Algorithms for Computing Maximum Agreement Subtrees

Nikolaj Skipper Rasmussen 20114373

Thomas Hedegaard Lange 20113788

Master's Thesis, Computer Science

June 2016

Advisor: Christian Nørgaard Storm Pedersen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis concerns the construction of maximum agreement subtrees between rooted binary trees. It describes two different algorithms for computing such trees. One with an upper bound runtime of $O(n^2)$ and one with an upper bound runtime of $O(nlogn)$ for input trees of size n. The former has a space consumption of $O(n^2)$ and the latter a space consumption of $O(nlogn)$.

The algorithms have been implemented, tested and evaluated, and experiments show that the time complexities are satisfied in practice even though the topology of the input trees affects both the runtime and the space consumption of the $O(nlogn)$ algorithm. For input trees of size greater than ~60, the $O(nlogn)$ algorithm outperforms the $O(n^2)$ algorithm.

A third algorithm has been implemented that reduces the MAST problem to a Longest Increasing Subsequence problem, also giving a runtime of $O(nlogn)$. This algorithm only works for input trees where each internal node has at least one leaf as a child, but outperforms the other $O(nlogn)$ algorithm.

Contents

Chapter 1

Introduction

The Maximum Agreement Subtree problem (MAST) is concerned with mutual information between rooted trees, and is defined as such: Given two rooted trees, T_1 and T_2, created over the same leaf-set $\{1, 2, 3..., n\}$, determine the largest possible subset of leaves inducing an agreeing subtree of T_1 and T_2. For a set of leaves to induce an agreeing subtree for T_1 and T_2, the subtrees restricted to the set of leaves must be isomorphic, implying structural equivalence.

Let us start by motivating the interest in MAST by giving an example of its application. Suppose that we are interested in inspecting the relationship between DNA obtained from different animal species. This is typically done by the use of Hierarchical Clustering or Neighbour Joining to construct evolutionary trees. However, finding the true evolutionary tree is often an elusive task, and evidence is required to support any suggested tree topology. Finding the Maximum Agreement Subtree will present the information that both trees agree on, which makes the information more reliable, given that multiple sources support it.

The MAST problem applies to all trees, but we will focus on the rooted, binary trees given that the motivation for the problem is primarily rooted in biology and linguistics, where these trees are most common.

Several different algorithms have been developed for solving the MAST problem with different time complexities. One of these is the algorithm described by Cole et. al. [1] which theoretically has a time complexity of $O(nlogn)$, where n is the number of leaves in each of the two input trees.

In the paper, we will specifically focus on this algorithm. We will give a detailed description of how the algorithm works and how it can be implemented. We will also walk through the algorithm described by Goddard et. al. [4] with time complexity $O(n^2)$ and compare the two algorithms in order to clarify strengths and weaknesses of each in theory and in practice.

1.1 Thesis Structure

The thesis is structured as follows. Chapter 2 gives some practical information about the programs we have implemented. In chapter 3 we explain the most

basic solution to the MAST problem, which is used as a verification tool for the other algorithms in the paper. In chapter 4 we walk through the $O(n^2)$ algorithm for solving the MAST problem described by Goddard et. al. [4]. In chapter 5 we focus on the $O(nlogn)$ algorithm described by Cole et. al. [1]. We specifically go though each step of the algorithm and describe both the time and space complexity by the end of the chapter. Chapter 6 describes how we used the naive algorithm to verify the correctness of our implementations of the $O(n^2)$ and $O(nlogn)$ algorithms. Chapter 7 covers the experiments conducted in order to verify the theoretical claims regarding time and space complexity. Finally, Chapter 8 briefly sums up the results of our work, and what future work might include.

1.2 Division of Labour

Throughout the development of the master thesis, Nikolaj has had a great deal of illness, which changed our initial plans of how the work should be shared between us. Initially, we worked together in understanding and implementing the algorithms and writing the thesis. However due to Nikolajs illness, the last part of the thesis was primarily conducted by Thomas.

Working on the master thesis comprised the following:

Reading and Understanding the Articles

Early on, we worked together in reading and understanding the articles to a point that made it possible for us to start implementing the algorithms.

Implementing the Algorithms

The first algorithms to implement was the $O(n^2)$ algorithm and the naive algorithm. This was done together. We worked together in planning and designing the implementation of the $O(nlogn)$ algorithm and did the first parts of the implementation together. Completing the implementation was done by Thomas.

Testing Algorithms and doing Experiments

Testing the correctness of the algorithms was done together. We designed the first experiments together, but the final experiments were designed and carried out by Thomas. We discussed the results of all experiments together.

Structuring and Writing the Thesis

Structuring the thesis was done together. The chapters describing the naive and the $O(n^2)$ algorithm was primarily written by Nikolaj, while the chapters describing the $O(nlogn)$ algorithm and the experiments was primarily written by Thomas. The rest of the thesis was done by both of us.

1.3 Definitions and Notations

Throughout this paper, all trees will be binary rooted trees. The size of a tree is the number of leaves and each input tree to a MAST algorithm has size n. When writing *log*, we refer to the binary logarithm.

Chapter 2

Implementation

We have successfully implemented the algorithms described in this thesis. All implementations were done in Java 8. For representing trees we used the library located at https://github.com/cmzmasek/forester.

The code for our implementations is available at:
https://github.com/batterihane/speciale

The implementation of the $O(n^2)$ algorithm consists of approximately 300 lines of code, whereas the implementation of the $O(nlogn)$ algorithm consists of approximately 3000 lines of code, excluding all libraries used. Besides that, we have created tests, experiments, etc.

The file 'readme.txt' can be found alongside the code on github. It describes the key classes to look out for and explains how to run the algorithms. Note that a recent JVM installation is required to run the program.

The test data used for testing correctness and for the experiments is located in the folder 'testTrees' alongside the code on github.

Chapter 3

The Naive algorithm

The MAST problem is not inherently hard to solve since a quite simple, although slow, algorithm can quickly be constructed. The naive solution to the problem is to run through all possible subsets of leaves. For each subset, induce the subtrees of the input trees according to the subset of leaves in question. Should the subtrees be isomorphic, we have an agreement subtree. The largest agreement subtree is simply the largest agreement subtree seen. In other words, we simply look through all possible solutions and pick the largest one. In order to do so, one must find a reasonably elegant way of iterating through all subsets of leaves, and a way of comparing two induced subtrees to see if they structurally agree over a given leafset.

We chose to represent a leaf subset as a bitstring, where each bit is dedicated to one specific leaf. The value of the bit naturally denotes if the leaf in question is included in the subset or not. By representing a subset as a bitstring, we can run through all possible subsets by starting with the bitstring of all zeros, $0_1 0_2 0_3 ... 0_n$, and incrementing it for each new subset needed until we reach the bitstring of all ones, $1_1 1_2 1_3 ... 1_n$. The pseudo code for the naive algorithm can be seen in Figure 3.1, where the tree pruning (the removal of 0-bit leaves) and comparison can be seen in the following subsections.

One can improve this naive solution slightly by realizing that the parity of the subset bit string represents the size of the agreement matching, if the agreement matching indeed exists. This means that you can simply run through all possible values of the bitstring, but starting with the values of highest parity instead. The instant we find a value representing an agreement matching, we know that it is the largest possible - the MAST. To keep the naive solution as simple as possible, we decided to refrain from implementing this small optimization.

As for the running time, we see that there are 2^n different subsets, which makes the naive approach exponential. The pruning of a tree can be done in $O(n)$ time, and checking for isomorphism can likewise be done in $O(n)$ time. All in all, the complexity for the entire algorithm is $O(3n * 2^n) = O(n2^n)$.

```
Function NaiveMast(Tree1, Tree2){
  n = Tree1.size;  //number of leaves
  largestMastSoFar = 0;
  mast = null;

  forall subset = 0_0 0_1 ... 0_n to 1_0 1_1 ... 1_n {
    tree1Mast = Prune(Tree1, subset);
    tree2Mast = Prune(Tree2, subset);

    if(isIsomorphic(tree1Mast, tree2Mast) &&
       tree1Mast.size > largestMastSoFar)
    {
      mast = tree1Mast;
      largestMastSoFar = tree1Mast.size;
    }

    return mast;
  }
}
```

Figure 3.1: Java pseudo code for The Naive Algorithm

3.1 Tree Pruning

Pruning is the act of removing a subset of leaves from a given tree and restoring the desired tree structure afterwards. In our case, where we deal with binary trees exclusively, we first remove the 0-bit leaves, and then remove any redundant internal vertices. An internal node is deemed redundant when it has less than two children. An example pruning can be seen in Figure 3.2, where $leaf_3$, $leaf_5$ and $leaf_7$ have been removed, making the internal nodes D,F and G redundant. Our pruning algorithm can be seen in Figure 3.3, where the sibling of a given node simply denotes the other child of the parent.

3.2 Tree Comparison

For each of the subsets in Figure 3.1, we have to ensure that the induced subtrees agree in the sense that they are isomorphic. The definition of isomorphism in this case can be seen in Definition 1 below.

Definition 1. *We say that two binary trees are isomorphic if they depict the same ancestral relationships. This is true exactly when there exists a mapping M, $T_1 \to T_2$, such that each leaf in T_1 maps to the corresponding leaf in T_2, and for any two leaves of T_1, L_1 and L_2:*

$$M(lca_{T_1}(L_1, L_2)) = lca_{T_2}(M(L_1), M(L_2))$$

where $lca_T(a, b)$ is the least common ancestor of two leaves a and b in T.

Intuitively, such a mapping is possible when you are able to obtain one tree

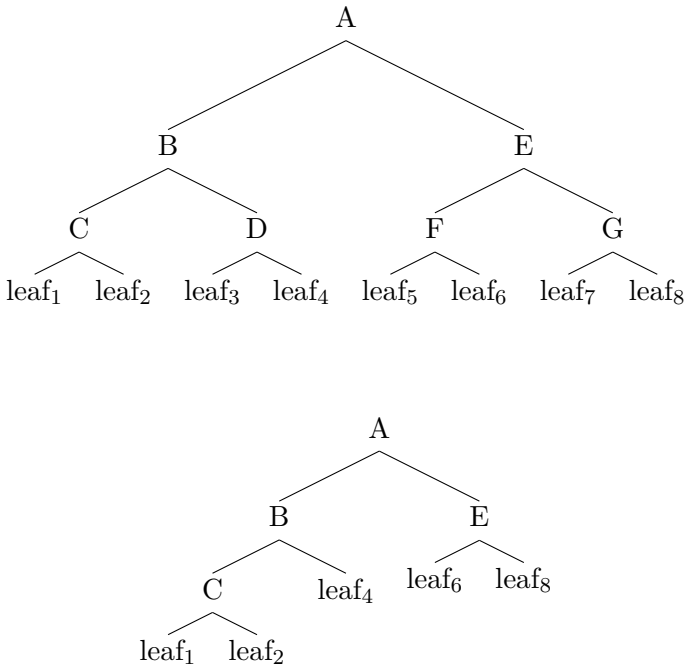

Figure 3.2: Pruning of the topmost tree according to bitstring 11010101

```
Function Prune(Tree1, subset){
  result = Tree1;

  forall (leaf: leaves in Tree1) {
    if(leaf ∈ subset) {
      parent = leaf.getParent();

      if(parent is the tree root) {
        result = Tree1.getSibling(leaf);
      }
      else
        /*make the leaf sibling a child of
          the leaf grandparent */
        Leaf sibling = leaf.getSibling().
        sibling.setParent(parent.getParent());
    }
  }
  return result;
}

```

Figure 3.3: Java pseudo code for Tree Pruning

from the other by swapping the order of children of any given internal node.

A simple recursive solution for checking isomorphism can be described as follows. Let T_a and T_x denote the input trees. We now have the following cases:

- Base Case: If either T_a or T_x is a leaf, check if the other denotes the same leaf. If it does, T_a and T_x are isomorphic, otherwise not.
- Recursive case: Let b and c denote the children of T_a, and let y and z denote the children of T_x. T_a and T_x are isomorphic if one of the following cases is true:

  T_b and T_y are isomorphic $\bigwedge$ T_c and T_z are isomorphic

  T_b and T_z are isomorphic $\bigwedge$ T_c and T_y are isomorphic

Chapter 4

The $O(n^2)$ algorithm

Goddard et. al.[4] describes a $O(n^2)$ algorithm for finding the maximum agreement subtree for two rooted binary trees. Given the two trees T_a and T_w each of size n, the idea is to iteratively find the largest agreement subtree and its size for every pair of subtrees from T_a and T_w. This can be done in quadratic time by using an algorithm based on Lemma 1. Note that the algorithm also works when T_a and T_w does not have the same leaf set.

Lemma 1. *Let T_a be a tree rooted at node a with the children b and c; And let T_w be rooted at w with children x and y. We now define $\#(T_a, T_w)$ as the size of the maximum agreement subtree of T_a and T_w. $\#(T_a, T_w)$ is given by the maximum of the following six numbers: $\#(T_b, T_x) + (T_c, T_y)$, $\#(T_b, T_y) + (T_c, T_x)$, $\#(T_a, T_x)$, $\#(T_a, T_y)$, $\#(T_b, T_w)$, $\#(T_c, T_w)$.*

Proof.

This is a proof by case over the possible subtrees contributing to the agreement subtree A_{aw} for T_a and T_w.

Case 1: Only T_b of T_a contributes to A_{aw}. If this is the case, then the agreement subtree can only exist between T_b and T_w, $\#(T_b, T_w)$.

Case 2: Only T_c of T_a contributes to A_{aw}. If this is the case, then the agreement subtree can only exist between T_c and T_w, $\#(T_c, T_w)$.

Case 3: Only T_x of T_w contributes to A_{aw}. If this is the case, then the agreement subtree can only exist between T_x and T_a, $\#(T_a, T_x)$.

Case 4: Only T_y of T_w contributes to A_{aw}. If this is the case, then the agreement subtree can only exist between T_y and T_a, $\#(T_a, T_y)$.

Case 5: This case covers all that case 1 through 4 doesn't, which means that all children b, c, x and y contibute to the MAST. Consider the agreement tree, A_r, rooted in r and with children s and t. We claim the following: A child of r cannot contain a leaf, b', from T_b and a leaf, c', from T_c at the same time.
If this is the case, then we only need to find the MAST size of child pairing permutations across T_a and T_w: $max\{\#(T_b, T_x)+\#(T_c, T_y), \#(T_b, T_y)+\#(T_c, T_x)\}$.

Let us now prove our claim by assuming that it doesn't hold, and use that to reach a contradiction. Suppose that A_s contains a leaf , b', from T_b and a leaf, c', from T_c. Let f be any leaf from A_t. Assume that f is from T_b (the other case is identical). We now see that the subtree induced by b', f and c' differs for A_r and T_a as illustrated in Figure 4.1. This is clearly contradictory to the

Figure 4.1: Two binary trees with different topology, induced by the same leaves.

$$f(T_a, T_w) = max \begin{cases} f(T_b, T_x) + f(T_c, T_y) & \text{if } a \text{ and } w \text{ are both internal nodes} \\ f(T_b, T_y) + f(T_c, T_x) & \text{if } a \text{ and } w \text{ are both internal nodes} \\ f(T_a, T_x) & \text{if } w \text{ is an internal node} \\ f(T_a, T_y) & \text{if } w \text{ is an internal node} \\ f(T_b, T_w) & \text{if } a \text{ is an internal node} \\ f(T_c, T_w) & \text{if } a \text{ is an internal node} \\ 1 & \text{if } a \text{ and } w \text{ are both leaves and } a{=}w \\ 0 & \end{cases}$$

Figure 4.2: Recursive MAST size function

definition of an agreement tree, which is why we must conclude our claim to be correct. □

What Lemma 1 essentially states is twofold. First, if the MAST contains leaves from all subtrees (T_b,T_c,T_x,T_y), then its size is given by the two largest distinct agreement matchings between pairs of these. Secondly, if the MAST does not include all subtrees, then its size is given by the largest agreement matching between T_a and T_w where one of the subtrees is excluded. This gives rise to a recursive solution for finding the size of the maximum agreement subtree. The recursive function is defined in figure 4.2.

Like many similar recursive definitions in bioinformatics, we run into the problem that the recursive function computes the same partial results multiple times. This problem is rectified by the method of dynamic programming. Specifically, we wish to store the partial results in a $O(|T_a|) \times O(|T_w|)$ table containing a row for each node in the first tree and a column for each node in the second tree. We seek to fill out the table row by row with our partial results, starting from the top left corner, such that a cell corresponding to two nodes contains the size of the maximum agreement subtree between the subtrees rooted at these two nodes. In the end, the size of the maximum agreement subtree should be stored in the bottom right corner. Looking at the function in figure 4.2 we see that the partial result for each tree pair is dependent on the results for their subtrees. This implies that we must list the table nodes in postorder, thereby ensuring that the partial results, on which a given node is dependent, have already been calculated when we reach it. By introducing

	$\mathbf{leaf_1}$	$\mathbf{leaf_2}$	**C**	**...**	**G**	**E**	**A**
$\mathbf{leaf_1}$	1	0	1	...	0	0	1
$\mathbf{leaf_8}$	0	0	0	...	1	1	1
J	1	0	1	...	1	1	2
...	...	...	...	...	...	...	...
F	0	0	0	...	0	0	2
M	0	1	1	...	1	1	3
H	1	1	2	...	2	3	6

Table 4.1: Score Matrix for the trees in figure 4.3

such a table we also introduce a requirement of $O(n^2)$ space and time, making it a quadratic algorithm. Simple techniques for reducing the space requirement, like only storing a few number of rows in the table at a time, cannot be applied in this case, since the computation of a new row may depend on any of the previous rows.

