subtrees formed by removing vertex v from T. A centroid decomposition $CD(T)$ is formed by starting with $\{T\}$, finding its centroid x, and adding $T(x)$ to the set of components. This procedure is applied on each tree in $CD(T)$ until the components added are single vertices. The depth of $CD(T)$ is $O(\log n)$ [14]. Note that a centroid decomposition can be represented by a (rooted) tree where each node corresponds to a subtree of T. This is known as the *decomposition tree*. The depth of this tree is equal to the depth of $CD(T)$. This is illustrated in Figure 2.2.

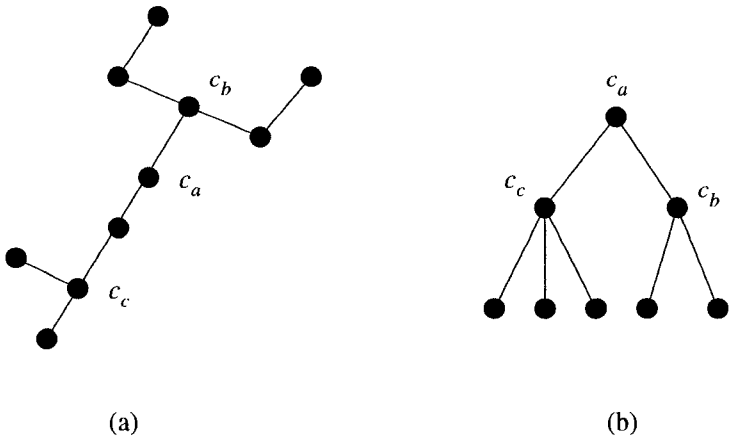

Figure 2.2: A tree (a) and the decomposition tree associated with its centroid decomposition (b).

2.3.2 Review of Wu-LCHP

Wu-LCHP accepts as input a tree T and constructs a decomposition tree of $CD(T)$. It then processes the decomposition tree in a bottom-up fashion, starting with the leaves, which correspond to individual vertices of T. The centroid decomposition tree for the example in Figure 2.1 can be seen in Figure 2.3.

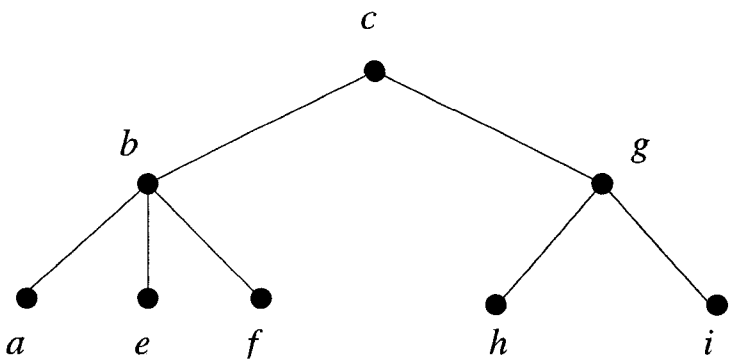

Figure 2.3: The centroid decomposition tree associated with the instance of LCHP in Figure 2.1

If v is a vertex in the decomposition tree, let T_v represent the subtree of T it represents. For each vertex v of the decomposition tree, the algorithm computes the local solution $hw(T_v, w, l, B)$ as well a list of all paths in T_v terminating at its root that is sorted by length. This list is denoted L_v.

When T_v is a leaf of T, this is trivial to compute. When v has children $v_0, \ldots, v_k$ in the decomposition tree, the situation is more complex. Wu-LCHP finds the best solution of LCHP passing through the root of T_v. This solution is then checked against the solutions for $T_{v_0}, \ldots, T_{v_k}$, and the best one is passed upwards. This is done as follows. For each list L_{v_i}, the path from the root of T_{v_i} to the root of T_v is appended to each list element. L_{v_i} remains in sorted order. Next, all such lists L_{v_i} are merged together into L_v, which is then re-sorted by length.

For every path $P \in L_v$, the associated paths P_0 and P_1 are defined as follows.

$$P_0 = \max\{w(Q) | Q \in L_v, l(Q) \leq l(P)\}$$

$$P_1 = \max\{w(Q) | Q \in L_v, l(Q) \leq l(P) \text{ and } Q, P_0 \text{ are in different subtrees of } T_v\}$$

Once this has been computed, the algorithm selects for every path $P \in L_v$ path Q of greatest length such that $l(Q) + l(P) \leq B$. Then, depending on which subtree of T_v P is in, either path PQ_0 or PQ_1 is the length-constrained path of greatest weight containing path P. This is computed for every path P in L_v, and the path of maximum weight is stored.

Constructing the centroid decomposition of T takes $O(n)$ time, sorting the list L_v takes $O(n \log n)$ time, and scanning L_v to find the best path running through the root of T_v takes $O(n)$ time. Suppose T has centroid c. Let $Q(n)$ denote the time complexity of the Wu algorithm on input of size n. Hence, $Q(n) = O(n \log n) + \sum_{i \in child(c)} Q(|T_i|)$. Since $|T_i| \leq \frac{n}{2}$ and $\sum_{i \in child(c)} |T_i| = n - 1$, $Q(n) = O(n \log^2 n)$ [40]. The requirement that L_v be re-sorted at every step is a bottleneck that increases the run-time of the algorithm by a factor of $\log n$.

2.3.3 Review of Kim-LCHP

Kim-LCHP also constructs a centroid decomposition, but it first transforms T into a binary tree T'. Thus, any centroid of T' has at most 3 children. Kim-LCHP is able combine these three solutions in $O(n \log \log n)$ time, which reduces the run-time to $O(n \log n \log \log n)$ [25].

2.3.4 Review of the Kim-LNP

The Kim-LNP algorithm accepts a fixed-degree tree T and function $w(e)$, which assigns weights to edges in T. It then finds the path P that has the maximum number of edges with $\sum_{e \in P} w(e) \geq 0$. Again, a centroid decomposition of T is processed bottom-up. This is a special case of LCHP where the weight function $w_{LCHP}(e) = 1, lengthfunctionl_{LCHP} = -w(e)$, and $B = 0$.

For every subtree T_i formed by removing the current centroid c, Kim-LNP computes the path c to the centroid maximizing $w(P)$ for every possible path length. Since length is defined as number of edges, the maximum possible length is $|T_i| - 1$. Once this has been computed, so-called *dominated paths* are eliminated from this list. Path P is dominated by path Q if Q is of greater length and weight. Once these paths are eliminated, the remainder are stored in an array L_i. These arrays are then scanned to find the longest nonnegative path containing c in the same manner as in the Wu-LCHP. When this path is found, it is compared to the solutions passed up from below, and the best one is retained. All this is done in $O(n)$ time, and hence the running time of the entire algorithm is $O(n \log n)$ [26].

2.4 The Spine Decomposition of Trees

A major weakness of the centroid decomposition is that there is no control on the path between a centroid and centroids on the level below or above in the decomposition - for example, vertices c_a and c_b in Figure 2.2. SLCHP utilizes the spine decomposition of a tree, first introduced by Benkoczi et al in [8].

A spine decomposition is built around *spines*, or paths from the root of a tree to a leaf. First, without loss of generality assume that T is a rooted binary tree. If T has no root, we can arbitrarily assign one. If T is not binary, we can transform it into a binary tree by adding $O(n)$ nodes and zero-length, zero-weight edges [36]. This process is known as ternarization. This transformed tree is denoted by T'. We denote the spine decomposition of a tree T with $SD(T)$.

Lemma 1. *Suppose (T, w, l, B) is an instance of LCHP, where T is an arbitrary tree. Let T' denote the rooted binary transformation of T. Given vertices $u, v \in T'$, we define functions w', l' as follows:*

$$w'(u,v) = \begin{cases} w(u,v) & \text{if } (u,v) \text{ is an edge in } T \\ 0 & \text{otherwise} \end{cases}$$

$$l'(u,v) = \begin{cases} l(u,v) & \text{if } (u,v) \text{ is an edge in } T \\ 0 & \text{otherwise} \end{cases}$$

Then, $hw(T, w, l, B) = hw(T', w', l', B)$.

Proof. Since all edges in $T' \setminus T$ have 0 weight and 0 length, any path in T has a corresponding path of identical weight and length in T', and vice-versa. □

For the remainder of this section, we assume T is a binary tree with n nodes and root r_T. $T(v)$ the subtree of T rooted at v.

The number of *descended leaves* from vertex v, denoted $N_l(v)$, is the number of leaf nodes in T that have v as an ancestor. The spine $\pi(r_T, l) = \{v_0 = r_T, v_1, \ldots, v_k = l\}$ is chosen such that if v_i is a spine node with children u_i and v_{i+1}, then $v_{i+1} \in \pi(r_T, l)$ if and only if $N_l(v_{i+1}) \geq N_l(u_i)$. In other words, the next edge in a spine is always

chosen to be the one with the most leaves descended from it. Next, we recursively compute the spine decompositions for each subtree $T(u_i)$ rooted at a node u_i adjacent to $\pi(r_T, l)$.