4.1 Algorithm example

Two example trees have been provided in figure 4.3. We now wish to compute the size of their maximum agreement subtree using the agorithm described above. For this reason, we create a 15×15 table, where the rows and columns correspond to the nodes of each tree in postorder. Using the function described in 4.2, we can now fill out the table row wise starting from the topmost row. The resulting table can be seen in 4.1, where we from the bottom right corner see that the maximum size of the MAST is 6. We can conclude that 6 is indeed correct, since removing only one leaf cannot produce an agreement tree in this case, while removing $leaf_1$ and $leaf_7$ will produce the desired agreement tree of size 6. Their MAST is likewise displayed in figure 4.3.

We will in the following sections show how one can extend this method for calculating the size of the MAST to computing the actual tree.

4.2 Computing the MAST

Extending the solution for computing the size of the MAST to include the computation of the actual MAST is fairly simple in this case. We found two different ways of going about it.

Option 1: Continual Tree Construction
The first option is to extend the size algorithm simply by, for each cell in the table, attaching the MAST responsible for that size. This cannot be done by copying subtrees from the input trees, since that would break our time constraints. For this reason, a cell instead stores either a leaf or a node with pointers to its children, and hence mutiple cells/masts will refer to the same subtrees. This means that we will end up creating a Directed Acyclic Graph, where the root node of the MAST is saved in the bottom right corner of the

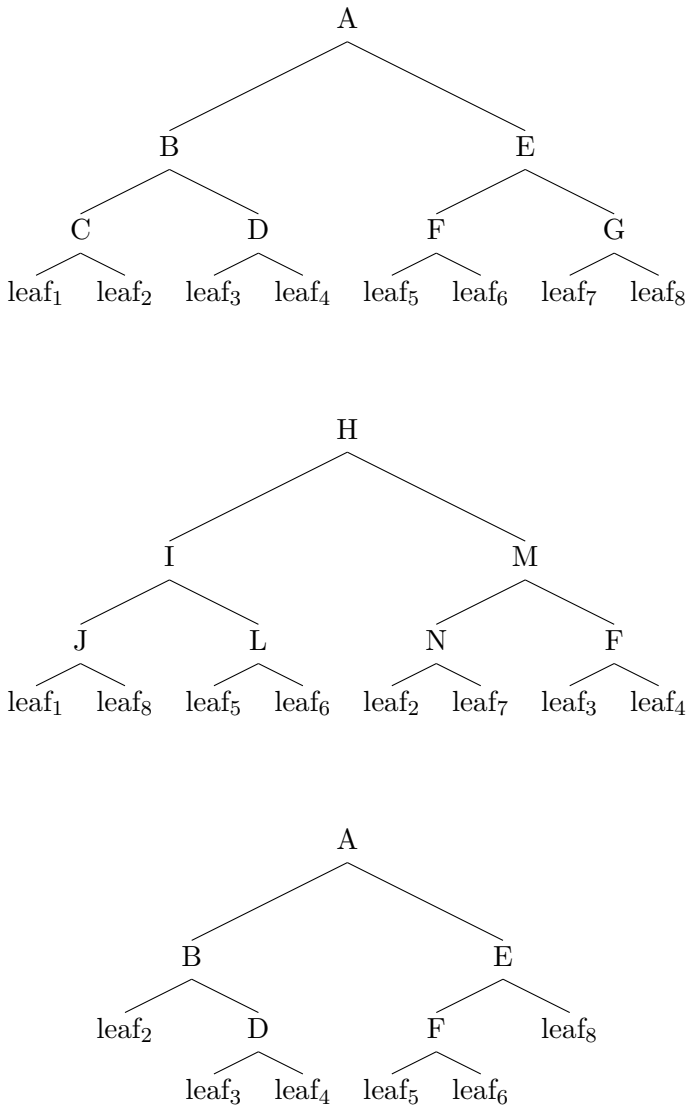

Figure 4.3: Two example binary trees and their resulting MAST

size table. Finally a traversal of the tree, separating it from the DAG is needed. More concretely, the algorithm is described as follows:

For each pair of nodes p and q, one from each input tree, we compute the agreement subtree $A_{p,q}$ for the two subtrees T_p and T_q having respectively p and q as roots. For such a pair of nodes there are three cases.

The first case is that p and q are both leaves. If the leaves are equal, i.e. they have the same name, then $A_{p,q}$ will be the tree consisting of exactly one leaf with that name. Otherwise $A_{p,q}$ is empty.

The second case is that we have a leaf and an internal node. Let's say q is the internal node having x and y as children. If the leaf corresponding to p is contained in T_q, it will also be contained in either T_x or T_y and $A_{p,q}$ will be the same as either $A_{p,x}$ or $A_{p,y}$. Since we do a postorder traversal of the trees, $A_{p,x}$ and $A_{p,y}$ have already been computed and $A_{p,q}$ can just be set to the largest of the two.

The third case is that both p and q are internal nodes, where p have children b and c and q have children x and y. In this case we can use Lemma 1 and determine which of the six cases gives the largest size. Again, all the agreement subtrees to consider have already been computed. If the largest size is $\#A_{b,x} + \#A_{c,y}$, then $A_{p,q}$ will be a tree where the root has the roots of $A_{b,x}$ and $A_{c,y}$ as children. Similarly if the largest size is $\#A_{b,y} + \#A_{c,x}$.

Option 2: Recursive Backtracking
While the first option is fairly simple to implement and understand, it can consume quite a bit of memory due to the trees we save that ultimately end up being irrelevant for the final result. Backtracking tries to rectify this problem by only computing and storing exactly the relevant subtrees for the final MAST. This is done by observing that the function in figure 4.2, on which the algorithm is based, is a maximization function. Therefore, the table is deterministically constructed, allowing us, for each cell, to determine what cells were responsible for the size it contains. By computing the entire size table first, we can start with the bottom right cell and recursively run back through the table determining all cells responsible for the MAST size. The tree is then constructed following the same logic used in Option 1.
Java inspired pseudo-code for the backtracking algorithm can be seen in Figure 4.4. Notice that the order of the if-cases in the pseudo code are significant, which was not previously the case when only computing the MAST size. The reason is simply that several of the cases might actually lead to the same MAST size, but not all will lead to a valid binary tree. For instance the second and third recursive case might lead to the same size, which means that at least one subtree does not contribute to the MAST. Choosing the third case therefore means that we end up producing a null-branch in the MAST, since this branch does not contribute to the MAST size.

```
Function RecBackTrack(Root1, Root2, ScoreMatrix)  {
  int Score = ScoreMatrix[Root1][Root2];

  /* Base Case */
  if (Type(Root1) = Type(Root2) = Leaf) {
    return Root1 = Root2 ? Root1 : null;
  }

  /* Recursive Case 1 */
  if (Type(Root1) = InternalNode) {
    int Score1 = ScoreMatrix[Root1.Child1][Root2];
    int Score2 = ScoreMatrix[Root1.Child2][Root2];

    if (Score = Score1) {
      return RecBackTrack(Root1.Child1, Root2);
    }
    else if (Score = Score2) {
      return RecBackTrack(Root1.Child2, Root2);
    }
  }

  /* Recursive Case 2 */
  if (Type(Root2) = InternalNode) {
    int Score1 = ScoreMatrix[Root1][Root2.Child1];
    int Score2 = ScoreMatrix[Root1][Root2.Child2];

    if (Score = Score1) {
      return RecBackTrack(Root1, Root2.Child1);
    }
    else if (Score = Score2) {
      return RecBackTrack(Root1, Root2.Child2);
    }
  }

  /* Recursive Case 3 */
  if(Type(Root1) = Type(Root2) = InternalNode) {
    int Score1 = ScoreMatrix[Root1.Child1][Root2.Child1];
    int Score2 = ScoreMatrix[Root1.Child2][Root2.Child2];
    int Score3 = ScoreMatrix[Root1.Child1][Root2.Child2];
    int Score4 = ScoreMatrix[Root1.Child2][Root2.Child1];

    if(Score = Score1 + Score2) {
      Tree subTree1 = RecBackTrack(Root1.Child1, Root2.Child1);
      Tree subTree2 = RecBackTrack(Root1.Child2, Root2.Child2);
      return /*internal node with subTree1 and
        subTree2 as children */
    }
    else if (Score1 = Score3 + Score4) {
      Tree subTree1 = RecBackTrack(Root1.Child1, Root2.Child2);
        Tree subTree2 = RecBackTrack(Root1.Child2, Root2.Child1);
      return /*internal node with subTree1 and
        subTree2 as children */
    }
  }
}
```

Figure 4.4: Java pseudo code for BackTracking

4.3 Evaluation

All in all, the $O(n^2)$ algorithm is fairly simple to implement and understand, but it also has its practical limitations when working with large trees. As we will see in the experiments section, the algorithm starts having trouble with binary trees of size larger than 6000 in terms of time and memory consumption. The main problem with the algorithm is that it considers too many combinations of subtrees that end up being irrelevant. Effectively most of the time spent filling out the size table is wasted. In our previous example of filling out the size table of 225 cells, only 25 cells ended up ultimately having an impact on the produced MAST, since those were the only ones accesed by the backtracking algorithm. The $O(nlogn)$ algorithm that we are about to examine improves on the 'wasted' work by only considering more relevant information between subtrees. Similarly to the naive algorithm, there are no worst-case or best-case trees to consider, since we always fill out the entire size table despite what the tree topology might look like.

Chapter 5

The $O(nlogn)$ algorithm

In this chapter we will walk through the $O(nlogn)$ algorithm for the MAST problem described in the paper [1] by Cole et. al. We will give a detailed description of each step of the algorithm and analyse its time complexity and space consumption.

5.1 Definitions and Notations

Trees

For a tree T, $T(x)$ will refer to the subtree of T having x as root. We will refer to the two input trees as T_1 and T_2 each of which have size n. Every leaf of T_1 has a unique name and for each of these leaves there is a corresponding leaf in T_2 with that same name. These two leaves are said to be twins.

Centroid Decompositions

The centroid decomposition for T_1 is a path of nodes $u_1, u_2, ..., u_p$ in the tree, starting from the root and ending at a leaf. Such a path is called a centroid path. It is referred to as π and has length p.

The centroid decomposition for T_2 is a set of disjoint centroid paths in the tree. The node in each path closest to the root of T_2 is referred to as the start node and $\pi(x)$ is the path having the node x as its start node. X will refer to the set of start nodes in the centroid decomposition for T_2.

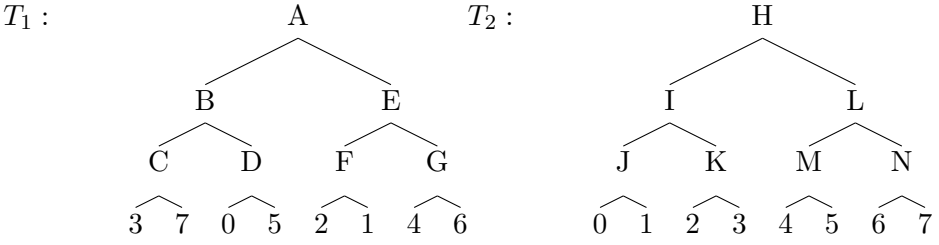

Figure 5.1: Two example input trees

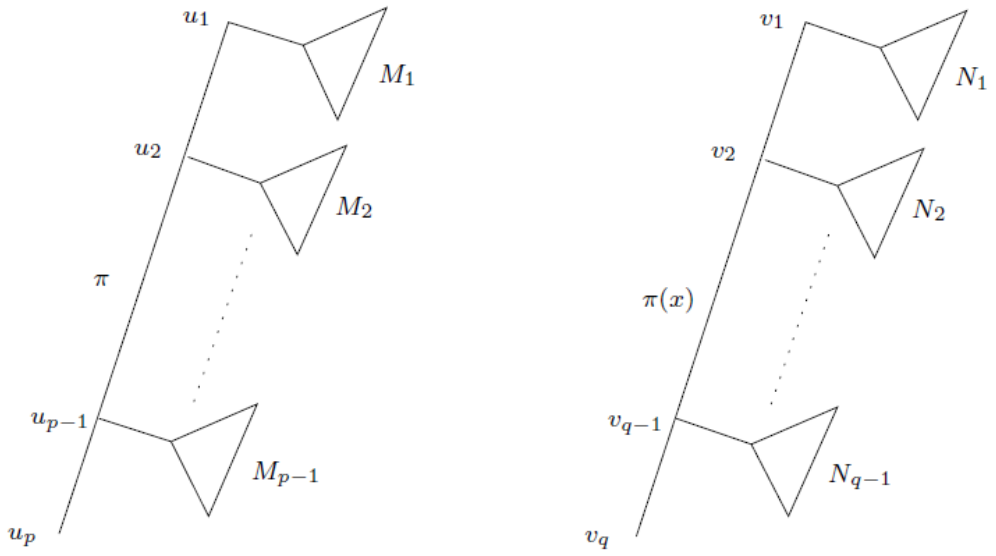

Figure 5.2: Centroid paths in T_1 and T_2 with π starting in root of T_1 and $\pi(x)$ starting in $x = v_1$ in T_2. [1]

Each node v in a centroid path has a side tree, which is the subtree having as root the child of v which is not on the centroid path. If v is a leaf, the side tree is v itself. The side tree of a node u_i in π is referred to as M_i and has size m_i. The side tree of a node v_j in a path $\pi(x)$ is referred to as N_j and has size n_j. Figure 5.1 illustrates the principle of centroid paths in T_1 and T_2.

Given a side tree M_i of $\pi, 1 \leq i \leq p-1$, S_i will refer to the subtree of T_2 induced by the twins of the leaves in M_i.

5.2 The Overall Strategy

The idea of the algorithm is that an Agreement Subtree A can be represented as a bipartite graph, where each edge represents a distinct subtree of A and the weight of the edge is the size of that subtree. This special kind of graph, called an Agreement Matching is obtained from a so called Matching Graph which contains edges that might be part of an agreement matching. An agreement matching is defined in a way that makes sure that it satisfies everything that an agreement subtree should satisfy, which is explained further in section 5.8.2. The purpose of the algorithm is now to compute the Largest Weight Agreement Matching (LWAM) for two input trees T_1 and T_2, which in the end can be translated to the Maximum Agreement Subtree.

The algorithm '$computeLWAM(T_1, T_2)$' has five overall steps, each of which will be covered in the following sections.

1. Create centroid decompositions.
2. Induce subtree S_i of T_2 for each side tree M_i of π.
3. Recursively construct $computeLWAM(M_i, S_i), 1 \leq i \leq p-1$.

4. Create a matching graph for each path $\pi(x)$ in the decomposition of T_2.

5. Compute agreement matchings and find the largest weight agreement matching.

5.3 Initial Setup

One thing we need to set up before starting the main algorithm, is to make sure that from any leaf in either T_1 or T_2, its twin in the other tree can be found in constant time. Note that this only needs to be done at the beginning of the algorithm and not for every recursive call.

First we want the names of all leaves to be numbers from 0 to $n-1$. This is done before starting the algorithm, e.g. when reading the input trees. Of course it should be possible to obtain the original names from the new names after the algorithm has finished, so one could for example store the original names in a list at indices corresponding to the new names.

Setting up twins is done by storing a list containing the leaves of T_2, where each leaf is at the index corresponding to its name. The twin of a leaf named i from T_1 can be looked up in constant time from the list for T_2 at index i. Then the twins of all leaves can be set in $O(n)$ time.

5.4 Centroid decompositions

The first step of the algorithm is to compute the centroid decompositions for the two trees. Recall that a centroid decomposition consists of one or more disjoint centroid paths through the tree. The first path starts at the root node in the tree. The next node is the child node holding the largest amount of leaves in its subtree and so forth. In case of a tie, we will pick the left node, but either one of the children could be picked. The path will continue until reaching a leaf.

For T_1, this is the only path in the centroid decomposition. For T_2, there is a new path starting at the root of each side tree if it is not a leaf, such that all internal nodes of T_2 is in a centroid path.

Creating the centroid decompositions can be done in $O(n)$ time in the following way:

First, at each node v in the two trees we need to store the number of leaves in the subtree having v as root. This can be done in a single post-order traversal of each tree. For a leaf, the number is 1. For an internal node, it is the sum of the numbers stored at its two children. Clearly this takes $O(n)$ time.

Now for the centroid decomposition of the first tree, we simply start with the root node and add the child node with the highest number stored. We continue until reaching a leaf. For the second tree, the same thing is done, but when adding a child node, another path is started at the child that wasn't added if it's not a leaf. While creating the paths for T_2 we will make each node on a path point to the start node of that path. This will be useful later in the algorithm.

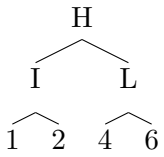

Figure 5.3: The subtree induced from $T_1(E)$ in figure 5.1

Each node is not visited more than once so the time complexity is $O(n)$.

5.5 Induced subtrees of T_2

Having created the centroid decompositions, the second step of the algorithm is to compute the induced subtrees of T_2 from each side tree of π. Given a side tree M_i of π, the induced subtree S_i of T_2 is the subtree containing only leaves that are twins of the leaves in M_i. Take figure 5.1 as an example. Let the side tree M_i be the subtree of T_1 with E as the root. Then S_i is the subtree having only the leaves named $1, 2, 4$ and 6 as shown in figure 5.3.

Given a tree T and an ordered set of leaves, $L = l_1, l_2, l_3, ..., l_{|L|}$, we will show how the subtree induced by L can be computed in $O(|L|)$ time after having preprocessed T in $O(|T|)$ time.

5.5.1 Preprocessing

To induce the subtree of T from L in $O(|L|)$ time, two requirements must be met.

1. The depth of any node in T can be retrieved in constant time.
2. The Least Common Ancestor (LCA) between two leaves can be computed in $O(1)$ time.

Both of these requirements can be achieved in $O(|T|)$ time.

The node depths can be set in $O(|T|)$ time by a simple pre-order tree traversal, where the depth of each node is set to its parent's depth plus one. The depth of the root is initialized to zero.

Computing LCA in $O(1)$ time is possible after having preprocessed T in $O(|T|)$ time. This is however a more complicated matter which we will not go into any detail with. Gusfield [5] explains an algorithm for how it can be done and our implementation directly follows this approach.

5.5.2 Inducing the subtree

The algorithm works in 3 steps:

1. Find the LCA of each consecutive pair of leaves in L and add them to L such that $LCA(l_i, l_{i+1}), 1 \leq i \leq |L| - 1$ is added between l_i and l_{i+1}.

   After this step, L will contain all nodes in the induced subtree.

   We will refer to the modified L as L'.

2. For each node v in L', find the closest node on either side, v_l and v_r, that has smaller depth than itself. v_l being the left node and v_r being the right node.

3. Construct the tree: The parent of each node $v \in L'$ will be whichever of v_l and v_r that has the greatest depth.

Since the LCA between two leaves can be computed in constant time, the first step takes $O(|L|)$ time.

The challenge was to perform step 2 in linear time, which is not covered by Cole et. al [1]. We did this by realizing that for any node v in L' that has greater depth than the node immediately to the right of it in L', v_l and v_r are the nodes immediately to the left and right in L' (because of how the LCAs were added to the set). This is also the case for the rightmost node in L'. By not considering these nodes any more, the same will be the case for each node that now has greater depth than the node to the right of it in L'. This resulted in the approach for step 2 seen in listing 5.1. Here, each node v of L' will eventually be added to S and removed when v_l and v_r has been updated correctly. For each node, a constant amount of computation needs to be performed, so the time complexity is $O(|L'|) = O(|L|)$.

Step three is done in a single iteration through L' where the parent of each node is found in constant time, giving a total runtime of $O(|L'|) = O(|L|)$.

```
Function induceSubtree(inputLeaves) {
  ...

  L‘ = The list of nodes computed in step 1;
  for each node v in L‘
    v_l = the node to the left of v in L‘;
    // if v is the leftmost node in L‘, v_l = null
    v_r = the node to the right of v in L‘;
    // if v is the rightmost node in L‘, v_r = null

  //Stack containing the first node in L‘
  S = new Stack();
  push L‘.get(0) on S;
  while(S is not empty)
  {
    v = S.peek();
    if(v_r == null || depth(v) > depth(v_r))
    {
      remove v from S;
      if(v_l != null) (v_l)_r = v_r;
      if(v_r != null) (v_r)_l = v_l;

      if(S is empty) push v_r on S;
    }
    else
      push v_r on S;
  }
  ...
}
```

Listing 5.1: Step 2 of induceSubtree

5.5.3 Computing the induced subtrees of T_2

Now that we can induce a subtree from a set of leaves L in $O(|L|)$ time, all we need is to find the leaves from which we can induce the subtree S_i of T_2, given the subtree M_i.

In order to induce subtree S_i, the input leaves need to be sorted by the order that they appear in T_2. Our approach to this is to first compute a sorted list of all leaves in T_1 according to T_2, then splitting the leaves into lists corresponding to the subtrees M_i, $1 \leq i \leq p-1$ and finally use their twins (which can be found in constant time) to induce the subtree S_i for each list.

The leaves of T_1 are sorted by iterating through the leaves of T_2, in order left to right, and adding the twin of each leaf to a list.

By iterating through each side tree M_i, we can use linear time to have each leaf store a number corresponding to the side tree to which it belongs. By a single iteration through the sorted list of leaves in T_1, we can use the stored numbers to split the leaves into separate lists, each corresponding to a side tree M_i. The twins of the leaves in these lists can now be used to compute the subtrees S_i, $1 \leq i \leq p-1$.

In our implementation, S_i does not actually consist of nodes from T_2. It is a new tree where each node contains a reference to the corresponding node in T_2 and vice versa.

5.5.4 Time Complexity

First, T_2 is preprocessed in $O(n)$ time. Next, the leaves of T_1 are sorted w.r.t. T_2 and split into lists corresponding to the side trees of π in $O(n)$ time. And finally each induced subtree S_i is computed in $O(m_i)$ time, $1 \leq i \leq p-1$. Since the sizes of all side trees of π sums to n, the total time complexity is $O(n)$.

5.6 Constructing Largest Weight Agreement Matchings recursively

When having computed the S_i subtrees, we can recursively construct the largest weight agreement matchings for each pair (M_i, S_i), $1 \leq i \leq p-1$. M_i and S_i are new trees, so before calling $computeLWAM(M_i, S_i)$, we need to transfer the twin pointers such that each leaf of M_i points to its twin in S_i. This is done while constructing the trees M_i and S_i. After having constructed the LWAMs for M_i and S_i, each node v of S_i will hold the LWAM and its weight for M_i and the subtree $S_i(v)$. We will explain later how this is achieved.

5.7 Matching Graphs

The next step of the algorithm is to compute the matching graphs. For each path $\pi(x), x \in X$ in the centroid decomposition for T_2, a matching graph $G(x)$ is created that will contain all edges that can be part of the largest weight agreement matchings between subtrees of T_1 and $T_2(x)$.

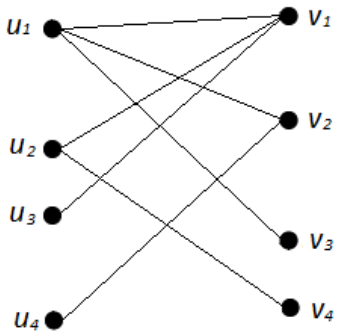

Figure 5.4: The matching graph $G(H)$ for the input trees in figure 5.1.

Let x be the start node of the centroid path $\pi(x)$ in T_2 with side trees $N_j, 1 \leq j \leq q$. Then the graph $G(x)$ is defined as follows:

- $G(x)$ consists of edges between two sets of nodes $L(x)$ and $R(x)$.
- $R(x)$ consists of the nodes from $\pi(x)$.
- $L(x)$ contains nodes from the centroid path π in T_1, for which there is an edge to some node in $R(x)$.
- The nodes of each set are in the order that they appear in the path that they are created from. I.e. the topmost node of a set is the node closest to the root and the bottommost is the one farthest from the root.
- An edge $(u_i, v_j), 1 \leq i \leq p-1, 1 \leq j \leq q$ exists if and only if $S_i \cap N_j \neq \emptyset$.

  Where $S_i \cap N_j$ is the intersection between the nodes of N_j and the nodes of T_2 corresponding to the nodes of S_i.
- An edge $(u_p, v_j), 1 \leq j \leq q$ exists if and only if the twin of u_p is in N_j.

Consider the trees in figure 5.1 where $\pi = \{u_1, u_2, u_3, u_4\} = \{A, B, C, 3\}$ and $\pi(H) = \{v_1, v_2, v_3, v_4\} = \{H, I, J, 0\}$. The matching graph $G(H)$ will for example get an edge from u_1 to v_1 since $S_1 \cap N_1 = \{L, 4, 6\}$ and an edge from u_2 to v_4 since $S_2 \cap N_4 = \{0\}$. The complete matching graph is the one showed in figure 5.4.

A matching graph $G(x)$ is used to create a Largest Weight Agreement Matching which can be translated directly to the MAST of T_1 and $T_2(x)$. For each $u_i \in L(x)$, starting from the bottom, we compute the LWAM containing only edges from u_i and nodes below u_i in $L(x)$. This will be covered in detail in section 5.8.

5.7.1 Creating the Matching Graphs

First of all, a graph is created for each centroid path in T_2, where all nodes in the path is added to the right set. In order to get access to such a graph in constant time from a node in the path, we will make each start node of the

paths point to the corresponding graph. Recall that each node of a path points to the start node.

Since the left set of each graph only contains nodes from π we can add these nodes and all edges by doing a walk through π. This walkthrough contains the following steps for each node $z \in S_i$ for each $u_i \in \pi, 1 \leq i \leq p-1$, where the nodes of S_i are traversed in preorder.

1. Find the corresponding node z' in T_2.

   This takes constant time since z has a reference to z'.

2. If z' is on a centroid path $\pi(x)$ (this is the case if z' has a pointer to the start node x of a path), do the following:

   Find the graph $G(x)$ that corresponds to the path $\pi(x)$. This is done in constant time since x has a pointer to the graph.

   Add u_i to $L(x)$ if it has not already been added.

   Add an edge between u_i and z' in $G(x)$.

   Repeat from step 2 using the parent of x as node z'.

3. If z' is not on a centroid path, repeat from step 2 using the parent of z'.

The loop stops either when reaching the root of T_2 or when reaching a node which is on the same path as the node in T_2 that corresponds to the parent of z in S_i. The second case means that the rest of the processing has already been done by the parent. This ensures that for each node visited in T_2, an edge is added to some graph, meaning that the process of adding all edges takes linear time with respect to the total number of edges in the graphs. In the article [1], an analysis proves that the total number of edges is bound by the sizes of the side trees of π namely $O(\sum_{i=1}^{p-1} m_i log\frac{n}{m_i})$ which is bound by $O(nlogn)$ since $\sum_{i=1}^{p} m_i = n$.

For the last node u_p in π, the process is very similar. z' starts being the twin of u_p and the loop continues until reaching the root of T_2.

For any node $z \in S_i, 1 \leq i \leq p-1$, z' being the corresponding node in T_2, every node v_j visited in the iteration will be the node that is the closest ancestor of z' in the path that v_j is on, which means that either $z' \in N_j$ or $z' = v_j$. Both cases mean that $S_i \cap N_j \neq \emptyset$ so an edge should be added.

For u_p, z' being the twin of u_p, if node v_j is visited in the iteration, then $z' \in N_j$ so an edge should be added.

5.7.2 Edge Weights

Each edge in a matching graph $G(x), x \in X$ is a multiedge consisting of a white, green and red edge, each with its own weight. The weights are defined as follows:

- White edge weight: $weight_w(u_i, v_j) =$ size of $MAST(M_i, N_j)$

- Green edge weight: $weight_g(u_i, v_j)$ = size of $MAST(M_i, T_2(v_j))$
- Red edge weight: $weight_r(u_i, v_j)$ = size of $MAST(T_1(u_i), N_j)$

Let's again consider the input trees of figure 5.1. The edges of the matching graph $G(H)$ in figure 5.4 are all multiedges that should get weights assigned to them. For example, the weight of the white edge (u_1, v_1) is 2, since $MAST(M_1, N_1)$ is the tree having only the two leaves named 4 and 6. The weight of the green edge (u_1, v_1) is 4, since $MAST(M_1, T_2(v_1))$ contains all four leaves of M_1.

As we will later explain, the size of a MAST is actually equal to the weight of the corresponding LWAM. Also, $LWAM(M_i, T_2) = LWAM(M_i, S_i), 1 \leq i \leq p-1$ which has already been computed in step 3 of the overall algorithm, so each node v of S_i holds the LWAM and its weight for M_i and $S_i(v)$, which is equal to the size of $MAST(M_i, T_2(v))$.

In order to compute the weight of each edge in constant time, a node $map(i, j)$ is defined for each multiedge (u_i, v_j). $map(i, j)$ is the node of S_i which is closest to the root and its corresponding node in T_2 is either a descendant of or equal to v_j. The edge (u_i, v_j) is added to a graph during an iteration of a node z in S_i, as described in the previous section. That node is exactly the node closest to the root in S_i which is also a descendant or equal to v_j in T_2, so $map(i, j) = z$.

The following sections explain how the edge weights are computed for a multiedge (u_i, v_j) in a matching graph $G(x), x \in X$. First, if one of u_i and v_j is a leaf, then the weight of each of three edges is 1. Otherwise, let $y = map(i, j)$.

White Edge Weight

y is either a descendant of or equal to v_j. In the former case, we know that $y \in N_j$ or the edge would not have been added. So the weight of the white edge is the weight of $LWAM(M_i, S_i(y))$, which can be looked up at y in constant time. In the latter case, one of $y's$ children is in N_j. That child z is then the node closest to the root of S_i which is in N_j, so the weight is equal to the weight of $LWAM(M_i, S_i(z))$, which is looked up at z. Since each node on a path of T_2 has a reference to the start node of that path, z can be found in constant time by finding out which child of v_j that does not have a reference to the same node as v_j, i.e. it must be the root of N_j. If that node is the first child of v_j, then z is the first child of y and vice versa.

Green Edge Weight

The green edge weight is equal to the weight of $LWAM(M_i, T_2(v_j))$. $S_i(y)$ is the subtree of $T_2(v_j)$ containing only leaves of M_i, so the weight is equal to the weight of $LWAM(M_i, S_i(y))$ which is looked up at y in constant time.

Red Edge Weight

Now let y be the root of N_j, which we showed how to find in constant time. Since y is a descendant of x and the matching graphs and LWAMs are computed in order bottom to top, the LWAMs of $G(y)$, has already been computed. In that computation, the LWAM containing only edges from u_i and nodes below u_i in $L(y)$ was also computed, corresponding to $LWAM(T_1(u_i), N_j)$. The red edge weight is the weight of that matching which is looked up in constant time. We know that u_i is in $L(y)$ since M_i and $N_j = T_2(y)$ intersects.

The weight of each edge is computed in constant time, so the time of computing all edge weights is linear w.r.t. the number of edges which we showed is limited by $O(nlogn)$.

5.8 Agreement Matchings

An agreement matching is a subset of a matching graph $G(x), x \in X$, that corresponds to an agreement subtree between some subtree of T_1 and some subtree of $T_2(x)$.

For each $u_i \in L(x)$, starting from the bottom, we will compute the largest weight agreement matching containing only edges from u_i and nodes below u_i in $L(x)$ and for each $v_j \in R(x)$, we will compute the largest weight agreement matching containing only edges from v_j and nodes below v_j in $R(x)$.

This will be done for all matching graphs in order bottom to top, where a graph $G(x)$ is above the graph $G(x')$ if and only if x is an ancestor of x' in T_2. This means that the final LWAMs to be computed are from the matching graph $G(r)$, where r is the root of T_2. The LWAM for T_1 and T_2 is then the LWAM containing only edges from u_1 and nodes below u_1 in $L(r)$.

5.8.1 Definitions

We will start by describing some definitions used in agreement matchings.

Nodes and Edges

For nodes and edges in the graph $G(x)$ we have the following definitions:

- $d_x(u_i)$: The 'degree' of a node $u_i \in L(x)$ is the number of white edges incident on it.

  A node in $L(x)$ is a 'singleton' node if it has degree 1.

  Edges incident on singleton nodes are called 'singleton' edges.

- $nsav(x)$: The number of nodes in $R(x)$ having at least one incident non-singleton edge.

For any edge (u_i, v_j), we say that u_i and v_j are adjacent. For two edges (a, b) and (a', b') in $G(x)$, we will say they 'cross' if a is above a' in $L(x)$ and b is below b' in $R(x)$. The two edges 'touch' if they are either crossing or $a = a'$ or $b = b'$. Finally (a, b) 'dominates' (a', b') if a is above a' and b is above b'.

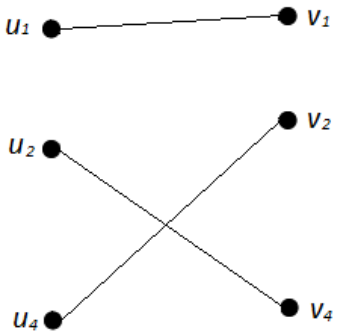

Figure 5.5: Example of an agreement matching.

Agreement Matchings

In the graph $G(x)$, a 'Proper Crossing' is either a single red edge, a single green edge or a crossing of a red edge (u_i, v_j) and a green edge (u'_i, v'_j) where u'_i is above u_i in $L(x)$.

Now an agreement matching is defined as a proper crossing and zero or more white edges, where all white edges dominate the edge(s) in the proper crossing and no white edge touches any other edge in the agreement matching.

This definition makes sure that an agreement matching holds exactly the information needed to uniquely determine the corresponding agreement subtree.

Consider again the matching graph from figure 5.4. Given this graph, an agreement matching could be the one in figure 5.5, where the edge (u_1, v_1) is white, the edge (u_2, v_4) is green and the edge (u_4, v_2) is red.

5.8.2 Agreement Matchings and MASTs

Consider the two trees T_1 and T_2 for which the graph $G(x), x \in X$ is constructed. T_1 has the centroid path π. For each multiedge $e = (u_i, v_j), u_i \in L(x), v_j \in R(x)$ of $G(x)$, the white edge of e corresponds to the tree $MAST(M_i, N_j)$, the green edge of e corresponds to $MAST(M_i, T_2(v_j))$ and the red edge of e corresponds to $MAST(T_1(u_i), N_j)$. Since the weight of each edge in $G(x)$ is the size of its corresponding tree, the largest weight agreement matching gives the set of trees, satisfying the conditions for an agreement matching, with the maximum number of leaves. These trees will be side trees on a path in the MAST between T_1 and $T_2(x)$.

Consider the MAST $\mathcal{A}$ between T_1 and $T_2(x)$. Each node of $\mathcal{A}$ will correspond to both a node in T_1 and a node in $T_2(x)$.