However, in certain trees, a spine can be of length $O(n)$. Consider an algorithm that processes $SD(T)$ bottom-up. Gathering information from that many subtrees in one level of the recursion is cumbersome and impractical. This is circumvented by building a binary search tree on top of every spine. The leaves of the BST are nodes on the spine. To build the BST with root x on spine $\pi = \{v_0, \ldots, v_k\}$, denote $\lambda(v_i) = N_l(T(u_i))$, where u_i is the child of v_i that is not in π. If u_i does not exist, $\lambda(v_i) = 1$. Compute m such that $|\sum_{(i=0)}^{m} \lambda(v_i) - \sum_{(i=m+1)}^{k} \lambda(v_i)|$ is minimized, and then recursively compute the BSTs for node x_1 with spine $\{v_0, \ldots, v_m\}$ and node x_2 with spine $\{v_{m+1}, \ldots, v_k\}$. Minimizing this difference balances the search tree by weight. x_1 and x_2 are then assigned as the left and right child of x, respectively. Consequently, a spine node with many descended leaves will be closer to the root of the binary search tree. The spine decomposition of the tree T in Figure 2.4 is shown in Figure 2.5. It is important to note that $SD(T)$ includes every vertex and edge in T.

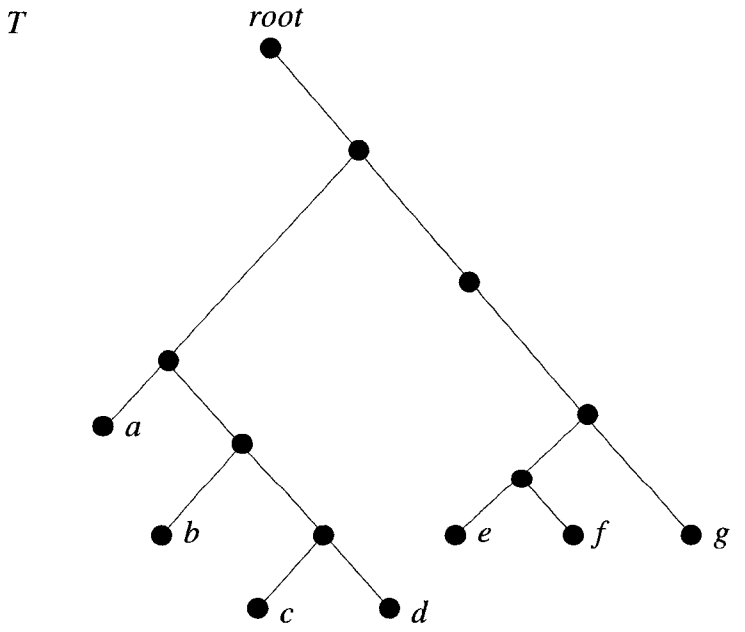

Figure 2.4: Tree T.

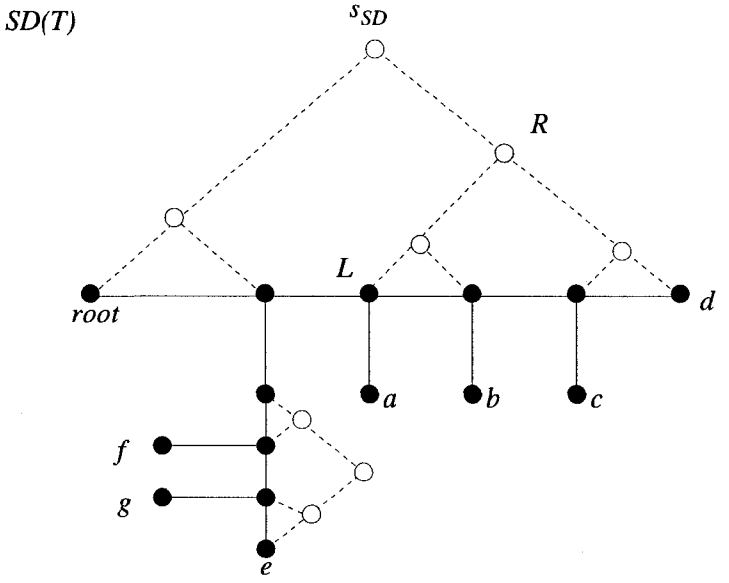

Figure 2.5: The spine decomposition $SD(T)$ of the tree T in Figure 2.4. Black vertices and solid lines represent nodes and edges of T. White vertices and dashed edges represent the binary search trees. From this diagram, we see that all nodes in T are also in $SD(T)$.

$SD(T)$ can be computed in $O(n)$ time. The resulting decomposition tree is of height $O(\log n)$ and has $O(n)$ vertices [8]. Note that the height of this tree is independent of the height of T. We denote s_{SD} as the root of the search tree of the first spine in $SD(T)$. s_{SD} is the root of the decomposition tree of T.

2.5 SLCHP: Our Novel Algorithm

Our algorithm is presented in three parts. For readability, we compute only the weight of the heaviest path. However, it is a simple modification to compute the path itself, as well.

Initially, *LCHPsolve* (Algorithm 1) pre-processes T by converting it to a rooted binary tree T' and computing the spine decomposition $SD(T')$. In otherwords, it computes the transformation illustrated in Figure 2.4 and Figure 2.5. It then initiates the recursion by calling *recurseLCHP* (Algorithm 2). However, before we describe *recurseLCHP*, we need some notation:

- If v is a node of a binary search tree, $left(v)$ is the left child of v. $right(v)$ is

defined analogously.

- If v is a node of a binary search tree, $leftmost(v)$ defines the spine node found by repeatedly traversing the left edge from v. $rightmost(v)$ is defined analogously. If v is a spine node, $leftmost(v) = rightmost(v) = v$. In Figure 2.5, $leftmost(s_{SD}) = root$ and $rightmost(s_{SD}) = d$. We adopt the convention that leftmost always points towards the head of the spine.

- When discussing $recurseLCHP$ (Algorithm 2) and $BSTnode$ (Algorithm 3), we may refer to rooted binary tree T' as T. The notation can be simplified since both of these algorithms are oblivious as to whether T was pre-processed or not.

We now outline algorithms 2 and 3. $recurseLCHP$ solves LCHP for the subtree of the **decomposition tree** of $SD(T)$ that is denoted by a node x in the tree.

2.5.1 $recurseLCHP$

When processing $SD(T)$, there are three cases to consider. The first case is when the current node x being processed is a leaf of T. In Figure 2.5, these correspond to vertices a, b, c, d, e, f, and g. The second case is where x is not a leaf, yet is still a spine vertex. This corresponds to the remaining black vertices in Figure 2.5. The final case is when x is a search node of $SD(T)$, or a white node in Figure 2.5.

In addition to solving LCHP, $recurseLCHP$ also returns two length-sorted lists of paths in the subtree. One list is of all paths that terminate at $leftmost(x)$, the other is of all paths that terminate at $rightmost(x)$. These paths are denoted X and Y, respectively. In the first case, where x is a leaf of T, these lists are empty and the solution to LCHP is $-\infty$ ($recurseLCHP$, line 6).

In the second case, where x is a (non-leaf) spine node, the situation is more complex. If $deg(x) = 2$ we can treat x as if it is a leaf of T. Otherwise, we must first recurse on the subtree of $SD(T)$ rooted at node y, the child of x that is **not** in the current spine. We take the list of paths returned and append edge(x, y) to all of them, adjusting path weight/length accordingly (the list remains sorted) ($recurseLCHP$, lines 13-15). If any of these new paths are a better solution to LCHP than the one

returned by the recursive call, we record that ($recurseLCHP$, line, 16). Note that in these cases the left list and the right list will be identical.

2.5.2 *BSTNode*

The most complicated case is the third one, when x is a node in a binary search tree above a spine. This is handled by *BSTnode* (Algorithm 3).

Definition 5. *If v is a node in a binary search tree in $SD(T)$, the subtree of T that is formed by taking the spine segment from $leftmost(v)$ to $rightmost(v)$ and all spines incident to it is the subtree of T that is* **covered** *by v, denoted T_v. In Figure 2.5, R covers the spine segment from L to d, as well as leaf nodes a, b, and c.*

This is the only case where x is not a node in the original tree T. We solve LCHP for the subtree T_x of T. After computing LCHP for $left(x)$ and $right(x)$ (denoted L and R, respectively), we look for the maximum length-constrained path in T_x passing through edge $e = (rightmost(left(x)), leftmost(right(x))$. We first append e to all the paths in the list $R.X$ and merge with $L.Y$. This results in a list of paths terminating at vertex $w = rightmost(left(x))$.

To compute the best path containing e, we first check the current best solution against all paths in T_x terminating at w (*BSTnode*, line 13). We then check all paths that contain e using the method of [41]. For each path P that terminates at vertex w we first compute the path of maximum weight Q such that $w(Q) \leq w(P)$ for both the left and right subtree of T_x descended from w (*BSTnode*, lines 14-17). Thus, the path starting at some vertex v and passing through e can be quickly calculated by first finding the vertex u such that $path(u, v)$ is the path of greatest length passing through u, v, and w (*BSTnode*, line 20). We then replace the segment $path(u, v)$ with the heaviest path of lesser or equal length in the appropriate subtree (*BSTnode*, lines 23-26). This path is guaranteed to be the heaviest path passing through w and v obeying the length constraint.

Once the solution for the T_x has been computed, we construct a length-sorted list of paths terminating at $leftmost(x)$ and $rightmost(x)$ and pass the solution upwards (*BSTnode*, lines 27-29).