If $\mathcal{A}$ doesn't contain any node corresponding to a node from $\pi(x)$, then it must be equal to the MAST between T_1 and N_j for some sidetree N_j of $\pi(x)$. If u_i is the topmost node in π for which $M_i \cap N_j \neq \emptyset$, then $MAST(T_1, N_j) = MAST(T_1(u_i), N_j)$ which can be represented by a single red edge between u_i and v_j.

If $\mathcal{A}$ doesn't contain any node corresponding to a node from π, it is equal

to the MAST between M_i and $T_2(x)$ for some sidetree M_i of π. If v_j is the topmost node in $\pi(x)$ for which $M_i \cap N_j \neq \emptyset$, then $MAST(M_i, T_2(x)) = MAST(M_i, T_2(v_j))$. This MAST would be represented by a single green edge between u_i and v_j.

Now let's look at the final case where $\mathcal{A}$ contains nodes from both π and $\pi(x)$. Then at least one node in $\mathcal{A}$ will correspond to both a node in π and a node in $\pi(x)$. This will always be the case for the root of $\mathcal{A}$. Let w be one such node and $u_i \in \pi$ and $v_j \in \pi(x)$ be the nodes in T_1 and T_2 corresponding to w. If w is not the bottommost such node in $\mathcal{A}$, then one of its children will also correspond to a node in both π and $\pi(x)$, and the other will not. Let z be the child which does not. Then $\mathcal{A}(z)$ must be the MAST between M_i and N_j. This MAST can be represented by a single white edge between u_i and v_j. Now assume that w is indeed the bottommost such node in $\mathcal{A}$. If w is a leaf, then u_i is the leaf u_p and v_j is the leaf v_q and $\mathcal{A}$ is the MAST between $T_1(u_p)$ and N_q which is represented by a single red edge between u_p and v_q. If w is not a leaf, then the subtrees rooted at the two children of w are one of the following.

- One subtree is the MAST between M_i and N_j. The other is the MAST between $T_1(u_{i+1})$ and $N_{j'}$ for some $j', j < j' \leq q$.
- One subtree is the MAST between M_i and N_j. The other is the MAST between $M_{i'}$ and $T_2(v_{j+1})$ for some $i', i < i' \leq p$.
- One subtree is the MAST between $T_1(u_{i+1})$ and N_j. The other is the MAST between M_i and $T_2(v_{j+1})$.

The first case can be represented by a white edge between u_i and v_j and a red edge between $u_{i'}$ and $v_{j'}$, where $u_{i'}, i' > i$ is the topmost node below u_i in π for which $M_{i'} \cap N_{j'} \neq \emptyset$.

The second case can be represented by a white edge between u_i and v_j and a green edge between $u_{i'}$ and $v_{j'}$, where $v_{j'}, j' > j$ is the topmost node below v_j in $\pi(x)$ for which $M_{i'} \cap N_{j'} \neq \emptyset$.

The third case can be represented by a red edge between $u_{i'}$ and v_j, where $u_{i'}, i' > i$ is the topmost node below u_i in π for which $M_{i'} \cap N_j \neq \emptyset$ and a green edge between u_i and $v_{j'}$, where $v_{j'}, j' > j$ is the topmost node below v_j in $\pi(x)$ for which $M_i \cap N_{j'} \neq \emptyset$.

All possible cases of $\mathcal{A}$ have been covered and in each case the subtrees of $\mathcal{A}$ could be represented by edges in an agreement matching.

5.8.3 The Weighted Search Tree

In order to compute the LWAMs in $O(nlogn)$ time, we will create a weight balanced binary search tree $\mathcal{T}$ for each matching graph $G(x)$. This tree will be used to store the information needed to determine which edges of $G(x)$ forms the LWAMs. The structure of the tree will ensure that filling the tree and extracting the LWAMs will not take more than $O(nlogn)$ time.

$\mathcal{T}$ is created from the nodes in $R(x)$ such that $\mathcal{T}$ will have a leaf for each node in $R(x)$. The order of the leaves from left to right should correspond to the top-down order of the nodes in $R(x)$ such that the leftmost leaf in $\mathcal{T}$ corresponds to the topmost node in $R(x)$. Leaf v_j is given weight $n_j + \frac{|T_2(x)|}{nsav(x)}$ if a non-singleton edge in $G(x)$ has endpoint in v_j and weight n_j otherwise. The reason for choosing these weights is explained later.

Recall that for each $u_i \in L(x)$, we want to compute the LWAM containing only edges from u_i and nodes below u_i in $L(x)$. This is done by processing each node $u_i \in L(x)$ in order bottom-to-top, where each node in $R(x)$, that has an edge to u_i, is searched for in $\mathcal{T}$, while storing information about the edges in $\mathcal{T}$. After processing u_i, the LWAM can be extracted from $\mathcal{T}$.

Constructing the Weighted Search Tree

For a matching graph $G(x)$, the weighted search tree can be constructed in $O(|R(x)|)$ time. Since $O(\sum_{x \in X} |R(x)|) = O(n)$, all the search trees needed for the matching graphs can be constructed in $O(n)$ time. This is not covered by Cole et. al. [1], but in the paper [3] by Fredman, it is described how a balanced search tree can be constructed in $O(N)$ time, given $O(N)$ probabilities which should be associated with the nodes of the tree.

Though not completely applicable to our situation, we used it as a basis for the following algorithm. Each node in the tree will get an index number which is used for searching.

- Given weights $w_0, w_1, ..., w_{|R(x)|-1}$, construct two lists of sums $L_0, ..., L_{|R(x)|}$ and $R_0, ..., R_{|R(x)|}$, where

  $L_j = \sum_{k=0}^{j-1} w_k, 0 < j \leq |R(x)|$, $L_0 = 0$

  $R_j = \sum_{k=j}^{|R(x)|-1} w_k, 0 \leq j < |R(x)|$, $R_{|R(x)|} = 0$

- Construct the tree using the recursive procedure $constructST(w_0, w_1, ..., w_{|R(x)|-1})$ which returns the root of the search tree.

- $constructST(w_i, ..., w_j)$ has the following steps:

  If $i + 1 = j$ then create and return a node with index j having a left child with index i and a right child with index j.

  Otherwise, determine the smallest $k, i < k \leq j$ such that

  $L_k - L_i \geq R_k - R_{j+1}$

  Define a node with index k having $constructST(w_i, ..., w_{k-1})$ as left child and $constructST(w_k, ..., w_j)$ as right child.

  If $i = k - 1$, the left child is just a leaf with index i.

  If $j = k$, the right child is just a leaf with index j.

Determining k from the weights $w_i, ..., w_j$ is done by finding the smallest k satisfying $L_k - L_i \geq R_k - R_{j+1}$. This can be done in $O(log(k - i))$ time as follows:

- For the middle index $k' = i + (j - i + 1)/2$, determine whether $L_{k'} - L_i \geq R_{k'} - R_{j+1}$.
- If so, let $k' = i + 1, k' = i + 2, k' = i + 4, k' = i + 8, ...$ until $L_{k'} - L_i \geq R_{k'} - R_{j+1}$.

  This takes $O(log(k' - i))$ time for the final value of k'.

  $k' - i \leq 2(k - i)$, so $O(log(k' - i)) = O(log(k - i))$.

  In the interval between k' and the previous value of k', use binary search to find the smallest k' satisfying $L_{k'} - L_i \geq R_{k'} - R_{j+1}$.

  This takes $O(log(k - i))$ time.

  Then $k = k'$.
- The case where $L_{k'} - L_i < R_{k'} - R_{j+1}$ is treated symmetrically.

The linear time complexity can be explained as follows:

Let $F(N)$ be the time taken to construct a search tree with N internal nodes. $F(N)$ satisfies the following inequality

$F(N) \leq max\{A + Blog(k) + F(k - 1) + F(N - k), 1 \leq k \leq (N + 1)/2\}$

where A and B are constants and N and k are natural numbers. This function can be proven to grow at most linearly with N, thus constructing a search tree has linear runtime w.r.t. the number of internal nodes. The complete proof can be seen in section 5.13. $|R(x)|$ is greater than the number of internal nodes, so a weight balanced binary search tree can be constructed in $O(|R(x)|)$ time.

Searching

In order to minimize the runtime when searching for leaves in the search tree $\mathcal{T}$, we will specify how two kinds of searches should be performed.

In a search tree $\mathcal{T}$, each node v will hold an index number $index(v)$. Searching for a leaf in $\mathcal{T}$ with index number i, is done in the standard way by starting at the root and picking the left child if $i <$ *the current node index*, and the right child otherwise until reaching the desired leaf.

Each node v will also hold the lowest index number $low(v)$ and largest index number $high(v)$ in its subtree, which is used when searching for an ordered subset of $R(x)$. These numbers are added while creating the tree. For the root r, $low(r)$ is 0 and $high(r)$ is $|R(x)| - 1$. For a left child v, $low(v) = low(parent(v))$ and $high(v) = index(parent(v)) - 1$. For a right child v, $low(v) = index(parent(v))$ and $high(v) = high(parent(v))$.

Given the subset $v_i, ..., v_j$ of $R(x)$ in bottom to top order, let $l_i, ..., l_j$ be the corresponding leaves in $\mathcal{T}$. Searching for these leaves is done with a sequential search by first searching for the leaf l_i in the standard way. When at a leaf $l_{i'}, i' < j$, searching for $l_{i'+1}$ is done by starting at $l_{i'}$ and go up through the tree until reaching a node whose lowest index number is less than or equal to $index(l_{i'+1})$. $l_{i'+1}$ is now found by searching in the standard way starting at the node just reached.

Given a subset of $R(x)$ in top to bottom order, searching for the corresponding leaves in $\mathcal{T}$ is done similarly, but considering the largest index numbers instead of the lowest.

Properties of the Search Tree

A search tree $\mathcal{T}$ is created such that a leaf l corresponding to $v_j \in R(x)$ has weight $n_j + \frac{|T_2(x)|}{nsav(x)}$ if a non-singleton edge in $G(x)$ has endpoint in v_j and weight n_j otherwise. Balancing the tree with these weights ensures that the depth of any leaf l corresponding to node $v_j \in R(x)$ is $O(log\frac{|T_2(x)|}{n_j})$. When searching for an ordered subset of k leaves in $\mathcal{T}$ as specified in the previous section, the structure of $\mathcal{T}$ also ensures that the number of nodes visited is $O(k * log\frac{nsav(x)}{k})$. Each of these nodes is visited a most thrice. The reasoning behind these claims can be found in [1].

5.8.4 Computing the Largest Weight Agreement Matchings

Computing the LWAMs for the graph $G(x)$ is done by processing each node of $L(x)$ in order starting with the bottom node. Recall that we will compute the LWAM containing only edges from u_i and nodes below u_i in $L(x)$, for each $u_i \in L(x)$. This is done by adding that information to $\mathcal{T}$ while processing the nodes of $L(x)$.

First, we will describe the information that needs to be stored in $\mathcal{T}$. Next, we will explain how nodes are processed and how to extract the LWAMs from $\mathcal{T}$. Finally we explain the total time complexity of this step.

Auxiliary Information in $\mathcal{T}$

While processing the edges of $G(x)$ we will maintain information at each node of $\mathcal{T}$ about the LWAM of the currently processed edges. For a node $z \in \mathcal{T}$, we define $anc(z)$ as the set of ancestors of z, including z itself in order top to bottom. $lfringe(z)$ is defined as the left children of the vertices in $anc(z)$ which are not in $anc(z)$ themselves. $lfringe(z)$ will only contain nodes with indices less than $index(z)$. $rfringe(z)$ is defined analogously where the nodes all have indices greater or equal to $index(z)$. $anc(z)$ need not be stored in $\mathcal{T}$, but can be created while searching for z in $\mathcal{T}$. Likewise, $lfringe(z)$ and $rfringe(z)$ can be created by iterating through $anc(z)$. After processing an edge (u, v), we say that the edge is in $\mathcal{T}$. For a node $z \in \mathcal{T}$ we also say that the edge (u, v) is in $\mathcal{T}(z)$ if $\mathcal{T}(z)$ contains the leaf corresponding to v.

The following information is maintained at each node $z \in \mathcal{T}$. This is taken directly from [1], but slightly elaborated.

- $g(z)$: After updating $g(z)$, it will hold the heaviest green edge in $\mathcal{T}$ which forms a proper crossing with each red edge in $\mathcal{T}(z)$.

  However, $g(z)$ is not maintained at all times, but the correct edge is always $max_{z' \in anc(z)} g(z')$.

If there are no red edges in $\mathcal{T}(z)$ when updating $g(z)$, then $g(z)$ will be the heaviest green edge in $\mathcal{T}(z')$, for all $z' \in rfringe(z)$.

- $x(z)$: This is the largest weight proper crossing, that is not a single red edge, among the edges in $\mathcal{T}(z)$.
- $m(z)$: This is the largest weight agreement matching in $\mathcal{T}$ containing a white edge such that the topmost white edge is in $\mathcal{T}(z)$.
- $y(z)$: This the largest weight proper crossing, which is not a single edge, such that the green edge is in $\mathcal{T}$ but not in $\mathcal{T}(z)$, the red edge is in $\mathcal{T}(z)$, and the green edge does not form a proper crossing with each of the red edges in $\mathcal{T}(z)$.
- $r(z)$: This is simply the heaviest red edge in $\mathcal{T}(z)$.

Processing Nodes

For each node $u_i \in L(x)$, starting from the bottommost, we need to process each edge $e = (u_i, v_j)$ incident on it. This is done by searching for the leaf corresponding to v_j in $\mathcal{T}$ and updating information at nodes in $\mathcal{T}$ so that e is contained in $\mathcal{T}$. Leaves are searched for as described in section 5.8.3, where a standard search is used if $d_x(u_i) = 1$ and a sequential search is used if $d_x(u_i) > 1$.

In the following, we will explain how each colour edge is processed when $d_x(u_i) = 1$ and afterwards when $d_x(u_i) > 1$. For a green edge g and a red edge r, $g + r$ will refer to the proper crossing between g and r.

Let $e = (u_i, v_j)$ be the edge in $G(x)$ that should be processed and l be the leaf in $\mathcal{T}$ corresponding to v_j. We know that all edges incident on nodes below u_i in $L(x)$ have already been processed.

Processing white singleton edges

The only values that might change in $\mathcal{T}$ when adding a white edge is $m(z), z \in anc(l)$. First, $anc(l)$ is found while searching for l in $\mathcal{T}$ and $rfringe(l)$ is subsequently created. Now the LWAM with e as topmost edge needs to be determined. This is done by finding the LWAM among the edges in $\mathcal{T}$ containing only edges dominated by e and then adding e as the topmost white edge. Since the nodes of $L(x)$ are processed starting from the bottom and e is a singleton edge, we know that $\mathcal{T}$ does not contain any edges incident on u_i or nodes above u_i in $L(x)$. The subtrees of nodes in $rfringe(l)$ contain exactly the edges in $\mathcal{T}$ only incident on nodes below v_j in $R(x)$, so these are the subtrees that should be considered. There are four possible cases for the LWAM:

- The LWAM contains one or more white edges.

  This matching is given by $max_{z \in rfringe(l)} m(z)$

- The LWAM is a single green edge or a green-red crossing where both edges are in the subtree $\mathcal{T}(z)$ for some $z \in rfringe(l)$.

  This matching is given by $max_{z \in rfringe(l)} x(z)$

- The LWAM is a single red edge or a green-red crossing where the red edge is in the subtree $\mathcal{T}(z)$ for some $z \in rfringe(l)$. The green edge forms a proper crossing with all red edges in $\mathcal{T}(z)$.

  This matching is given by

  $$max_{z \in rfringe(l)}(max_{z' \in anc(z)} g(z')) + r(z), \text{ if the edge } r(z) \text{ exists}$$

- The LWAM is a green-red proper crossing where the red edge is in the subtree $\mathcal{T}(z)$ for some $z \in rfringe(l)$. The green edge is not in $\mathcal{T}(z)$ and does not form a proper crossing with all red edges in $\mathcal{T}(z)$. This matching is given by $max_{z \in rfringe(l)} y(z)$

These LWAMs can be found doing a single iteration through $rfringe(l)$ starting from the top. The four cases cover all possible LWAMs containing only edges dominated by e, so adding e as the topmost white edge to the largest of these agreement matchings, gives us the LWAM that should replace $m(z), z \in anc(l)$ if it has larger weight than the agreement matching already stored.

In order to avoid copying the agreement matching just found, which would take linear time w.r.t. the number of edges, the new agreement matching can be stored as the white edge e and a pointer to the other agreement matching. This will also minimize the amount of space needed.

Processing red singleton edges

The values that might change in $\mathcal{T}$ when adding a red edge is $y(z), g(z)$ and $r(z), z \in anc(l)$ and $g(z'), z' \in lfringe(l) \cup rfringe(l)$. Since none of the green edges in $\mathcal{T}$ can form a proper crossing with e, $y(z)$ might change for $z \in anc(l)$. So $y(z)$ is updated:

$$y(z) = max \begin{cases} y(z) \\ (max_{z' \in anc(z)} g(z')) + r(z) & \text{if both edges exist} \end{cases}$$

For the same reason $g(z)$ should be reset for all $z \in anc(l)$. However, in order to not lose the information at $g(z)$, $g(z')$ needs to be updated to $max_{z'' \in anc(z')} g(z'')$ for each $z' \in lfringe(l) \cup rfringe(l)$ before resetting. Finally, $r(z)$ need to be updated for $z \in anc(l)$ if the weight of e is greater than that the current value of $r(z)$. $r(z) = max\{r(z), e\}$.

Updating all these values only requires a single iteration through $anc(l)$ starting from the root, where both the nodes of $anc(l)$ and children of these nodes are updated.

Processing green singleton edges

Adding a green edge to $\mathcal{T}$ might change the values $g(z), z \in lfringe(l)$ and $x(z), z \in anc(l)$. When adding the green edge e, it will form proper crossings with all red edges in $\mathcal{T}$ incident on nodes in $R(x)$ above v_j. This is true since each red edge in $\mathcal{T}$ is incident on a node in $L(x)$ below u_i. Therefore

$g(z), z \in lfringe(l)$ is updated to e if the weight of e is greater than that of the currently heaviest green edge forming a proper crossing with all red edges in $\mathcal{T}(z)$. $g(z) = max\{e, max_{z' \in anc(z)} g(z')\}, z \in lfringe(l)$. The value $x(z), z \in anc(l)$ might also need to be updated if e or a proper crossing between e and a red edge in $\mathcal{T}(z)$ is heavier than the current value of $x(z)$. $x(z) = max\{x(z), e + max(r(z'))\}, z \in anc(l)$. Here $max(r(z'))$ maximizes over the vertices $z' \in lfringe(l)$ which are descendants of z.

The values $g(z), z \in lfringe(l)$ are updated during a single iteration through $anc(l)$ starting at the root. The values $x(z), z \in anc(l)$ are updated during a single iteration through $anc(l)$ starting at l.

Processing nodes with degree > 1

Recall that we process the edges of $G(x)$ by iterating through $L(x)$ starting from the bottommost node, and process the edges incident on each of these nodes. For a node $u_i \in L(x)$ we will start by processing the white edges incident on u_i in order, starting from the topmost edge, i.e. the edge incident on the topmost node in $R(x)$. The reason for doing top-down processing is explained later. For these edges, the leaves in $\mathcal{T}$ that needs to be searched for, are found with a sequential search as described in section 5.8.3.

Next, we process the red and green edges incident on u_i in order, starting from the bottommost edge. This ensures that when processing a green edge $e = (u_i, v_j)$, no red edges incident on u_i and on a node in $R(x)$ above v_j has been processed. Searching for leaves in $\mathcal{T}$ is also done with sequential search.

Processing edges from non-singleton nodes using sequential search is important for the runtime of this step. Let $l_1, l_2, ..., l_k$ be the leaves in the search tree that are searched for when processing $u_i \in L(x)$. The only nodes in the search tree that need to be updated are nodes from the set $anc(l_1) \cup anc(l_2) \cup ... \cup anc(l_k)$, but in order to ensure that only a constant amount of work is performed at each node, we need a slightly different approach than for the singleton edges. This was not covered by Cole et. al. [1], so we came up with our own approach.

In the following, we will explain how each colour edge is processed when $d_x(u_i) > 1$. Let $e_1, e_2, ...$ be the edges in $G(x)$ incident on u_i and $l_1, l_2, ...$ be the corresponding leaves in $\mathcal{T}$ that should be searched for. The white edges are in top-down order and the red and green edges are in bottom-up order. Again, we know that all edges incident on nodes below u_i in $L(x)$ have already been processed.

Processing white non-singleton edges

For processing white non-singleton edges, we will introduce some extra information that should be stored in $\mathcal{T}$, but only while processing the white edges incident on u_i.

The following information is stored at each node $z \in anc(l_1) \cup anc(l_2) \cup ...$ as they are being visited the first time.

- $rm(z)$: This is the heaviest of the agreement matchings of $m(z'), z' \in rfringe(z)$.

$$rm(z) = max_{z' \in rfringe(z)} m(z')$$

- $rx(z)$: This is the heaviest of the agreement matchings of $x(z'), z' \in rfringe(z)$.

  $$rx(z) = max_{z' \in rfringe(z)} x(z')$$

- $ry(z)$: This is the heaviest of the agreement matchings of $y(z'), z' \in rfringe(z)$.

  $$ry(z) = max_{z' \in rfringe(z)} y(z')$$

- $ag(z)$: This is the heaviest of the edges $g(z'), z' \in anc(z)$.

  $$ag(z) = max_{z' \in anc(z)} g(z')$$

- $gr(z)$: This is the heaviest single red edge or green-red proper crossing where the red edge is in the subtree $\mathcal{T}(z')$ for some $z' \in rfringe(z)$. The green edge forms a proper crossing with all red edges in $\mathcal{T}(z')$.

  $$gr(z) = max_{z' \in rfringe(z)}(max_{z'' \in anc(z')} g(z'')) + r(z'), \text{ if } r(z') \text{ exists.}$$

Searching for leaves in $\mathcal{T}$ is done by starting from the root of $\mathcal{T}$. Thus finding the above values can be done in constant time for each node z in the following way, where $p(z)$ is the parent node of z:

- $ag(z) = max\{ag(p(z)), g(z)\}$

- If z is the left child of its parent $p(z)$ and z' is the right child, then

  $$rm(z) = max\{rm(p(z)), m(z')\}$$
  $$rx(z) = max\{rx(p(z)), x(z')\}$$
  $$ry(z) = max\{ry(p(z)), y(z')\}$$
  $$gr(z) = max \begin{cases} gr(p(z)) \\ ag(z') + r(z') \quad \text{if } r(z') \text{ exists} \end{cases}$$

- Otherwise if z is the right child of its parent $p(z)$, then

  $$rm(z) = rm(p(z))$$
  $$rx(z) = rx(p(z))$$
  $$ry(z) = ry(p(z))$$
  $$gr(z) = gr(p(z))$$

Processing the white edges in top-down order ensures that when processing a white edge $e_{j'} = (u_i, v_j)$, no other edge incident on u_i and on a node in $R(x)$ below v_j has been processed. This means that the agreement matchings $rm(l_{j'}), rx(l_{j'}), ry(l_{j'})$ and $gr(l_{j'})$ only contains edges dominated by $e_{j'}$.

When processing the topmost edge e_1, we will search for leaf l_1 in $\mathcal{T}$ with a standard search while storing the above values at the nodes of $anc(l_1)$. Now as when processing singleton white edges, we need to find the LWAMs containing only edges dominated by e_1 and then adding e_1 as the topmost white edge.

These LWAMs are exactly the ones we have now stored at l_1: $rm(l_1), rx(l_1), ry(l_1)$ and $gr(l_1)$. The largest weight agreement matching with e_1 as topmost white edge is therefore the heaviest of these agreement matchings with e_1 added as topmost white edge. Now the values $m(z), z \in anc(l_1)$ needs to be updated with this LWAM, but since the remaining edges incident on u_i might have some of the same ancestors, we will instead update the values while searching for these edges.

When searching for a leaf $l_{j'}, j' > 1$, starting at $l_{j'-1}$, we first go up through the tree until reaching the first node which is an ancestor of both $l_{j'}$ and $l_{j'-1}$ and afterwards search for $l_{j'}$ in the standard way while updating values at the nodes visited as before. While going up through the tree, we can update the nodes visited with the LWAM found when processing the previous edge $e_{j'-1}$, since these nodes are all ancestors of $l_{j'-1}$. Reaching a node z where $m(z)$ is heavier than the agreement matching we are currently updating with, means that it is an agreement matching found when processing one of the previous edges, so the following nodes to be visited should instead be updated with that matching.

Having processed the last edge, we will continue to the root of $\mathcal{T}$ in order to update the remaining ancestors.

Processing red and green non-singleton edges

The only extra information that should be stored in $\mathcal{T}$ while processing red and green edges incident on u_i is $ag(z)$ for each node $z \in anc(l_1) \cup anc(l_2) \cup ...$, where $ag(z)$ is defined as when processing white non-singleton edges and found the same way while searching for leaves in $\mathcal{T}$.

The procedure is similar to processing white non-singleton edges. The leaves of $\mathcal{T}$ are searched for using sequential search, but starts from the bottommost edge. In the end, we return to the root of $\mathcal{T}$.

As when processing singleton edges, adding a red edge $e_{j'}$ to $\mathcal{T}$ might changes the values $y(z), g(z)$ and $r(z), z \in anc(l_{j'})$ and $g(z'), z' \in lfringe(l_{j'}) \cup rfringe(l_{j'})$. For $z \in anc(l_{j'})$, $y(z)$ should be updated:

$$y(z) = max \begin{cases} y(z) \\ (max_{z' \in anc(z)} g(z')) + r(z) \quad \text{if both edges exist} \end{cases}$$

$$= max \begin{cases} y(z) \\ ag(z) + r(z) \quad \text{if both edges exist} \end{cases}$$

For each $z \in lfringe(l_{j'}) \cup rfringe(l_{j'})$, $g(z)$ should be updated to $max_{z' \in anc(z)} g(z') = ag(z)$ and finally for $z \in anc(l_{j'})$, $g(z)$ should be reset and $r(z)$ updated to $max\{r(z), e_{j'}\}$.

Having stored $ag(z)$ at each node z visited while searching, all of the above values can be updated in constant time for each node while going up through the tree.

Notice that when resetting $g(z)$ for some node z, $ag(z)$ also needs to be reset since z might be visited again and the value of $ag(z)$ is no longer valid.

Adding a green edge $e_{j'}$ to $\mathcal{T}$ might changes the values $g(z), z \in lfringe(l_{j'})$ and $x(z), z \in anc(l_{j'})$. For $z \in lfringe(l_{j'})$, $g(z)$ is updated to $max\{e_{j'}, max_{z' \in anc(z)} g(z')\} = max\{e_{j'}, ag(z)\}$. Since $ag(z)$ is stored at each node z visited while searching, $g(z), z \in lfringe(l_{j'})$ is updated in constant time for each node while going up through the tree. For $z \in anc(l_{j'})$, $x(z)$ should be updated to $max\{x(z), e_{j'} + max(r(z'))\}$, where $max(r(z'))$ maximizes over the vertices $z' \in lfringe(l_{j'})$ which are descendants of z. This is also done in constant time for node $z \in anc(l_{j'})$ while going up through the tree by keeping track of the heaviest red edge hre of $r(z'), z' \in lfringe(l_{j'})$, where z' is a descendant of z.

If z is the first node which is an ancestor of both $l_{j'}$ and $l_{j'+1}$ (assuming that $l_{j'+1}$ exists), we will store $e_{j'}$ at z as $ge(z)$. Now when going up through the tree after having reached leaf $l_{j'}$, we will also keep track of which green edge hge, incident on u_i is the heaviest that has been added to the subtree rooted at the current node z. When at $l_{j'}$, hge is obviously $e_{j'}$, and when at a node $z \in anc(l_{j'})$ it is simply the heaviest edge of hge and $ge(z)$. $x(z)$ can now be updated to $max\{x(z), hge + hre\}$ in constant time.

Extracting the Largest Weight Agreement Matchings

First, we need to compute the LWAM containing only edges from u_i and nodes below u_i in $L(x)$, for each $u_i \in L(x)$. For any $u_i \in L(x)$, all the information needed to compute such a matching is stored at the root r of $\mathcal{T}$ after having processed the edges incident on u_i and before processing edges incident on the above nodes in $L(x)$. The LWAM is extracted in the following way:

If the LWAM contains a white edge it is simply given by $m(r)$. If the LWAM is a proper crossing, but not a single red edge, it is given by $x(r)$. And finally, if the LWAM is a single red edge, it is given by $r(r)$. Therefore the LWAM is the heaviest of these three matchings. This LWAM is saved, such that it can be retrieved in constant time given u_i and x.

For each $v_j \in R(x)$, we also need to compute the LWAM containing only edges incident on v_j and nodes below v_j in $R(x)$. For leaf $v_j \in R(x)$, such a matching will correspond to the MAST between T_1 and $T_2(v_j)$, so this is the LWAM that needs to be stored at $v_j \in T_2$. The LWAMs are extracted from $\mathcal{T}$, after all edges in $G(x)$ have been processed. Each node of $R(x)$ is processed in order, starting from the bottom:

For node $v_j \in R(x)$, let l be the leaf in $\mathcal{T}$ corresponding to v_j and let $M(v_j)$ be the LWAM containing only edges incident on v_j and nodes below v_j in $R(x)$. $M(v_j)$ is either equal to $M(v_{j+1}), j < |R(x)|$, or it is the LWAM having a white edge incident on v_j as the topmost edge, or it is the largest proper crossing having a red edge incident on v_j, or it is a single green edge incident on v_j.

Thus, $M(v_j)$ is given by:

$$M(v_j) = max \begin{cases} M(v_{j+1}) & \text{if } j < |R(x)| \\ m(l) & \\ y(l) & \\ max_{z \in anc(l)} g(z) + r(l) & \text{if r(l) exist} \\ x(l) & \end{cases}$$

After having computed the LWAMs for all matching graphs, the LWAM corresponding to the MAST between T_1 and T_2 can be retrieved in constant time given the root u_1 of T_1 and the root v_1 of T_2. This LWAM was extracted from the search tree for the matching graph $G(v_1)$ and is the LWAM containing edges from u_1 and nodes below u_1 in $L(v_1)$ which is all possible edges and therefore corresponds to the MAST between T_1 and T_2.

Analysis

In [1], Cole et. al. proves two important lemmas:

Lemma 2. *Consider matching graph $G(x)$. Then*
$\sum_{i|d_x(u_i)>1} d_x(u_i) log \frac{nsav(x)}{d_x(u_i)} \leq \sum_{i|d_x(u_i)>1} d_x(u_i) log \frac{n}{m_i}$

Lemma 3. *Consider node $u_i \in \pi$. Then*
$\sum_{x \in X|d_x(u_i)>1} d_x(u_i) = O(m_i)$

Consider the matching graph $G(x)$ with search tree $\mathcal{T}$ and a node $u_i \in L(x)$. If $d_x(u_i) > 1$ and L is the ordered set of leaves in $\mathcal{T}$ that corresponds to the nodes in $R(x)$ incident on the edges from u_i, then it follows from the properties of the search tree and Lemma 2 that the amount of nodes in the search tree that is visited when searching for the leaves of L is $O(d_x(u_i) log(\frac{n}{m_i}))$.

Considering node $u_i \in \pi$, it follows from Lemma 3 that the amount of nodes visited in all search trees over all matching graphs $G(x)$ with $d_x(u_i) > 1$ is $O(m_i log \frac{n}{m_i})$.

For each node $u_i \in \pi$ over all matching graphs $G(x)$ with $d_x(u_i) = 1$, it turns out that the amount of nodes visited in all search trees is also $O(m_i log \frac{n}{m_i})$. This can be seen from the analysis in [1].

Thus the total amount of nodes visited in all search trees, for any node $u_i \in \pi$, is $O(m_i log \frac{n}{m_i})$.

Time Complexity

When processing an edge, whether it is a singleton or a non-singleton edge, each node of the search tree that is visited, is visited no more than a constant number of times. When visiting a node, only a constant amount of work is performed, so for any node $u_i \in \pi$, the time taken to process u_i over all matching graphs must be $O(m_i log \frac{n}{m_i})$.

Extracting the LWAM for edges from u_i and nodes below u_i in $L(x)$, for each $u_i \in L(x)$ takes constant time after having processed u_i. Extracting the LWAM for edges from v_j and nodes below v_j in $R(x)$, for each $v_j \in R(x)$ is done through a single traversal of $\mathcal{T}$ in $O(|R(x)|)$ time. This means that the total time for computing the largest weight agreement matchings is $O(\sum_{i=1}^{p} m_i log\frac{n}{m_i}) \leq O(nlogn)$.

5.9 Creating the Maximum Agreement Subtree

The last step of the algorithm is to construct the maximum agreement subtree from the LWAMs that have been computed.

As mentioned earlier, the Largest Weight Agreement Matching constructed from a matching graph $G(x)$ corresponds to the Maximum Agreement Subtree between T_1 and $T_2(x)$. We will show how the MAST $\mathcal{A}$ can be constructed given the LWAM with white edges $we_1, we_2, ...$ and proper crossing with edges ge, re or single edge ge or re (See figure 5.6).

Recall that a white edge (u_i, v_j) corresponds to the subtree $MAST(M_i, N_j)$, a green edge corresponds to $MAST(M_i, T_2(v_j))$ and a red edge corresponds to $MAST(T_1(u_i), N_j)$. These subtrees will appear in $\mathcal{A}$ in the same order that the edges appear in the matching. $\mathcal{A}$ will have an internal node for each white edge which forms a path $p_1, p_2, ...$ through $\mathcal{A}$. These nodes will be in the same order that the white edges appear in the matching. The off-path child of node p_i will then be the root of the subtree corresponding to edge we_i. The second child c of the last node in the path will hold the subtree(s) corresponding to the edge(s) in the proper crossing. If the proper crossing is a single edge, c will be the root of the subtree corresponding to that edge. Otherwise c will have two children, one being the root of the subtree corresponding to ge and one being the root of the subtree corresponding to re.

What we need now is to find the subtrees corresponding to the edges in the matching. All the LWAMs corresponding to these subtrees have already been computed. Each is found in constant time by the same procedure as the edge weights were determined. The subtrees can now be constructed recursively.

5.9.1 Challenges

Our first approach to construct the MAST for the two input trees, was to construct and save the MAST for each of the LWAMs created in the algorithm. When constructing the MAST for a LWAM, the subtrees corresponding to each edge had already been created and could therefore be looked up in constant time. However, this gave rise to a runtime complexity of $O(n^2)$. Consider a tree where each internal node has a leaf as one of its children. If the input trees are of this structure and have size n, the algorithm would create n LWAMs, one for each $u_i \in \pi$, with sizes $1, ..., n$ each of which should be used to construct a MAST. Constructing the MAST from a LWAM takes time linear w.r.t. to the size of the LWAM giving a total runtime of $O(n^2)$.