Algorithm 1 *LCHPsolve*

Input: Tree T, weight function w, length function l, threshold B
Output: $soln \leftarrow hw(T, w, l, B)$
if T is not a rooted binary tree **then**
  Convert T to a rooted binary tree T'. Create w' and l' accordingly
end if
Construct $SD(T')$
Output $recurseLCHP(SD(T'), s_{SD}, w', l', B).soln$

Algorithm 2 *recurseLCHP*

Input: Spine decomposition $SD(T)$, node x in $SD(T)$, weight function w, length function l, threshold B
Output: $soln \leftarrow hw(T, w, l, B)$, $X \leftarrow$ length-sorted array of paths in T ending at $leftmost(x)$, $Y \leftarrow$ length-sorted array of paths in T ending at $rightmost(x)$
$X_T := []$
$Y_T := []$
if x is a leaf of T **then**
  Return $(-\infty, X_T, Y_T)$
else if x is a spine node **then**
  if $deg(x) = 2$ **then**
    Return $(-\infty, X_T, Y_T)$
  else
    $y :=$ The child of x that is in the spine below x in $SD(T)$
    $z := s_{SD}$ of the spine decomposition of the subtree of T rooted at y
    $S := recurseLCHP(SD(T), z, w, l, B)$
    Append edge (x, y) to all paths in $S.X$ and adjust path weight/length accordingly
    Insert path $P = ((x, y))$ into $S.X$
    $lsoln := \max_{i \in S.X}\{S.X[i].weight | S.X[i].length \leq B\}$
    Return $(\max\{lsoln, S.soln\}, S.X, S.X)$
  end if
else if x is a node on a binary search tree **then**
  Return $BSTnodes(SD(T), x, w, l, B)$
end if

2.5.3 Example

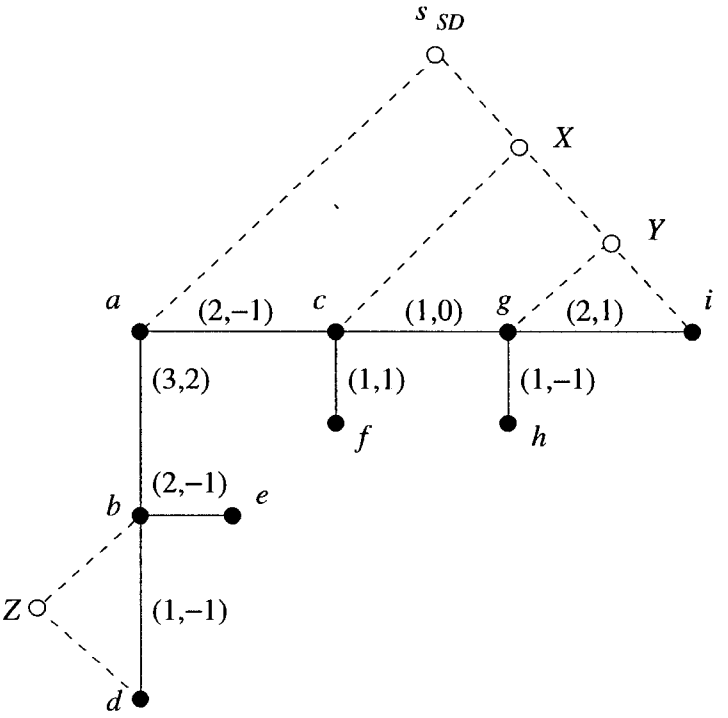

Figure 2.6: The spine decomposition of the example tree in Figure 2.1.

Before analyzing correctness and running-time, we present an example of SLCHP on the tree T shown in Figure 2.1. Algorithm SLCHP initially constructs the spine decomposition $SD(T)$, which is illustrated in Figure 2.6. SLCHP processes $SD(T)$ bottom-up, terminating when vertex s_{SD} is processed. The dependencies for each node in $SD(T)$ is illustrated in Figure 2.7.

At each node we compute a $(solution, left, right)$ tuple. Initially, the solution computed at vertices d, e, f, h, and i are $(-\infty, [], [])$. That is, no solution exists and the list of paths is empty. For easier reference, the results for each node are tabulated at Table 2.1.

From Figure 2.7, the next nodes to be processed are b, c, and g. At b, there is only one path in the subtree being processed (eb). Since it has negative length, it becomes the local solution: $(2, [eb], [eb])$. For spine nodes, the leftmost and rightmost nodes are the same, and so the same list is returned twice. Similarly, at g the solution is

Algorithm 3 *BSTnode*

Input: Spine decomposition $SD(T)$, binary search tree node x, weight function w, length function l, threshold B
Output: $soln \leftarrow hw(T, w, l, B)$, $X \leftarrow$ length-sorted array of paths in T ending at $leftmost(x)$, $Y \leftarrow$ length-sorted array of paths in T ending at $rightmost(x)$
$P := []$
$X_T, Y_T := []$
$L := recurseLCHP(SD(T), left(x), w, l, B)$
$R := recurseLCHP(SD(T), right(x), w, l, B)$
Let lt and rt denote the left and right children of x, respectively
$e := edge(rightmost(lt), leftmost(rt))$
Append edge e to all paths in $R.X$, and merge $L.Y$ and $R.X$ into list P
Insert the path consisting of the single edge e into P such that the list remains in sorted order
$lsoln := max\{L.soln, R.soln\}$
Let n denote the number of elements in P
$lsoln := \{lsoln, \max_{1 \le i \le n}\{P[i].weight | P[i].length \le B\}\}$
for all i such that $1 \le i \le n$ **do**
    $best[i] := P[\alpha]$ such that $P[\alpha].weight = max_{1 \le j \le i}\{P[j].weight\}$
    $otherbest[i] := P[\beta]$ such that $P[\beta].weight = max_{1 \le j \le i}\{P[j].weight\}$ and path $P[\beta]$ and $P[\alpha]$ do not share any edges. If no such α exists, $otherbest[\beta] := -\infty$
end for
$j := n$
for $i = 1$ to n **do**
    while $P[i].length + P[j].length > B$ **do**
        $j := j - 1$
    end while
    if $j < 1$ **then**
        break
    end if
    if $P[i]$ and $best[j]$ share one or more edges (and thus cannot be concatenated) **then**
        $lsoln := \max\{lsoln, P[i].weight + best[j].weight\}$
    else
        $lsoln := \max\{lsoln, P[i].weight + otherbest[j].weight\}$
    end if
end for
Append $path(leftmost(lt), leftmost(rt))$ to all paths in $R.X$, and merge $R.X$ and $L.X$ into X_T.
Append $path(rightmost(lt), rightmost(rt))$ to all paths in $L.Y$, and merge $L.Y$ and $R.Y$ into Y_T
Return $(lsoln, X_T, Y_T)$

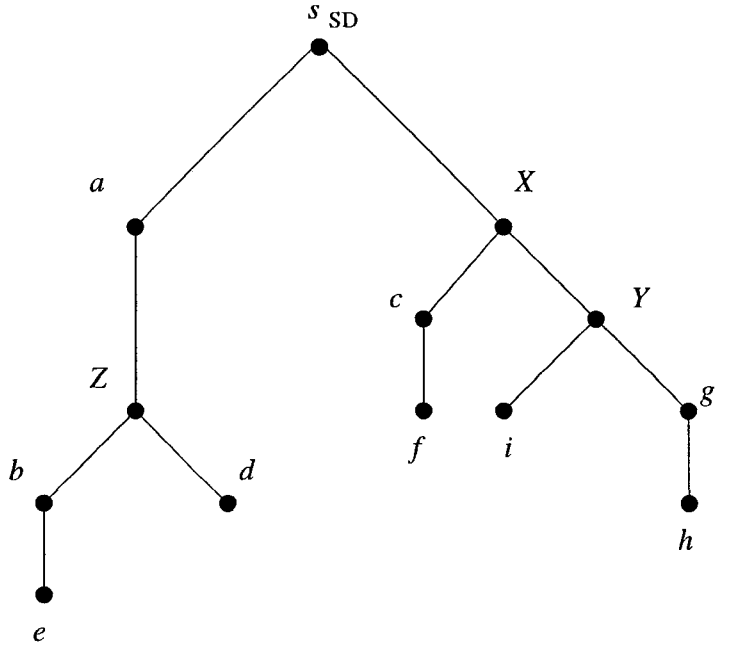

Figure 2.7: A dependency tree for SLCHP for the nodes of $SD(T)$ in Figure 2.6.

Vertex	Solution	Left List	Right List
d	$-\infty$	-	-
e	$-\infty$	-	-
f	$-\infty$	-	-
h	$-\infty$	-	-
i	$-\infty$	-	-
b	2	eb	eb
g	1	hg	hg
c	$-\infty$	fc	fc
Z	3	eb, db	ebd, bd
Y	3	hg, ig	hgi, gi
a	3	dba, eba, ba	dba, eba, ba
X	3	cgh, cg, cf, cgi	$hgi, cgi, gi, fcgi$

Table 2.1: The solution computed for each vertex of $SD(T)$ (Figure 2.6) by the algorithm SLCHP.

$(1, [hg], [hg])$. However, at c, the path fc has overall positive length, hence there is still no solution, so $(-\infty, [fc], [fc])$ is returned.

The next nodes to be processed are Z and Y. For Z, the path (db) is added to

the list for node d, and so the path list for Z is $[eb, db]$. Scanning this list results in solution ebd, and the tuple $(3, [eb, db], [ebd, bd])$ is returned. Similarly, for Y the path gi is appended to the path list at i, and the solution at g is hgi. Therefore, $(3, [hg, ig], [hgi, gi])$ is returned.

We can now process node a, which appends edge ab to the path list of Z. At a, the solution remains 3 (ebd), and $(3, [dba, eba, ba], [dba, eba, ba])$ is returned.

The final two nodes to be processed are X and s_{SD}. At X, cg is appended to the left list Y and merged with c. This results in the list of paths $[cgh, cg, cf, cgi]$, and the solution 3 (fgh). For s_{SD}, ac is appended to the left list for X and merged with a, resulting in the list of paths $[acgh, acg, acf, acgi, abd, abe, ab]$. Scanning this for the best pair of paths yields $ebacgh$ which has weight 9 and length -1. This is the solution of LCHP on tree T.

2.5.4 Analysis of SLCHP

Theorem 1. *Algorithm LCHP runs in time $O(n \log n)$, where n is the number of vertices in T.*

Proof. T can be transformed into a binary tree with $O(n)$ nodes and edges in $O(n)$ time [36], and the spine decomposition (of size $O(n)$) can be constructed in $O(n)$ time [8]. Therefore, $T_{LCHP}(n) = O(n) + T_{recurseLCHP}(n)$. For $T_{recurseLCHP}(n)$, we will consider total cost per node processed.

Consider vertex x in the tree. Trivially, when x is processed at a leaf node of $SD(T)$, thecost is $O(1)$. At a spine node of degree 3, an edge is appended to the path from the root to x, and then it is checked against the current solution to LCHP ($recurseLCHP$, lines 11-17). This also costs $O(1)$ time.

At a BST node, x is merged into a combined list, and then checked against the current solution. Depending on which subtree x is in, the path from x to the root may be extended, but in either case the cost remains the same. While computing *best* and *otherbest* for $1 \leq i \leq n$, we can remember and update the best path found so far, so x is checked a constant number of times ($BSTnode$,lines 14-16). In the nested loops, x is visited exactly twice (when it is indexed by i and j) ($BSTnode$, lines 19-20). Therefore, the total cost for x is again $O(1)$.

Since the depth of a spine decomposition is $O(\log n)$ [8], x appears in $O(\log n)$ subtrees of $SD(T)$. Therefore, with n vertices, the analysis yields

$$\begin{aligned} T_{LCHP}(n) &= O(n) + T_{recurseLCHP}(n) \\ &= O(n) + O(n \log n) \\ &= O(n \log n) \end{aligned}$$

□

Theorem 2. *Algorithm LCHP correctly computes* $hw(T, w, l, B)$.

Proof. It suffices to show that every path in T is checked by the algorithm. Consider an arbitrary path $P = \{u, \ldots, v\}$ in T. Let $Q = \{w, \ldots, z\}$ be the segment of P on the highest spine in $SD(T)$. Denote this spine S. For instance, in Figure 3, if $P = gbch, Q = bc$, and $S = abcd$. Let y be the lowest common ancestor of w and z in the binary search tree over S. P is checked by $LCHP$ when y is processed. □

Corollary 1. *Algorithm LCHP also solves the LNP problem for trees of* arbitrary *degree in time* $O(n \log n)$.

Proof. LNP is a special case of LCHP. □

Chapter 3

Fully Dynamic Trees

3.1 Problem Statement

In *dynamic trees problems*, attributes for 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 (see Figure 3.1). Because we are allowing for edge deletions, these trees are referred to as *fully* dynamic. Typical operations on fully dynamic trees include maintaining tree diameter, finding the minimum cost edge on a path, adding a constant weight to the cost of all edges on a path, or finding the maximum subsequence of a path.

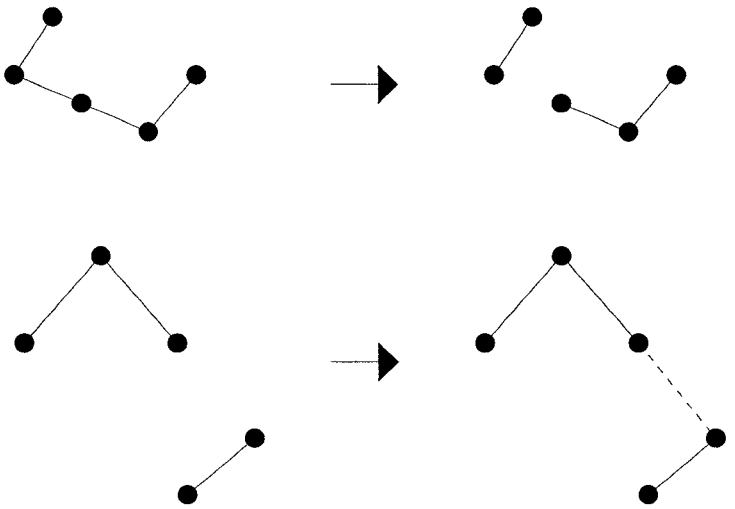

Figure 3.1: Edge deletions (above) and insertions (below) in trees

In the remainder of this chapter we discuss previous solutions and applications of the dynamic trees problem. In Chapter 4 we present our own solution to the dynamic trees problem, DS-trees, which we then use to solve the maximum subsequence

problem, which is a new problem on dynamic trees.

3.2 Previous Work

There are several well-known data structures addressing the dynamic trees problem in $O(\log n)$ time per update. In each case, an arbitrary tree is transformed into a balanced one, via a number of different methods. Sleator and Tarjan's ST-trees [32, 33] was one of the earliest solutions. ST-trees partition the underlying tree into vertex-disjoint paths, and represents each one with a binary tree. ET-trees, introduced by Henzinger et al in [23], represent the dynamic tree with an Euler tour (a tour that traverses each edge twice, once in each direction).

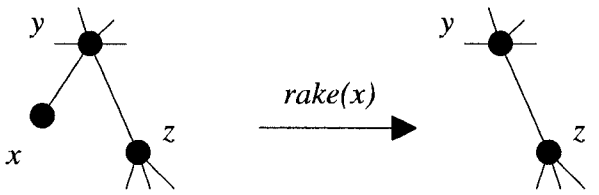

Figure 3.2: An example of a rake operation.

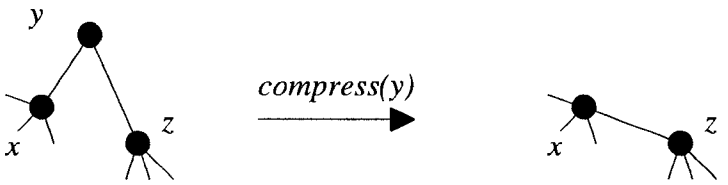

Figure 3.3: An example of a compress operation.

The final class of data structures for dynamic trees are based on *tree contractions*, which utilize *rake* (leaf removal) and *compress* (degree two vertex removal) operations, as illustrated in Figures 3.2 and 3.3. Each instance of these operations creates a *cluster* that stores information about the removed vertices. Frederickson's topology trees [15, 16, 17] and Acar et al's RC-trees [1, 2] use rake and compress operations. However, maintaining tree data during rake and compress operations is cumbersome, which led Alstrup et al to introduce top trees [5], a refinement of topology trees. Top

trees provide an interface hiding the rake/compress operations. Topology trees and top trees are both designed for dynamic trees of fixed degree, and extending them to handle arbitrary trees via ternarization is cumbersome and adds extra depth to the data structure. In [39], Tarjan et al introduce self-adjusting top trees which handle arbitrary trees without ternarization. However the run-time of edge insertion and deletion algorithms are now reduced to amortized $O(\log n)$. We briefly describe each method and discuss the types of problems on dynamic trees they are used to solve, before presenting our solution to the dynamic trees problem, DS-trees [11], in Chapter 4.

3.2.1 ST-trees

ST-trees partition the edges of the dynamic tree T into *solid* and *dashed* edges. For each vertex v, $size(v)$ is defined to be the number of vertices in T descended from v. An edge (v, w) in T is marked solid if and only if $2 \cdot size(v) > size(w)$. All other edges are dashed. Solid edges define a set of solid paths partitioning the vertices of T. If some vertex has no incident solid edge it is a one-vertex path. Solid paths are illustrated in Figure 3.4. The data structure provides function $expose(v)$ that repartitions T such that there is a unique solid path connecting v to the root of T. This allows the user to manipulate this path in some manner. Note that *expose* converts solid edges to dashed and vice-versa, and may violate the size condition. Thus, ST-trees also provide a *conceal* function to rectify the damage caused by *expose*. Other functions for ST-trees include *concatenate*, which combines two paths by inserting an edge between them, and *split*, which partitions a path by removing all edges incident to a vertex v in the path. Link and cut operations are implemented via sequences of *expose*, *conceal*, *concatenate*, and *split*.

To achieve amortized $O(\log n)$ time per update, every solid path is represented by a splay tree [33], a self-balancing binary search tree that also provides fast access to recently accessed items. These trees are then all connected to form a large virtual tree representing the underlying dynamic tree. A splay tree-based implementation does not require the *conceal* operation. To achieve worst-case $O(\log n)$ time per update, a globally-biased search tree is used. However, in [38], the authors admit that this

solution is prohibitively difficult to implement.

ST-trees associate a numerical cost with every vertex that is retrieved via the *findcost* operation. It is through these costs that information about the dynamic tree is maintained and manipulated. Sleator et al are able to compute a variety of tree attributes in $O(\log n)$ time per operation, such as nearest common ancestor and minimum cost vertex on a path. They also provide a method for adding a constant cost x to all edges on a given path [32].