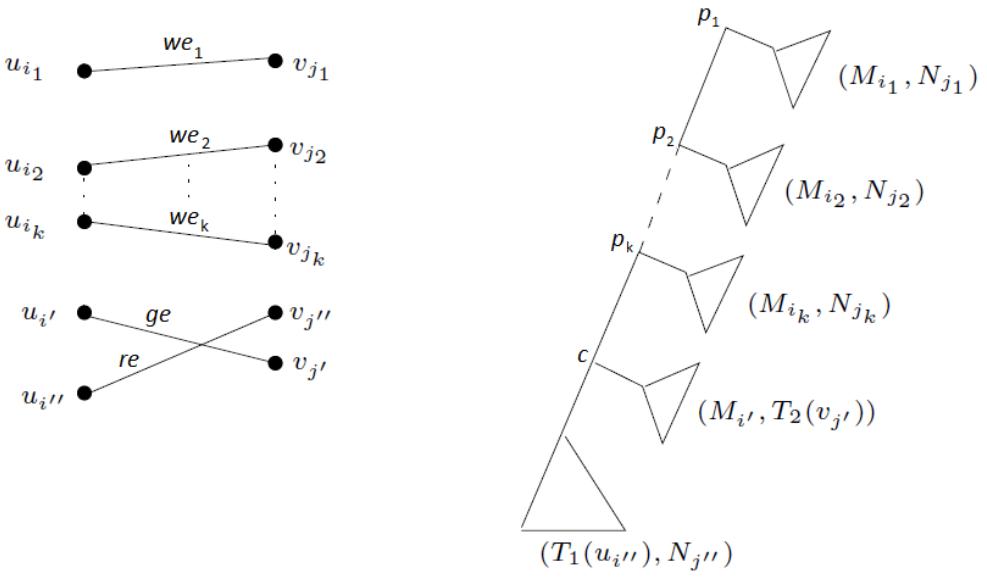

Figure 5.6: An Agreement Matching with the Associated Agreement Subtree. [1]

Instead, we chose to only create and store the LWAMs in the algorithm and construct the final MAST recursively after having created every LWAM.

5.9.2 Time Complexity

Creating the MAST is done by iterating through all LWAMs that correspond to each part of the MAST. For each white edge and each proper crossing, at least one node is added to the resulting tree, so the time taken to create the MAST is linear w.r.t. the number of nodes in the tree. The MAST can't be larger than the input trees, so the runtime is $O(n)$.

5.10 The Base Case

The base case for finding the largest weight agreement matching of T_1 and T_2 is when each tree consists of a single leaf. Then the LWAM is simply a green edge between the leaf of T_1 and the leaf of T_2 with weight 1. The LWAM could also be a single red edge. Either way, it will be translated to the MAST consisting of a single leaf, corresponding to the leaf that both trees hold.

5.11 Time Complexity

Throughout this chapter, we explained the runtime of each step of the algorithm, aside from the step of making recursive calls. Since each step satisfies the time complexity of $O(nlogn)$ and there is a constant number of steps, we can show that the total runtime of the algorithm is $O(nlogn)$.

Creating Centroid Decompositions	O(n)
Inducing Subtrees	O(n)
Constructing LWAMs recursively	O(nlogn)
Creating Matching Graphs	O(nlogn)
Computing LWAMs	O(nlogn)
Creating the MAST	O(n)
Total	O(nlogn)

Table 5.1: Runtime

For the two input trees T_1 and T_2, we recursively create LWAMs for each pair (M_i, S_i) of size $m_i, 1 \leq i \leq p-1$. We will inductively assume that each recursive call takes $O(m_i log m_i)$ time. Then, since $\sum_{i=1}^{p} m_i = n$, the runtime of all the recursive calls is $O(\sum_{i=1}^{p-1} m_i log m_i) \leq O(\sum_{i=1}^{p-1} m_i log n) = O((\sum_{i=1}^{p-1} m_i) log n) = O(nlogn)$. Since each other step of the algorithm is limited by $O(nlogn)$, the total runtime of the algorithm is $O(nlogn)$.

Table 5.1 gives an overview.

5.12 Space Consumption

As we will later explain from our experiments, it turns out that the amount of space used by the algorithm is an important factor for the runtime when running the algorithm in practice. We will therefore in this section try to get an overview of how much space is needed by the algorithm.

We will walk through each step of the algorithm and give an analysis of the amount of space that is needed.

5.12.1 Centroid Decompositions

When creating centroid decompositions, each node of T_1 and T_2 will be part of at most one centroid path which is all that needs to be stored. The amount of space needed is therefore $O(n)$.

5.12.2 Induced Subtrees

Preprocessing

Before inducing subtrees, T_2 needs to be preprocessed for finding LCAs in $O(1)$ time. This requires storing a constant amount of data at each node of T_2 and creating a new tree of size $O(n)$ where a constant amount of data is also stored at each node.

Preprocessing T_2 for inducing subtrees in linear time is also done by storing a constant amount of data at each node of T_2, giving a space consumption of $O(n)$.

Inducing Subtrees

Inducing a subtree of T_2 from a set of leaves L only requires storing data for each node of the subtree which will have size $O(|L|)$.

For each side tree $M_i, 1 \leq i \leq p-1$ of π, the subtree S_i is induced from the leaves of T_2 that are twins of the leaves of M_i. The total amount of leaves used to induce subtrees is therefore $\sum_{i=1}^{p-1} m_i < n$ and the space needed is $O(n)$.

5.12.3 Matching Graphs

Recall that a matching graph $G(x), x \in X$ consists of two sets of nodes $L(x)$ and $R(x)$ with edges between them. Only a constant amount of data needs to be stored at each node and each edge. As mentioned earlier, the total number of edges in all matching graphs has an upper bound of $O(nlogn)$ and since $|L(x)| \leq n$ and $|R(x)| \leq n$, the total amount of space needed for the matching graphs must be $O(nlogn)$.

5.12.4 Agreement Matchings

The Weighted Search Tree

For each matching graph $G(x), x \in X$ a binary search tree is created. The search tree gets a leaf for each node in $R(x)$, so the amount of space needed for a single search tree is $O(|R(x)|)$. For all search trees, that gives a space consumption of $O(\sum_{x \in X} |R(x)|) = O(n)$.

Largest Weight Agreement Matchings

Recall that for each matching graph $G(x), x \in X$, we compute and store the LWAM containing only edges from u_i and nodes below u_i in $L(x)$, for each $u_i \in L(x)$, and we compute and store the LWAM containing only edges from v_j and nodes below v_j in $R(x)$, for each $v_j \in R(x)$. The LWAMs are constructed while processing the edges of $G(x)$ and the edges are stored in the search tree. When updating a node in the search tree, only a constant amount of information is added. Either an edge or a proper crossing is updated, or an agreement matching is updated which is either a proper crossing or a white edge and a pointer to an agreement matching. So the amount of space needed to store the LWAMs is proportional to the number of times a node is updated.

In section 5.8.4 we concluded that the total amount of nodes visited when searching for leaves in the search trees, for any node $u_i \in \pi$, is $O(m_i log\frac{n}{m_i})$ and these nodes are visited only a constant number of times. So the amount of space needed for LWAMs must be $O(\sum_{i=1}^{p} m_i log\frac{n}{m_i}) \leq O(nlogn)$.

5.12.5 Creating the MAST

Creating the final MAST requires no more space than the size of the tree which can't be larger than n, so the amount of space used is $O(n)$.

Creating Centroid Decompositions	O(n)
Inducing Subtrees	O(n)
Constructing LWAMs recursively	O(nlogn)
Creating Matching Graphs	O(nlogn)
Computing LWAMs	O(nlogn)
Creating the MAST	O(n)
Total	O(nlogn)

Table 5.2: Runtime

5.12.6 Total

We will show that the total amount of space needed for storing the LWAMs that are recursively constructed is $O(nlogn)$. We can inductively assume that each recursive call $computeLWAM(M_i, S_i)$ uses $O(m_i log m_i)$ space, $1 \leq i \leq p-1$. Creating centroid decompositions, inducing subtrees and creating the final MAST uses $O(n)$ space and creating matching graphs and LWAMs uses $O(nlogn)$ space, so the total amount of space used by the algorithm is $O(n) + O(nlogn) + O(\sum_{i=1}^{p-1} m_i log m_i) = O(nlogn)$.

Table 5.2 gives an overview.

5.13 Linear Time Search Tree Construction Proof

Let us start by reminding ourselves that the function F(N) for the weighted search tree construction satisfies the following inequality, where A and B are constants and N and j are natural numbers.

$$F(N) \leq Max\{A + Blog(j) + F(j-1) + F(N-j); 1 \leq j \leq (N+1)/2)\} \quad (5.1)$$

Initially it is not obvious that F(N) grows at most linearly with N. As such, we want to prove that any function restricted by inquality (5.1) is at most linear in complexity. For the sake of generality, let F(N) denote an undefined function that fulfils (5.1). We will start by investigating the function F(N) by defining a secondary function G(N) that is defined over the constants of F: A, B and F(0).

$$G(N) = \begin{cases} F(0) & \text{if N=0} \\ Max\{A + Blog(j) + F(J-1) + F(N-J; 1 \leq j \leq (N+1)/2)\} & \text{if N>0} \end{cases}$$

From this definition it follows that $G(1) = A + 2G(0)$, and $G(2) = A + G(0) + G(1)$. In this manner we can investigate the first few terms of G.

$$
\begin{aligned}
G(0) &= F(0) \\
G(1) &= A + 2F(0) \\
G(2) &= 2A + 3F(0) \\
G(3) &= max\{3A + 4F(0), 3A + Blog(2) + 4F(0)\} \\
&= 3A + Blog(2) + 4F(0) \\
G(4) &= max\{4A + 5F(0), 4A + Blog(2) + 5F(0)\} \\
&= 4A + Blog(2) + 5F(0)
\end{aligned}
$$

From these first few terms it quickly becomes apparent that the coefficient of A is N, and the coefficient of F(0) is N+1. However, the coefficient of B is not clear. To determine the growth of B, quite a few more terms are needed to gain a clear picture. We will not include all of those here, but simply state that some investigative work revealed the coefficient of B to be $Nlog(2) - log(N+1)$. It is not immediately apparent that this coefficient is in fact at most linear in complexity. However, the following limit clearly shows that it is indeed the case.

$$\lim_{N \to \infty} \frac{Nlog(2) - log(N+1)}{N} = log(2)$$

This means that if the claims we made about the coefficients of A, B and F(0) are true, then the following inequality will show that F(N) can at most grow linearly as a function of N.

$$F(N) \leq AN + F(0)(N+1) + B(Nlog(2) - log(N+1)) \tag{5.2}$$

Let us now prove that this inequality indeed holds.

5.13.1 Proof

We prove inequality (5.2) by the use of strong induction.

Base case

Our base case is defined for the lowest value, N=0. We simply insert this value into the inequality.

$$
\begin{aligned}
F(0) &\leq A * 0 + F(0)(0+1) + B(0 * log(2) - log(0+1)) && \Rightarrow \\
F(0) &\leq 0 + F(0) + B(0-0) && \Rightarrow \\
F(0) &\leq F(0)
\end{aligned}
$$

Clearly, the inequality holds in the base case.

Inductive case

We are now given a natural number N, and assume in the spirit of strong induction our Induction Hypothesis to be

$$\forall x < N,\ F(x) \leq Ax + F(0)(x+1) + B(xlog(2) - log(x+1)) \tag{5.3}$$

and it that to prove that

$$F(N) \leq AN + F(0)(N+1) + B(Nlog(2) - log(N+1)) \tag{5.4}$$

Let J be the value of j that maximizes $A + Blog(j) + F(j-1) + F(N-j)$ in inequality (5.1). From the Induction Hypothesis (5.3), we now have for $x = J-1$ and $x = N-J$ the following two inequalities.

$$F(J-1) \leq A(J-1)+F(0)((J-1)+1)+B((J-1)log(2)-log((J-1)+1)) \tag{5.5}$$

$$F(N-J) \leq A(N-J)+F(0)((N-J)+1)+B((N-J)log(2)-log((N-J)+1)) \tag{5.6}$$

We can now insert $F(J-1)$ and $F(N-J)$ into (5.4), creating an inequality to verify.

$$\begin{aligned} F(N) \;\leq\; & A + Blog(J) + F(J-1) + F(N-J) \leq \\ & AN + F(0)(N+1) + B(Nlog(2) - log(N+1) \end{aligned}$$

We expand $F(J-1)$ and $F(N-J)$ according to (5.5) and (5.6), and simplify the resulting inequality.

$$\begin{aligned} & A + Blog(J) + A(J-1) + F(0)((J-1)+1)+ \\ & B((J-1)log(2) - log((J-1)+1)) + A(N-J) + F(0)((N-J)+1) \\ & + B((N-J)log(2) - log((N-J)+1)) \qquad \leq \\ & AN + F(0)(N+1) + B(Nlog(2) - log(N+1)) \end{aligned}$$

$\Rightarrow$ (Add up coefficients of A,B and F(0))

$$\begin{aligned} & AN + B((N-1)log(2) - log(N-J+1)) + F(0)(N+1) \qquad \leq \\ & AN + F(0)(N+1) + B(Nlog(2) - log(N+1)) \end{aligned}$$

$\Rightarrow$ (Subtract AN and F(0)(N+1))

$$\begin{aligned} & B((N-1)log(2) - log(N-J+1)) \qquad \leq \\ & B(Nlog(2) - log(N+1)) \end{aligned}$$

$\Rightarrow$ (Divide with B, and subtract (N-1)log(2))

$$\begin{aligned} & -log(N-J+1) \qquad \leq \\ & log(2) - log(N+1) \end{aligned}$$

$\Rightarrow$ (Add log(N+1) and apply the log rule of division)

$$log(\frac{N+1}{N-J+1}) \;\leq\; log(2)$$

We know from (5.1) that $1 \leq J \leq \frac{N+1}{2}$, meaning that $J = \frac{N+1}{2}$ maximizes $log(\frac{N+1}{N-J+1})$. If the inequality holds for the maximum J, then it will hold for the entire interval:

$$log(\frac{N+1}{N-\frac{N+1}{2}+1}) = log(2) \ \leq \ log(2); \quad N \geq 0$$

As such, the inductive case in (5.4) is proven to hold, which completes the proof. □

5.14 An Alternative Base Case

An alternative base case for the algorithm could be when the two input trees have a structure such that each internal node has a leaf as one of its children, i.e. the internal nodes of each tree form a single path through its tree. In this case the maximum agreement subtree can be computed by using a modified version of the $O(nlogn)$ solution to the Longest Increasing Subsequence (LIS) Problem.

Even though this procedure works, it can't be part of the final algorithm since it cannot compute all the LWAMs that are needed for the rest of the algorithm to work in $O(nlogn)$ time. However, we chose to implement it anyway in order to find out if it would be faster for these kind of tree inputs.

We will first describe the $O(nlogn)$ solution the the LIS problem and afterwards describe how we modified it to help us find the MAST of T_1 and T_2.

5.14.1 Solving the Longest Increasing Subsequence Problem

Given a list L of distinct natural numbers, we want to compute the longest increasing subsequence of numbers from that list. Fredman [2] explains how this can be done in $O(|L|log|L|)$ time.

The idea of this solution is to compute a list I where the last index holds the smallest number that ends a longest increasing subsequence of L. By having each number point to the number which was previous to it in I when it was added (we will refer to this number as the parent), the LIS can be constructed by iterating through these pointers starting from the last number in I.

The procedure is as follows:

- Initialize list I and add the first number of L.
- For every other number x in L do the following:

  If x is greater than the last number of I, add x to the end of I.

  Otherwise, find the smallest number in I which is greater than x and replace it with x. This number can be found in $O(log|L|)$ time using binary search.
- Construct the list LIS.

The last number of LIS is the last number x of I.

The second last number is the parent x' of x.

The third last number is the parent of x'.

...

5.14.2 The Modified Longest Increasing Subsequence Problem

Having the two trees T_1 and T_2, we can assign numbers in increasing order to the leaves of T_1 top to bottom, where the ordering of the bottommost two leaves is arbitrary, and number the leaves of T_2 such that twins have the same number. This can be done in linear time by having each leaf of T_1 point to its twin. By iterating through the leaves of T_2 top to bottom, we get a list L of possibly non-increasing numbers. Since all internal nodes except from the bottommost have exactly one leaf as a child and the numbers of the leaves in T_1 are increasing top to bottom, we can construct a MAST from the two trees by finding the longest increasing subsequence of L where the last number need only be greater than the third last number. Since the ordering of the bottommost two leaves of the result doesn't change the topology of the tree, the ordering of the last two numbers of the LIS shouldn't change the result either.

This gives rise to our modified version of the Longest Increasing Subsequence Problem (MLIS):

Given a list L of integers, we want to compute the longest increasing subsequence of integers from that list, but where the last integer only needs to be greater than the third last.

Our solution to this problem is simply a modification of the solution described in the previous section.

The list I is constructed similarly, but the last number of I might not be greater than the previous number. Besides having a parent, the last number x of I will also have a pointer to the number which preceded the parent when x was added to I (We will refer to this number as the grandparent of x).

Initially, the first two numbers of L is added to I. For every other number x in L there are 2 cases:

- Case 1: The last number y of I is greater than its parent $p(y)$.

  If $x > p(y)$, then x is added at the end of I.

  Otherwise, find the smallest number in I which is greater than x, using binary search, and replace it with x.

- Case 2: The last number y of I is smaller than its parent.

  If x is greater than one of the two last numbers of I, y and y', then x is put at the last index and the smallest of y and y' at the second last.

  If $y < y'$, then the parent of y is updated so that it now points to its grandparent.