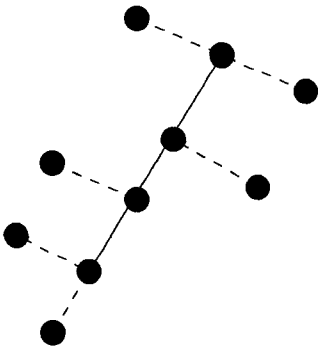

Figure 3.4: An example of a solid path in a tree.

3.2.2 ET-trees

ET-trees represent a rooted tree T by its Euler tour which is defined as follows [23]:

Algorithm 4 *ET*

```
1: Input: Vertex x
2: Visit x
3: for Each child c of x do
4:     ET(c)
5:     Visit x
6: end for
```

This tour begins and ends at the root vertex of T, and hence can be considered a circular list. A given vertex v in T appears in $ET(T)$ more than once; each appearance is referred to as an *occurrence* of v, denoted o_v. Every edge in T appears twice. However, this list has $O(n)$ length. The method of Henzinger et al [23] breaks this

list at an arbitrary point, and then builds a search tree over the list such that the leaves of the search tree are vertices in the list.

To delete an edge $e = (u, v)$ from T (splitting T into two trees T_1 and T_2), first locate the two instances of e in $ET(T)$, (o_{u_1}, o_{v_1}) and (o_{u_2}, o_{v_2}). Assuming o_{v_1} comes before o_{v_2}, $E(T_2)$ is represented the interval $ov_1 \ldots o_{v_2}$ of T, and $ET(T_1)$ is the what remains of $ET(T)$ when $ET(T_2)$ is spliced out.

The root of T can be switched to an arbitrary vertex v by finding (any) o_v in $ED(T)$, removing the entire prefix before it, appending it to the end of the tour, and then adding a new occurrence of o_v to the end of the tour.

This root switch operation is necessary for edge insertion. To connect T_1 and T_2 via edge (u, v), reroot T_2 at v, and then append $ET(T_2)$ to $ET(T_1)$.

All these operations have $O(\log n)$ worst-case running time in a tree of fixed degree, and $O(\frac{d \log n}{\log d})$ worst case running time in a tree of degree d. ET-trees are able to efficiently perform operations over subtrees of T (such as locating the minimum weighted edge in a subtree, or adding a constant value to every edge in a subtree). However, since the euler tour of T is broken at an arbitrary point, consecutive edges in a given path in T may be arbitrarily far apart in $ET(T)$. This limits the ability of ET-trees to store information over paths [38].

3.2.3 Top Trees

In [5], Alstrup et al refine the work of Frederickson [15, 16, 17] and present *top trees*. Top trees support edge insertion and deletion in $O(\log n)$ time. Each node of a top tree is a *cluster* which represents a subtree and a path in the original tree. It is represented by a subtree C and set δC of one or two vertices in C, referred to as the *boundary vertices*. Clusters are joined via rake and compress operations, which aggregates the information stored at each child. Starting with a cluster representing every edge in the original tree T, a top tree of T is the binary tree representing all the contractions used to construct T.

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, such as the minimum edge weight between any two vertices (in $O(\log n)$ time per query). In [5] 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.

Frederickson's topology trees use individual vertices as base clusters instead of edges, and contracted clusters are connected by an edge that is in neither base cluster. This complicates the aggregation of child clusters' data, and is undesirable.

3.2.4 Self-Adjusting Top Trees

In [39], Tarjan et al extend top trees to include trees of arbitrary degree. However, the cost for edge insertion and deletion is now *amortized* $O(\log n)$.

3.3 Applications

In the network flow problem, we are given a graph G, a source vertex s, a sink vertex t, and a set of edge capacities. The objective is to find the maximum flow from s to t that doesn't exceed the capacity of any single edge in G. In [37], the authors use ET-trees to implement the network simplex algorithm of Goldfarb et al [22]. In the minimum cost max flow problem, the edges in G also have an associated cost per unit flow. We now seek to find the maximum flow that minimizes total flow cost. The algorithm given by Orlin in [30] for this problem is also implemented by dynamic forests in [37]. Algorithms for the maximum flow utilizing ST-trees are given in [20, 21].

Dynamic trees are also used to perform computations on dynamic graphs. For instance, in [23], dynamic trees are used to maintained a $1 + \epsilon$ approximation of the minimum spanning tree for a dynamic graph G. They are also used to check bipartiteness and k-edge-connectivity through edge insertions and deletions in G.

Chapter 4

DS-Trees: Our Solution for Fully Dynamic Trees

We now present DS-trees, our novel data structure for maintaining non-local properties in dynamic forests. It, like ST-trees, is based on a path decomposition of the input tree. Unlike ST-trees, however, DS-trees easily implement worst-case $O(\log n)$ edge insertion and deletion algorithms. Furthermore, queries to an ST-tree often result in the path partition being changed (via *expose* and *conceal*). This is not the case with DS-trees, which are static throughout all queries. This allows parallel queries to be run on DS-trees with no cost, which is not true for ST-trees and top trees. This also allows users with read privileges (but not write) to execute queries on DS-trees.

The DS-tree is again based on the spine decomposition introduced in Chapter 2. We utilize the fact that a vertex v in search tree S has depth $O(\log \frac{w_S}{w(v)})$ where $w(v)$ denotes the number of leaf nodes descended from v and w_S denotes the total such weight for the tree S. We maintain this attribute through edge insertions (in Section 4.1) and deletions (in Section 4.2).

We then use DS-trees to compute the maximum subsequence of the path between any two nodes in the dynamic tree, and various other standard dynamic tree operations in Section 4.3 and 4.4.

4.1 Edge insertions

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 $S = \{v_0, \ldots, v_k\}$

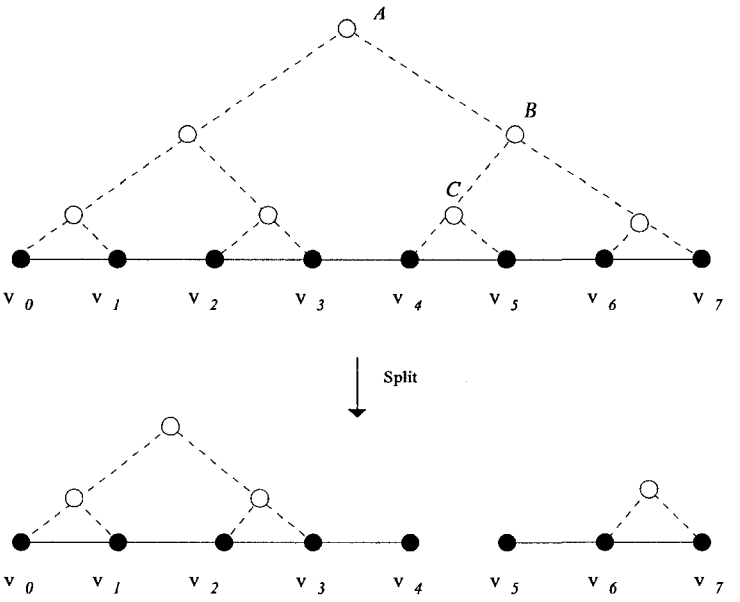

Figure 4.1: Splitting the spine at edge (v_4, v_5) requires that search tree nodes A, B, and C are deleted

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 4.1). 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 4.1). 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 4.1) 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.

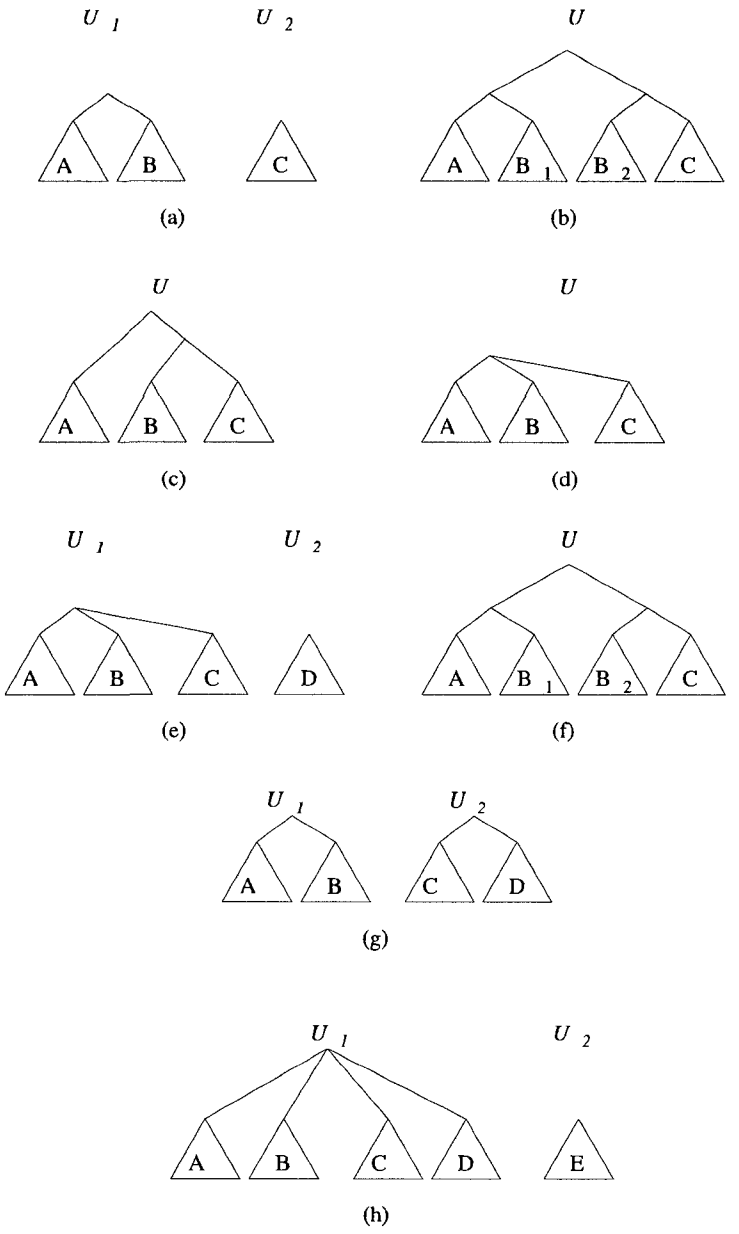

Figure 4.2: The various cases of *mergeTree* input.