If x is smaller than both of the two last numbers of I, find the smallest number in I which is greater than x, using binary search, and replace it with x.

As in the LIS solution, the result is constructed by iterating through the parent pointers starting from the last number in I.

Creating the MAST from the MLIS, is simply done by iterating through the MLIS while adding the leaves with those numbers to the result tree. The first child of the root should be the leaf with the number that comes first in the MLIS, the second child should be an internal node whose first child is the leaf with the number that comes second in the MLIS, and so forth.

Chapter 6

Testing Correctness

Having implemented the algorithms for the Maximum Agreement Subtree Problem, we wanted to test the correctness of each of them. Since the naive algorithm simply tries all possible combinations of leaves to prune from the two input trees and picks the one giving the largest agreement subtree, we believe it to be correct though very inefficient. By manually verifying results of the algorithm for small input trees, we convinced ourselves that the implementation did not contain any errors.

The $O(n^2)$ algorithm could now be tested against the naive algorithm by running the two algorithms on the same input trees and verifying that the result trees were of the same size. Because of the time complexity of the naive algorithm, we could only to this for small input trees. This was done for random input trees of sizes between 1 and 50. The $O(nlogn)$ algorithm was tested against the $O(n^2)$ algorithm for the same type of input trees, but of sizes between 1 and 10000. The fact that the $O(n^2)$ algorithm had already been tested and that it is very unlikely that the two algorithms would fail on the same input giving the same false result, indicates that both algorithms are correct when passing the test.

Also the algorithm using the MLIS solution has been verified. This algorithm was tested against the $O(nlogn)$ algorithm.

Chapter 7

Experiments

7.1 Setup

All the experiments were performed on a Lenovo G50-80 laptop with the following specifications:

- Running Windows 10 x64
- CPU: Intel i5-5200U with 2 cores running at 2.20 GHz
- 4 GB RAM

The experiments were performed from IntelliJ with a maximum memory heap size of 2048 MB. In all experiments, the algorithm was run five times on each input and we plotted the median runtime in the graphs. By using the median value instead of the mean value, we avoid outliers affecting the graph. Such outliers could among other things be caused by the OS prioritizing other processes so ignoring these will give a more representative graph. Before each experiment, we ran the algorithm under test with random inputs 100 times in order to warm up the cache.

7.2 The $O(N^2)$ Algorithm

The algorithm by Goddard et. al. [4] produces an $n * n$ table no matter the topology of the input trees The runtime of the algorithm must therefore be $O(n^2)$ for any input and we don't expect any significant difference in runtime when inputting trees of different topologies. We verified this with an experiment where we measured the runtime of the algorithm when inputting trees of different topologies. Two of the graphs that we created is seen in figure 7.1 and 7.2, where the difference in runtime is minimal. These graphs should also verify the runtime of $O(n^2)$. However, it is not obvious that this is the case. Especially because of the huge increment in runtime for input trees of size greater than ~6000. At size ~6400, the JVM ran out of memory and the program terminated. A reason for the increment in runtime could be that the runtime is affected by the garbage collector working while the program is running. The

garbage collector used in the execution is the default garbage collector for the JVM. This garbage collector pauses the program execution when collecting and therefore directly affects its runtime.

7.2.1 Garbage Collecting

We created a test to measure how much time was used by the garbage collector for each run of the algorithm. We ran the algorithm for the exact same input trees and plotted the runtime of the garbage collector for each tree size. The result is the graph shown in figure 7.3. Here we see some of the same behaviour as for the runtime of the algorithm which indicates that the unexpected behaviour was caused by the garbage collector. Figure 7.4 shows a graph where the time used on garbage collecting is subtracted from the time used on running the algorithm. This graph looks more like what we expected of the algorithm. Having divided the runtime by n^2, we would expect the graph to stay below some constant. This seems to be the case for a constant of ~600 which suggests that the runtime of the algorithm is indeed $O(n^2)$.

7.3 The $O(nlogn)$ Algorithm

We know that the runtime of the algorithm by Cole et. al. [1] is $O(nlogn)$ for n sized trees. However, we would like to find out how the algorithm runs in practice. How is the average runtime for random looking trees? What kind of trees give the best runtime? What kind of trees give the worst?

In the following sections we will walk through some of the experiments we did in order to answer these questions.

7.3.1 Random Trees

In practice, an algorithm like this will most likely be used on many very different looking trees. We therefore made an experiment to test how the runtime of the algorithm would look for randomly generated trees.

We ran the algorithm on randomly created trees of sizes $100, 200, ..., 60000$.

The result of this experiment is seen in Figure 7.5. Since we expected the algorithm to have runtime $O(nlogn)$, we divided the runtime with $nlogn$ and the graph should stay below some constant. However, this is not exactly what we see in the graph. There are two things to notice. At size ~5000, ~30000 and ~40000 there is a sudden increase in runtime and from size ~40000 the graph seems to keep increasing. At size ~52000 the JVM ran out of memory which stopped the execution.

We suspected this behaviour to be caused by the garbage collector and therefore executed the same experiment again, but this time plotting the runtime of the algorithm after having subtracted the runtime of the garbage collector. The result is shown in figure 7.6 after having divided the runtime with $nlogn$. This graph indeed looks like it will stay below some constant. We therefore believe that the runtime of the algorithm has an upper bound of $O(nlogn)$.

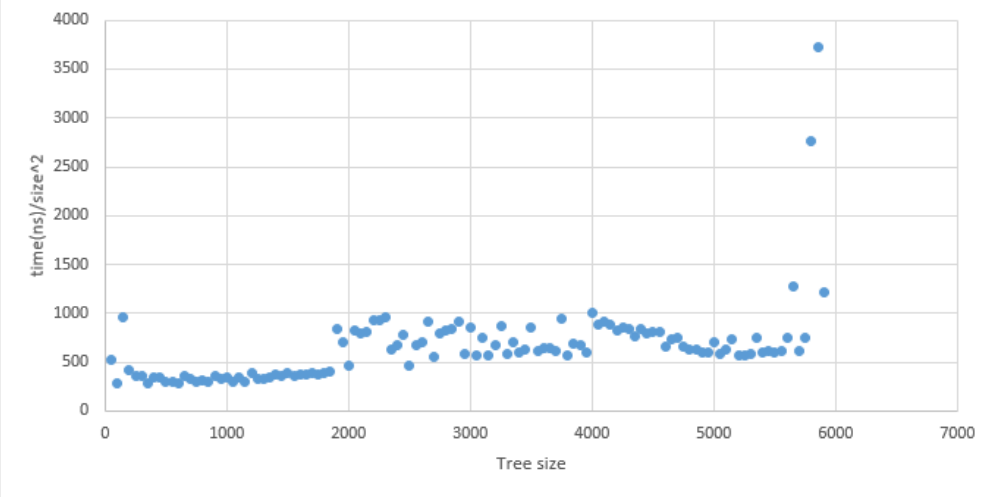

Figure 7.1: The runtime of the $O(n^2)$ algorithm given trees where each internal node has a leaf as one of its children.

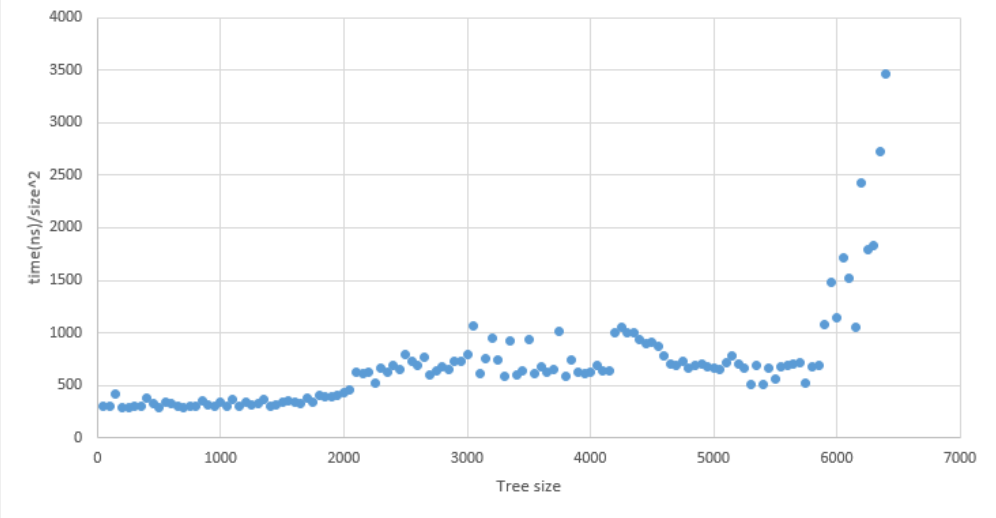

Figure 7.2: The runtime of the $O(n^2)$ algorithm given identical complete trees.

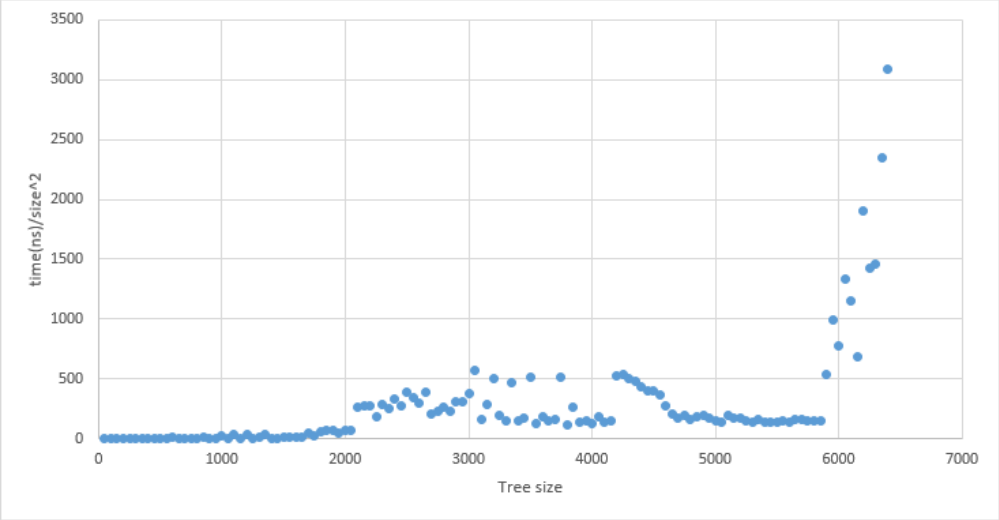

Figure 7.3: The runtime of the garbage collector when running the $O(n^2)$ algorithm given identical complete trees.

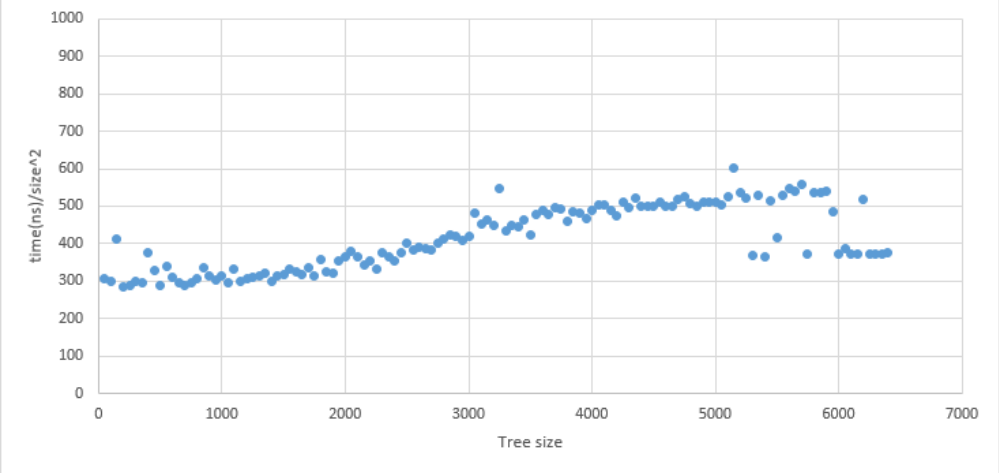

Figure 7.4: The runtime of the $O(n^2)$ algorithm given identical complete trees, after subtracting the runtime of the garbage collector.

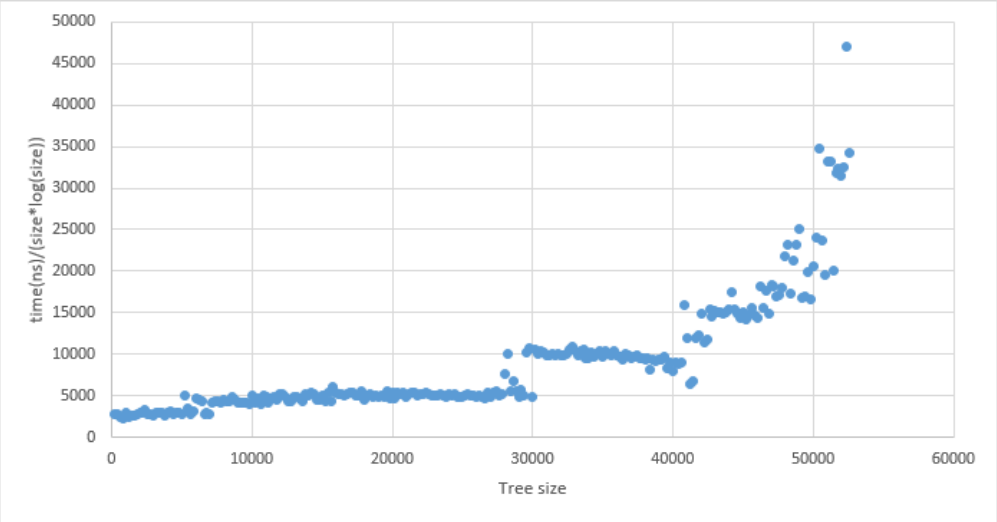

Figure 7.5: The runtime of the *nlogn* algorithm given random trees.

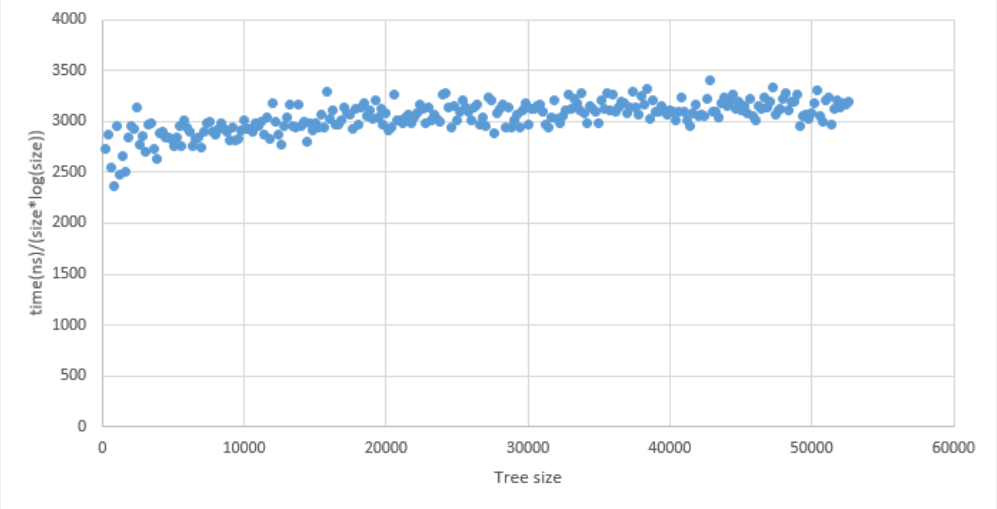

Figure 7.6: The runtime of the *nlogn* algorithm given random trees after subtracting the time used on garbage collecting.

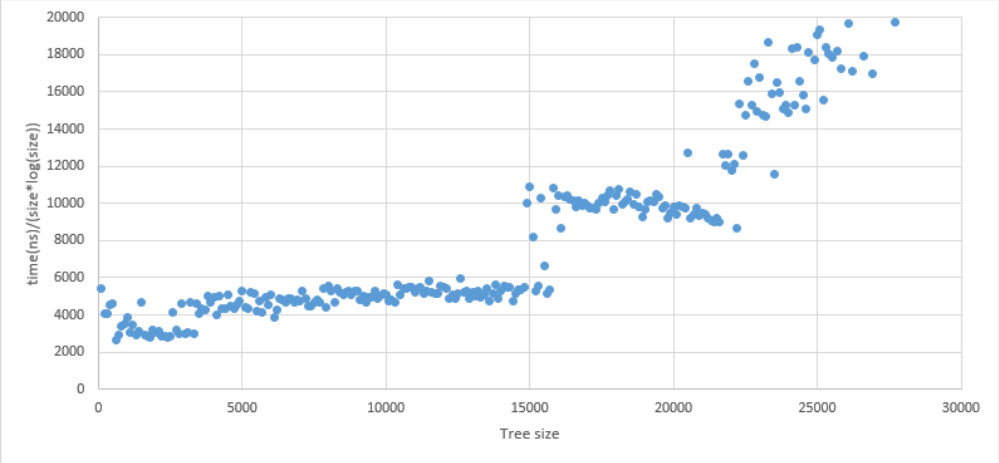

Figure 7.7: The runtime of the *nlogn* algorithm given random trees with 1024 MB allocated memory.

Behaviour of the Garbage Collector

We still want to be able to explain this behaviour of the garbage collector, e.g. why the amount of work needed by the garbage collector increases at those exact tree sizes. So we tried running the same experiment again, but with a different amount of allocated memory. In the previous tests, a memory heap of 2048 MB were allocated. In figure 7.7 we see the result of the same experiment, but with a memory heap of 1024 MB allocated. In this experiment, we see the same increases in runtime of the garbage collector, but they happen earlier than in the previous experiment. This indicates that the garbage collector used in these experiments will run depending on how much free memory is left. Using a computer with more RAM so that a larger memory heap can be used could therefore result in a better runtime for large input trees. The algorithm should also be able to work for input trees larger than 50000 when allocating more than 2048 MB of memory.