Lemma 2. *Algorithm mergeTree results in a tree U such that for any vertex $u \in U$, the depth of u, denoted $d_U(u)$, is at most $3 \log \frac{w_U}{w(u)}$.*

Proof. In all cases, the depth of U_2 in U is at most 3. Hence, $d_U(U_2) \leq 3 \leq 3 \log \frac{2w_{U_2}}{w_{U_2}} \leq 3 \log \frac{w_U}{w_{U_2}}$.

If the depth of a subtree T does not change during the merge operation, the depth

condition is still satisfied. Let w_{old} denote weight of the tree containing T before the merge, and w_{new} denote the weight of the newly-merged tree. Since $w_{new} > w_{old}$, $d_{new}(T) = d_{old}(T) \leq 3\log\frac{w_{old}}{w_T} \leq 3\log\frac{w_{new}}{w_T}$.

Consider the case where U_1 has root of degree 2 and $w_A \leq w_C$ (line 5). In this case (Figure 4.2b), $d_U(B_i) = d_{U_1}(B_i) \leq 3\log\frac{w_U}{w_{B_i}}$ holds for $i \in \{1, 2\}$, and $d_U(A) = d_U(C) \leq 3\log(\frac{w_U}{w_C}) \leq 3\log(\frac{w_U}{w_A})$.

Consider the case where $w_A > w_C$ and $w_B \leq w_C$ (line 11). In this case (Figure 4.2c), $d_U(A) = d_{U_1}(A) \leq 3\log\frac{w_U}{w_A}$. For $B, d_U(B) = d_U(C) \leq 3\log(\frac{w_U}{w_C}) \leq 3\log(\frac{w_U}{w_B})$.

For the case where w_C is small and is connected to the root of U_1 (Figure 4.2d), $d_U(A) = d_{U_1}(A) \leq 3\log\frac{w_U}{w_A}$ and $d_U(B) = d_{U_1}(B) \leq 3\log\frac{w_U}{w_B}$.

We now consider the case where the root of U_1 is of degree 3 (Figure 4.2e). 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 4.2f), $d_U(A) = 2 \leq 3\log(\frac{2w_A}{w_A}) \leq 3\log(\frac{w_U}{w_A})$.

A similar argument can be used for B. From the degree-2 case, we have that $w_C \leq w_A + w_B$. Therefore, $d_U(C) = 2 \leq 3\log\frac{w_U}{w_C}$.

If $w_C + w_D < w_A + w_B$, we construct T_1 and T_2 as in Figure 4.2g (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 $d_U(C) = 3 \leq 3\log\frac{w_U}{w_C}$.

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 4.2h) and merge as if it is a 3-ary tree (the depth of A is unchanged). □

Lemma 3. *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 4.2e) 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 search tree for the new spine we iteratively apply *mergeTree* to all tree fragments.

Algorithm 5 *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 4.2a.
  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 4.2b and return.
  else if $w_A > w_C$ **then**
    if $w_B \leq w_C$ **then**
      Arrange A, B, C as in Figure 4.2c 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 4.2d) 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 4.2e.
  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 4.2f and return.
    else if $w_C + w_D < w_A + w_B$ **then**
      Join A and B as in Figure 4.2g and denote this joined tree T_1
      Join C and D as in Figure 4.2g 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 4.2e)
    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 4.2h
  We know that w_A and w_D are small relative 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

Lemma 4. *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 in $O(\log n)$ 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 4.1, $P = \{A, B, C, v_5\}$). Hence $k = O(\log n)$. *mergeTree* runs in $O(1)$ time. Iteratively applying it to all tree fragments takes $O(\log n)$ time. □

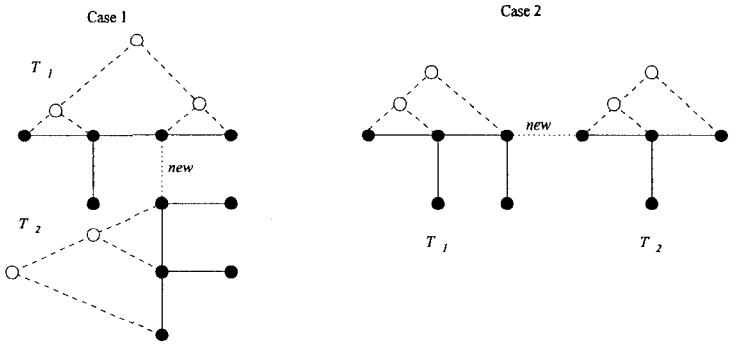

Figure 4.3: Edge insertion: Trees T_1 and T_2 are joined by edge *new*.

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

4.2 Edge deletion

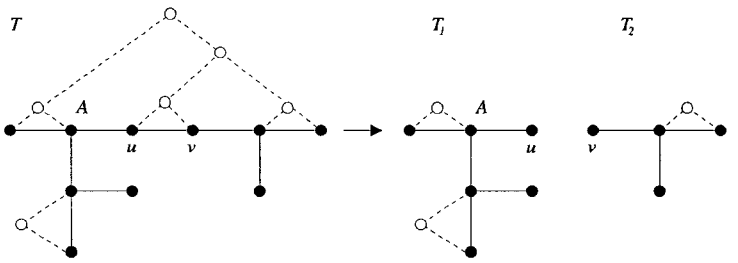

Figure 4.4: Edge removal: Tree T is split into T_1 and T_2 after edge (u, v) is removed. Vertex A is a breakpoint of T_1; the spine must be split at this point, as the child spine has more leaves than the rest of the topmost spine.

Edge deletion in a dynamic forest of trees with spine decompositions is more complex than edge insertion (Figure 4.4). Suppose edge (u, v) is removed from a DS-tree T, resulting in T_1 and T_2 where v is the root of T_2. To update the DS-tree for T_2 it suffices to remerge all the subtrees on the topmost spine of T_2.

However, in T_1, we must check every spine node on the path from u to the root to see if is a *breakpoint*. Given that $N_l(u)$ has decreased, at certain spine vertices the spine must be broken. For instance, in Figure 4.4, vertex A is one such breakpoint.

We define algorithm *FindBP* which accepts as input a search tree node z and outputs all breakpoints between $z.leftmost$ and $z.rightmost$. At each vertex z we store the largest value β_z such that there is a spine node c descended from z such that

$$w_c \geq w_{c+1} + w_{c+2} + \ldots + w_{rightmost(z)} + \beta_z$$

If, due to an edge deletion, $N_l(rightmost(z) + 1)$ becomes less than β_z, we can conclude that at least one breakpoint lies between $leftmost(z)$ and $rightmost(z)$. It is also easy to maintain β_z at each search node. If z is a search node with left child z_L and right child z_R,

$$\beta_z = \max\{\beta_{z_R}, \beta_{z_L} - (w_{leftmost(z_R)} + w_{leftmost(z_R)+1} + \ldots + w_{rightmost(z_R)})\}$$

Algorithm 6 $FindBP(v)$

1: **Input:** Vertex v in $SD(T)$
2: **Output:** All breakpoints descended from v
3: $L \leftarrow$ the number of leaf nodes in T descended from the spine vertex to the right of $rightmost(v)$
4: **if** $L < \beta_v$ **then**
5: **if** v is a spine node **then**
6: Output v
7: **else**
8: For all children c of v that are not in P_{new}, $FindBP(c)$.
9: **end if**
10: **end if**

Lemma 5. *Suppose edge (u, v) is removed from a DS-tree T, resulting in DS-trees T_1 and T_2 where v is the root of T_2. There are $O(\log l)$ breakpoints in T_1, where l is the number of leaf nodes in T_1.*

Proof. Consider breakpoints $b_0, b_1, \ldots, b_k$, where b_0 is closest to the root of T_1. Note that $N_l(b_0) \leq l$, where $N_l(v)$ is the number of leaf nodes descended from v.

If $j < i$, b_j is an ancestor of b_i.

Breakpoint b_i has two children. One is in the current spine, and one is in the spine connected below. The former child is an ancestor of b_{i+1}. This child has fewer descended leaves than the other. Hence, $N_l(b_i) \geq 2N_l(b_{i+1})$.

As we approach the root of T_1 the number of descended leaves from each breakpoint doubles. Therefore, there are $O(\log l)$ breakpoints. □

We can now state algorithm *DeleteEdge*, which removes edge (u, v) from the DS-Tree T.

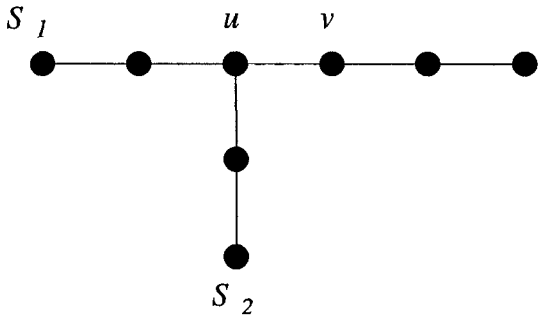

Figure 4.5: When (u, v) is removed, if there is a spine S_2 below u it must be merged with the segment of S_1 that is in T_1.

Algorithm 7 *DeleteEdge*

1: **Input:** DS-Tree T and cut edge (u, v)
2: **Output:** DS-Trees T_1 and T_2 where v is the root of T_2
3: Remove edge (u, v) from T by removing all search nodes on the path from u to the root of the spine containing u.
4: Rebuild the search tree for the topmost spine for T_2.
5: Rebuild the search tree for the spine of T_1 containing u. If u is connected to a second spine below, it is merged with the current spine, as in Figure 4.5.
6: Delete all search node vertices on the path from u to the root of the topmost spine of $SD(T_1)$, and re-merge the resulting subtrees, now based on the perturbed weight of u.
7: Construct path $P_{new} = \{v_0 = v, v_1, ..., v_m = s_{SD}\}$ through the new set of search trees.
8: Determine breakpoints $b_0, \ldots, b_k$ by executing $FindBP(v_i)$ for all $v_i \in P_{new}$. b_0 is the breakpoint closest to the root of T_1.
9: Starting with breakpoint b_0, we delete all vertices on the path from b_i to the root of the search tree, and re-configure the spines as in the case of edge insertion.

Lemma 6. *Algorithm DeleteEdge correctly updates DS-trees T_1 and T_2.*

Proof. In Section 4.1 we showed how to join two spine segments. Hence, it suffices to show that *DeleteEdge* finds all breakpoints. Since P_{new} passes through the root of every search tree it traverses, executing $FindBP$ on all vertices in P_{new} ensures that every breakpoint will be identified (see Figure 4.6). □

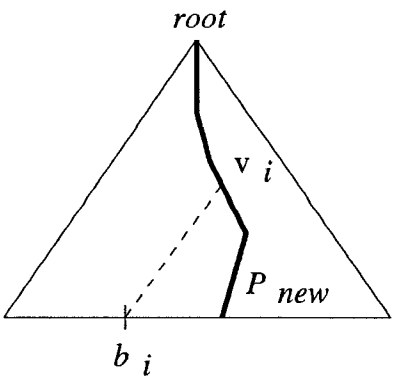

Figure 4.6: Since P_{new} passes through the root of the search tree, for every b_i there exists a v_i as illustrated in the diagram.

Lemma 7. *Algorithm DeleteEdge finds all breakpoints $b_0, \ldots, b_k$ in $O(\log n)$ time.*

Proof. Each b_i is connected to some vertex v_i in P_{new} by path M_i.

$FindBP(v_i)$ runs in $O(|M_i|)$ time.

By the property of DS-trees, $M_i \leq c \log \frac{w(v_i)}{w(b_i)})$ where $c \geq 3$.

Note that $w(v_i) \leq w(b_{i-1})$, since all leaf nodes descended from v_i are also descendants of b_{i-1}.

$$\begin{aligned}
\sum_{i=0}^{k} |M_i| &\leq c \log \frac{w(v_0)}{w(b_0)} + c \log \frac{w(v_1)}{w(b_1)} + \ldots + c \log \frac{w(v_k)}{w(b_k)} \\
&\leq c \log \frac{w(v_0)}{w(b_0)} + c \log \frac{w(b_0)}{w(b_1)} + \ldots + c \log \frac{w(b_{k-1})}{w(b_k)} \\
&= c \log \frac{w(v_0) w(b_0) w(b_1) \cdots w(b_{k-1})}{w(b_0) w(b_1) \cdots w(b_k)} \\
&= c \log \frac{w(v_0)}{w(b_k)}
\end{aligned}$$

This is the upper bound of length of the the path from b_k to v_0. Therefore, the total length of all M_i is $O(\log n)$, and all calls to *FindBP* execute in $O(\log n)$ time. □

With the result of Lemma 7, we are able to prove that the running time of *DeleteEdge* is $O(\log n)$.

Lemma 8. *The time complexity of algorithm DeleteEdge is* $O(\log n)$.

Proof. Steps 3/4/5: The number of trees to merge is linear in the length of the path from (u, v) to the root, which is $O(\log n)$ (as in the case of edge insertion). Therefore the time complexity is $O(\log n)$.

Step 6: The number of trees to merge is linear in the length of the path, which is $O(\log n)$.

Step 7: The length of path $P_{new} = \{v_0 = v, v_1, ..., v_m = s_{SD}\}$ is $O(\log n)$.

Step 8: From Lemma 7, we obtain all breakpoints $b_0, \ldots, b_k$ in $O(\log n)$ time.

Step 10: The path from b_i to the root overlaps at some point with P_{new}. This path is denoted Q_i. Let X_i be the segment of Q_i not in P_{new}, and Y_i be the segment of Q_i that overlaps with P_{new}.

X_i is equivalent to M_i from the proof of Lemma 7. Therefore $\sum_{i=0}^{k} |X_i| \leq c \log n$.

Consider breakpoints b_i, b_j where $i < j$. Processing b_i first ensures that when splitting b_j, the root of the search tree has changed to a vertex to the right of P_{new}. Therefore, Y_i and Y_j do not overlap, as shown in Figure 4.7. This implies $\sum_{i=0}^{k} |Y_i| \leq |P_{new}|$.

The number of trees to be merged is linear in the sum of the lengths of all paths Q_i, which is $O(\log n)$. Since trees can be merged in constant time, the time complexity of step 10 is $O(\log n)$. □

4.3 Maximum subsequence queries in a dynamic forest

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* [9]. In the field of bioinformatics, this problem arises frequently in the analysis of DNA and protein sequences [27], homology modeling [19], ontology

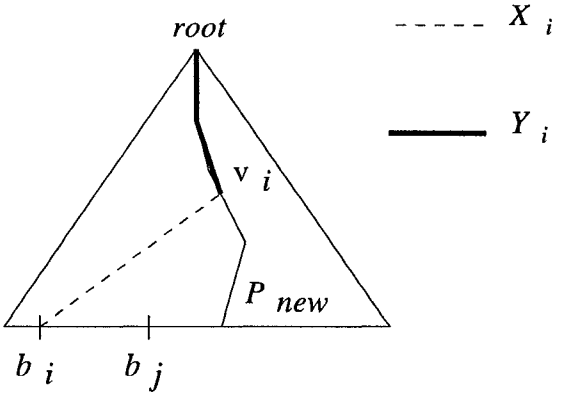

Figure 4.7: If b_i is split before b_j, we ensure that when splitting b_j, the root of the search tree is to the right of P_{new}. Therefore, Y_j cannot include any of the bolded section of P_{new}.

matching [18], and microarray design [10]. The maximum subsequence is also used when ranking k maximum sums [7] and computing the longest and shortest subarrays satisfying a sum or average constraint [13]. In [31], Ruzzo et al present a $O(n)$ time algorithm that computes all maximum subsequences in a given sequence. This problem is extended to trees as follows:

Definition 6. *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, computing the maximum subsequence requires the aggregation of data over paths in the underlying tree. This makes ET-trees also unsuitable.

We first discuss the maximum subsequence problem for a sequence before extending it to 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 sequence $S_1.S_2$ 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 [6]. 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 [6], 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 [31], 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. We again use the notation established in Chapter 2, for the leftmost, rightmost, and cover of a vertex v.

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 9. *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 [31] 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 3. *Edge insertion and deletion in a dynamic forest while maintaining S_v at every search tree vertex v takes $O(\log n)$ time.*

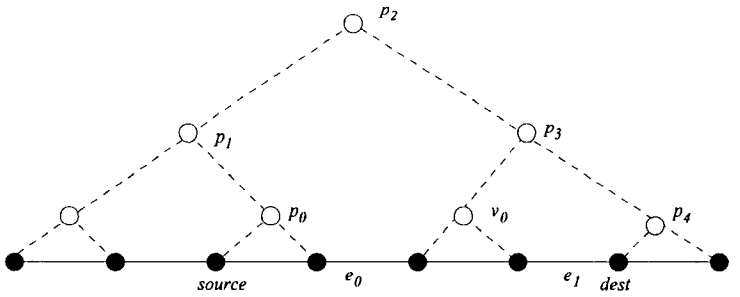

Figure 4.8: Path $P' = \{source, p_0, p_1, p_2, p_3, p_4, dest\}$ connects *source* and *dest*. Vertices $source, p_0, v_0$, and *dest* are chosen by our algorithm. Their covers are connected by edges e_0 and e_1.

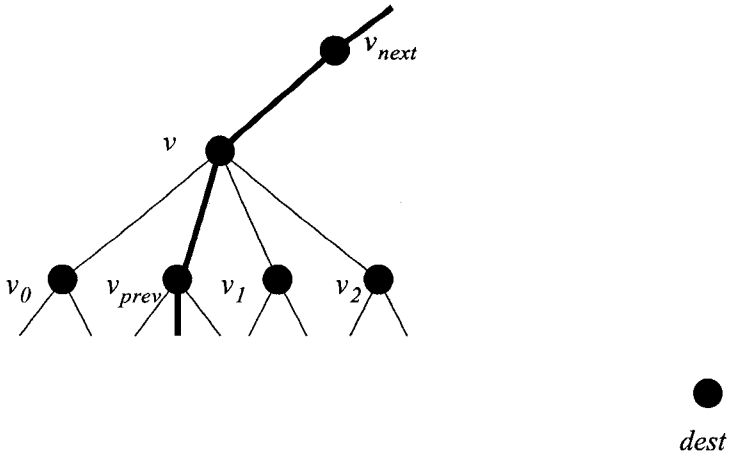

Figure 4.9: If v is not selected by our algorithm, then vertices v_1 and v_2 are.

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 4.8).

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.

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 (as in Figure 4.9) and add them to our collection.

Lemma 10. *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 4. *When S_v is stored at all search nodes in $SD(T)$, we can compute $MS(u, v)$ in $O(\log n)$ time.*

4.4 Other results

Lemma 11. *SD-trees are able to select the minimum weight edge on a path $P = \{u, \ldots, v\}$ in $O(\log n)$ time per query.*

Proof. At each search node s we store the edge of minimum weight on the spine segment covered by s. As for the solution to $MS(u, v)$, we construct the sequence M as before. We then check all edges and search nodes in M, and pick the one of least weight. This can be done in $O(\log n)$ time.

When deleting a search node, this information is discarded. When a new search node is created, we examine the value stored at each of its children and pick the smallest one. Thus, the SD-tree is still maintained in $O(\log n)$ time. □

Lemma 12. *SD-trees are able to add a constant value c to all edges on a path $P = \{u, \ldots, v\}$ in $O(\log n)$ time.*

Proof. At each search tree node s we store a "lazy" weight w that is applied to the spine edges covered by s. Again, we construct the sequence M covering P, with length $O(\log n)$. We add c to all edge weights and "lazy" search tree node weights in M. □

We are also able to maintain the tree *diameter*, the longest path in the tree. We are able to support $O(1)$ time diameter queries.

Lemma 13. *SD-trees maintain tree diameter in $O(1)$ time per query and $O(\log n)$ time per tree update.*

Proof. For each search tree node s we maintain the diameter of the tree covered by s, $s.D$. We also store the longest path in the cover of s ending at $leftmost(s)$, the longest path ending at $rightmost(s)$, which we denote $s.left$ and $s.right$, respectively, and the path $s.cross$ connecting $leftmost(x)$ and $rightmost(s)$.

When creating a new search node s with children s_L and s_R, we concatenate $s_L.right$ with $s_R.left$ into a new path *concat* and set $s.D = \max\{concat, s_L.D, s_R.D\}$.

We concatenate $s_L.cross$ with $s_R.left$, compare it to $s_L.left$, and set $s.left$ to the maximum of those two values. We similarly compute $s.right$. All this can be done in $O(1)$ time and therefore does not add any overhead to edge insertion or deletion.

It remains to handle the case where a search node s is appended to a new parent q with existing children $(q_0, \ldots, q_k)$ where $k \leq 2$. Without loss of generality, assume s is being appended as the new rightmost child of q. Construct a virtual search node v with left child q and right child s via the aforementioned method. We then replace q with v and attach children $q_0, \ldots, q_k$, and s.

When querying the diameter of a dynamic tree, we simply return $s_{SD}.D$. □

4.5 Conclusion

In Table 4.1 we present an overview of various solutions to the dynamic trees problem and and compare their ability to compute the minimum edge weight on a path and tree diameter, and add a constant value to all edge weights on a path. Note that while both top trees and DS-trees can maintain the diameter of dynamic trees, top trees use $O(\log n)$ time queries while DS-trees require only $O(1)$ time. We also list which data structures are updated in $O(\log n)$ time in the worst case, and which are amortized $O(\log n)$.

Data Structure	**Min Edge**	**Diameter**	**Add Value**	**Worst-case $O(\log n)$**
DS-trees	yes	yes	yes	yes
ST-trees	yes	no	yes	no
Top Trees	yes	yes	yes	yes
ET-trees	yes	no	no	yes

Table 4.1: A comparison of solutions to the fully dynamic forests problem

Chapter 5

Future Work

DS-trees can be further refined to handle queries for other, different tree attributes. For example, tree center and tree median, more examples of attributes that are typically computed by tree contraction based solutions to the dynamic trees problem [5].

Currently there is no process by which a DS-tree can be re-rooted. A $O(\log n)$ time algorithm that changed the root of a DS-tree would allow arbitrary edge insertion between trees in $O(\log n)$ time.

Additionally, DS-trees only process binary trees. Trees of arbitrary degree are handled via ternarization of high-degree vertices, which does not add any time or space complexity to DS-trees, but is still cumbersome. Extending DS-trees to more gracefully handle such trees would eliminate this.

Bibliography

[1] Acar U, Blelloch G, Harper R, Vittes J, Woo S, "Dynamizing static algorithms, with applications to dynamic trees and history independence," *Proc. 15th Symposium on Discrete Algorithms, 2004*, 524-533

[2] Acar U, Blelloch G, Vittes J, "An experimental analysis of change propogation in dynamic trees," *Proc. 7th Workshop on Algorithm Engineering and Experiments, 2005*, 41-54

[3] Ahuga R, Orlin J, Tarjan R, "Improved time bounds for the maximum flow problem," *SIAM Journal on Computing, 1989*,18:939-954

[4] Allison L, "Longest biased interval and longest nonnegative sum interval," *Bioinformatics, 2003*, 9:1294-1295

[5] Alstrup S, Holm J, Thorup M, de Lichtenberg K, "Maintaining information in fully dynamic trees with top trees," *ACM Transactions on Algorithms, 2005*, 1:243-264

[6] 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

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

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

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

[10] 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

[11] Bhattacharyya B, Dehne, "Efficient maximum subsequence queries and updates for dynamic forests," *Carleton University Technical Report 0805, 2008*

[12] Bhattacharyya B, Dehne F, "Using spine decompositions to efficiently solve the length-constrained heaviest path problem for trees," *Carleton University Technical Report 0806, 2008*, submitted

[13] 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

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

[15] Frederickson G, "Data structures for on-line update of minimum spanning trees, with applications," *SIAM Journal of Computing, 1985*, 14:781-798

[16] Frederickson G, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees," *SIAM Journal of Computing, 1997*, 26:484-538

[17] Frederickson G, "A data structure for dynamically maintaining rooted trees," *Journal of Algorithms, 1997*, 24:37-65

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

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

[20] Goldberg A, Grigoriadis M, Tarjan R, "Use of dynamic trees in a network simplex algorithm for the maximum flow problem," *Mathematical Programming, 1991*, 50:277-290

[21] Goldberg A, Tarjan R, "A new approach to the maximum flow problem," *Journal of the ACM, 1988*, 38:921-940

[22] Goldfarb D, Hao J, "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2)$ time," *Mathematical Programming, 1990*, 47:353-365

[23] Henzinger M, King V, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *Proceedings of the 27th Symposium on Theory of Computing, 1997*, 519-527

[24] Huang X, "An algorithm for identifying regions of a DNA sequence that satisfy a content requirement," *Computer Applications in the Biosciences, 1994*, 10:219-225

[25] Kim S, "Algorithm for finding a length-constrained heaviest path of a tree," *Transactions of the Korea Information Information Processing Society, 2006*, 13A:541-544

[26] Kim S, "Finding a longest nonnegative path in a constant degree tree," *Information Processing Letters, 2005*, 93:275-279

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

[28] Lin Y, Jiang T, Chao K, "Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis," *Proc. 27th International Symposium on Mathematical Foundations of Computer Science, 2002*, 459-470

[29] Nekrutenko A, Li W-H, "Assessment of compositional heterogeneity within and between eukaryotic genomes," *Genome Research, 2000*, 10:1986-1995

[30] Orlin J, "A polynomial time primal network simplex algorithm," *Mathematical Programming, 1996* 78:109-129

[31] 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

[32] Sleator D, Tarjan R, "A data structure for dynamic trees," *Journal of Computer and System Sciences, 1983*, 3:362-391

[33] Sleator D, Tarjan R, "Self-adjusting binary search trees," *Journal of the ACM, 1985*, 32:652-686

[34] Stojanovic N, Florea L, Riemer C, Gumucio D, Slightom J, Goodman M, Miller W, Hardison R, "Comparison of five methods for finding conserve sequences in multiple alignments of gene regulatory regions," *Nucleic Acids Research, 1999*, 19:3899-3910

[35] Stojanovic N, Dewar K, "Identifying multiple alignment regions satisfying simple formulas and patterns," *Bioinformatics, 2005*, 20:2140-2142

[36] Tamir A,"An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs," *Operations Research Letters, 1996*, 19:59-64

[37] Tarjan R, "Dynamic trees as search trees via euler tours, applied to the network simplex algorithm," *Mathematical Programming, 1997*, 78:169-177

[38] Tarjan R, Werneck R, "Dynamic trees in practice," *Proceedings of the 6th Workshop on Efficient Algorithms, 2007*, 80-93

[39] Tarjan R, Werneck R, "Self-adjusting top trees," *Proceedings of the 16th SODA, 2005*, 813-822

[40] Wu BY, Chao K-M, Tang CY, "An efficient algorithm for the length-constrained heaviest path problem on a tree," *Information Processing Letters, 1999*, 69:63-67

[41] Wu BY, Tang CY, "An $O(n)$ algorithm for finding an optimal position with relative distances in an evolutionary tree," *Information Processing Letters, 1997*, 63:263-269