7.3.2 Best Case Trees

The part of the algorithm that requires most processing power is creating the matching graphs and processing them to compute largest weight agreement matchings. Matching graphs consists of edges and computing LWAMs is done by processing these edges. A tree topology that minimizes the runtime of the algorithm could therefore be a topology that minimizes the number of edges that needs to be created. Since the garbage collector has a large impact on the runtime in practice, it will also make sense to find a tree structure that minimizes the space consumption of the algorithm. The algorithm uses space on storing matching graphs, so minimizing the number of edges that needs to be created seems like a good choice.

Recall how the edges are determined. For each leaf in T_2, an edge will be created for each centroid path encountered from the leaf to the root of T_2.

For any two trees T_1 and T_2 that are given the algorithm, at least one edge will be created for each leaf of T_2 since all leaves will encounter the centroid path starting at the root of T_2. The number of edges created will therefore be minimal when T_2 has as few centroid paths as possible. Having a tree topology where each internal node has a leaf as one of its children, T_2 will have only one centroid path, so only one edge will be added for each leaf in T_2.

The process of creating edges is however done in every recursive call of the algorithm, so we want a structure of T_1 that gives the fewest possible recursive calls. Choosing the same tree structure as for T_2 means that each side tree of π will have size 1 so there will be no recursive calls. The result is that the total number of edges created by the algorithm is equal to the number of leaves in T_2, namely n. This actually means that the step of creating matching graphs in the algorithm only requires $O(n)$ space, instead of $O(nlogn)$.

When processing a matching graph $G(x), x \in X$ to compute LWAMs, each edge (u_i, v_j) will be processed where the leaf in the search tree corresponding to v_j is searched for in $O(log\frac{|T_2(x)|}{n_j})$ time for singleton edges. Having input trees of the topology just mentioned, means that there is only one graph $G(r)$, where r is the root of T_2. All edges are singleton edges and for each node v_j, $n_j = 1$. So even with this tree topology, the total runtime of the algorithm is $O(\sum_{v_j \in \pi(r)} log|T_2(r)|) = O(nlogn)$. We therefore don't expect an asymptotic improvement of the runtime compared to the result with random trees, but we do expect a noticeable constant improvement.

Another thing about the input trees that affects the runtime of the algorithm, is how similar the two trees are. The more they have in common, the larger the LWAMs will become, and processing these LWAMs will take more time. Since the similarity of the trees does not affect the amount of edges created for this tree structure, the algorithm will be fastest for trees that are not similar. Again we don't expect an asymptotic improvement compared to using identical trees, but a slight constant improvement.

We created an experiment for running the algorithm on trees of the structure explained earlier where the two trees were identical, and one where the leaves in the second tree appeared in the opposite order than that of the first tree.

The graphs in figure 7.8 and 7.9 shows the runtime of the algorithm, using the two kinds of inputs just explained. Comparing the two graphs, it looks like there is actually no significant difference in runtime, so it seems like the size of the LWAMs doesn't affect the runtime of the algorithm much for this kind of input trees. As expected, it looks very much like the algorithm is running in $O(nlogn)$ time, but it is still significantly faster than what we saw for random tree inputs.

Another thing to notice is that we don't see the sudden increases in runtime as we did when inputting random trees. This can be due to the fact that less space is used on the matching graph, but also that there are no recursive calls meaning that the program does not use up memory on side trees, induced subtrees etc. Therefore the garbage collector has less work to do.

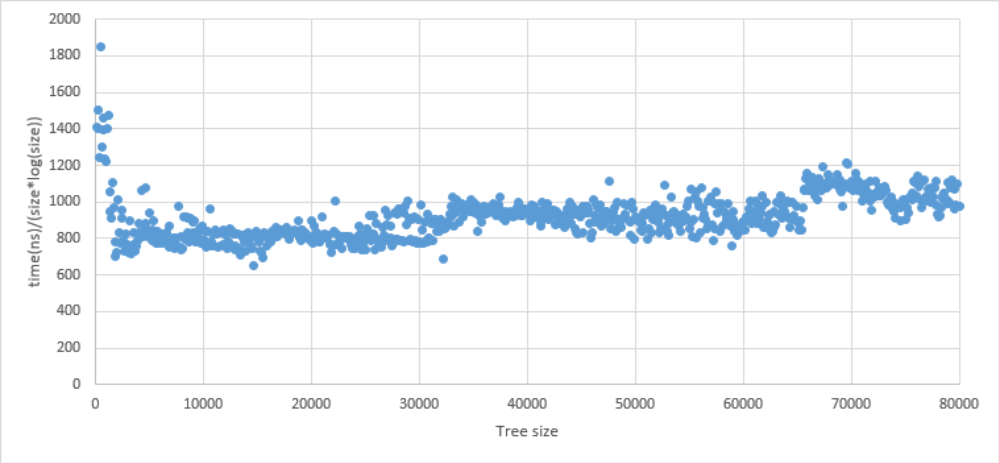

Figure 7.8: The runtime of the *nlogn* algorithm given two identical best case trees.

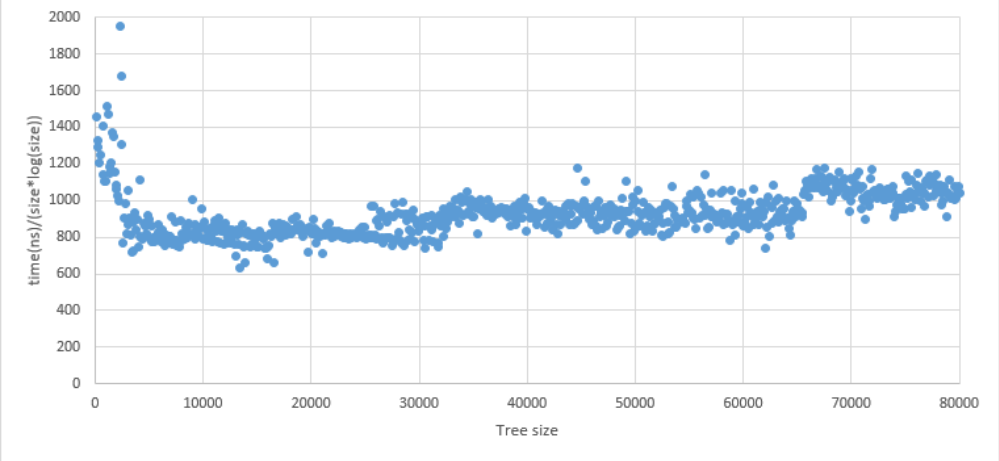

Figure 7.9: The runtime of the *nlogn* algorithm given two nonidentical best case trees.

7.3.3 Worst Case Trees

Since the runtime of the algorithm can be minimized by inputting trees that minimizes the number of edges created, we expect that the input trees that gives the worst runtime are trees that maximizes the number of edges that are created by the algorithm. This should also increase the space consumption.

Again, recall that for each leaf in T_2, an edge will be created for each centroid path encountered from the leaf to the root of T_2. Now consider the centroid path $\pi(r)$ with nodes $v_1, v_2, ..., v_q$ starting at the root node r, where $q \geq 4$. The leaf v_q will encounter only one centroid path on the path to r, namely $\pi(r)$ no matter the structure of T_2, so a single edge is created. The side tree N_{q-1} is a single leaf. Otherwise $\pi(r)$ would include the root of N_{q-1} instead of v_q. That leaf also encounters exactly one graph. The side tree N_{q-2} contains at most 2 leaves or $\pi(r)$ would have included the root of N_{q-2} instead of v_{q-1}. If N_{q-2} is a single leaf, it will encounter only one centroid path, but if N_{q-2} contains 2 leaves, both leaves will encounter 2 centroid paths. Continuing this way we can argue that in order to maximize the number of edges created in the algorithm, each side tree of the centroid paths of T_2 should hold as many leaves as possible. Creating T_2 with this in mind, results in a complete binary tree.

Now let's consider the structure of T_1. Since the process of creating edges is done in every recursive call of the algorithm, we want a structure of T_1 that results in as many recursive calls as possible with the largest possible trees. A recursive call is made for each side tree of π, so making these as large as possible will result in the most edges being created. This means that also T_1 should be a complete binary tree. Maximizing the amount of recursive calls and the size of the trees in the recursive calls also means that the amount of space needed by the algorithm for these trees is maximized.

To see the runtime of these kind of input trees, we created an experiment for running the algorithm with only complete binary trees as input. We explained earlier that if the input trees are similar, the LWAMs will be larger. However it also means that there will be fewer edges in the matching graphs, since the leaves of a side tree in T_1 will be split out over fewer side trees in T_2, so that might affect the runtime even more.

The graphs in figure 7.10 and 7.11 shows the result of the experiment when the input trees are identical and when the leaves are randomly distributed in the trees. From these graphs, we see that there is not a significant difference in runtime between the two. In both cases, it is easily seen that the runtime is worse than when inputting random trees, and again we see sudden increases in runtime when reaching certain tree sizes. Already at tree size ~42000, the JVM ran out of memory and could not continue the execution. By choosing this tree topology, we tried to maximize the space consumption of the algorithm and the garbage collector therefore needs to run more often and causes the increase in runtime. The graph in figure 7.12 shows the runtime of the algorithm after having subtracted the runtime of the garbage collector. As expected, the algorithm still seems to satisfy the runtime upper bound of $O(nlogn)$.

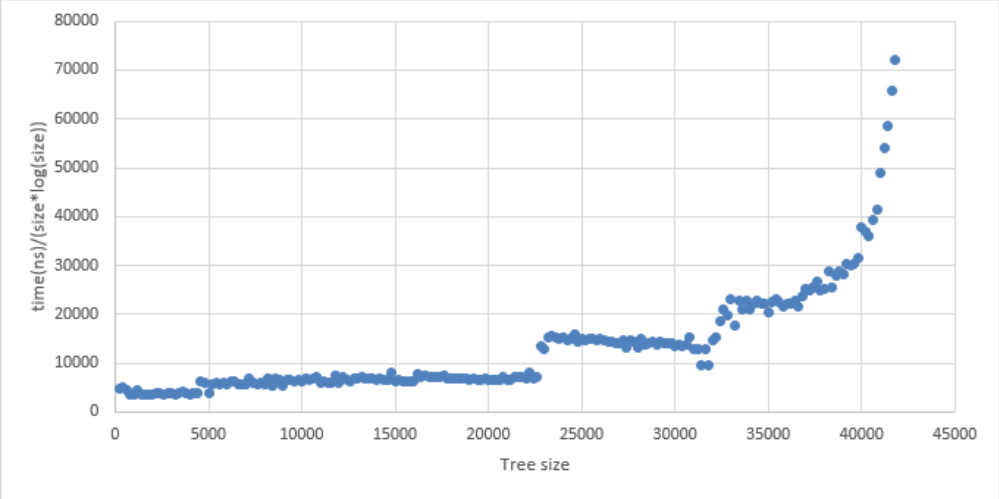

Figure 7.10: The runtime of the *nlogn* algorithm given two identical complete trees.

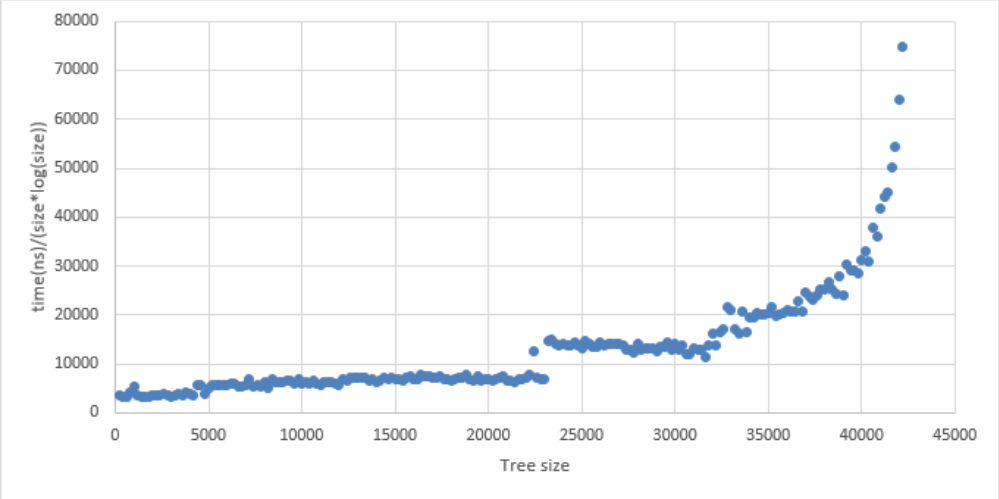

Figure 7.11: The runtime of the *nlogn* algorithm given two nonidentical complete trees.

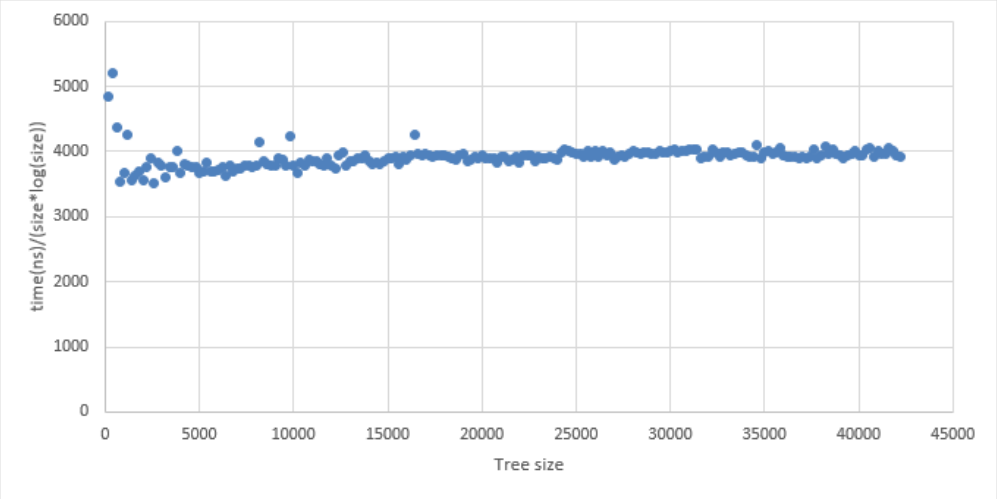

Figure 7.12: The runtime of the $O(nlogn)$ algorithm given two identical complete trees after having subtracted the runtime of the garbage collector.

7.4 Comparing Algorithms

In figure 7.13 we see the runtime of the two algorithms, given identical complete trees as input. Clearly, there is a huge improvement from the $O(n^2)$ algorithm to the $O(nlogn)$ algorithm and even for small input trees, the $O(nlogn)$ algorithm is dominant. In figure 7.14, we can see the difference in runtime for smaller input trees. For input trees of size smaller than ~60, the runtime of the two algorithms are very similar.

7.5 MLIS

The small algorithm that we described in section 5.14.2 only works for input trees of a special structure; the same tree structure that we evaluated would give the shortest runtime for the algorithm by Cole et. al. It is expected to run in $O(nlogn)$ time, but because of its simplicity, we expect it to be faster than the algorithm by Cole et. al. in practice. We tested this through an experiment where we ran the two algorithms with the exact same input trees and plotted the results in the graphs seen in figure 7.15

From the second graph, we conclude that both algorithms satisfy the $O(nlogn)$ runtime and from both graphs it is clear that the algorithm using MLIS has a better runtime.

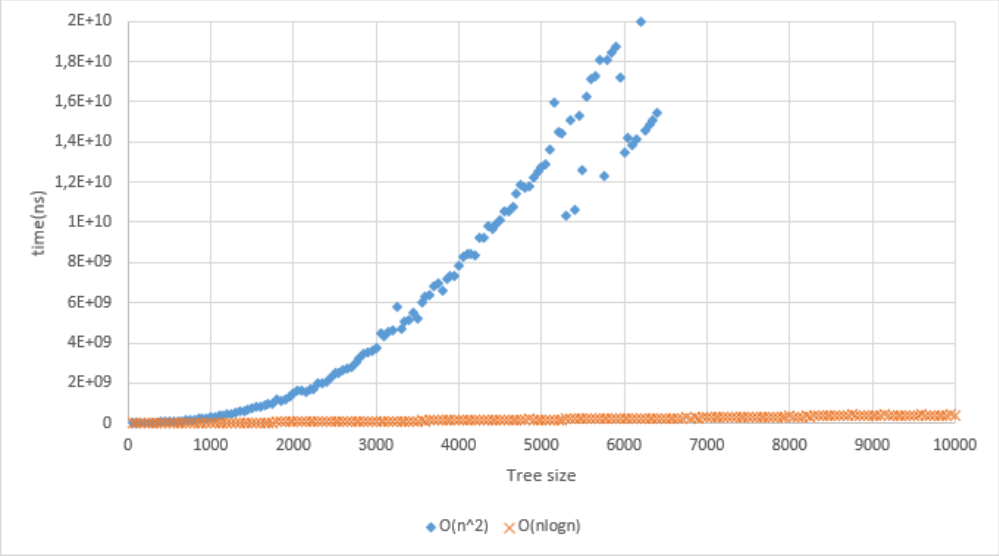

Figure 7.13: The runtime of the $O(n^2)$ algorithm and the $O(nlogn)$ algorithm given identical complete trees after having subtracted the runtime of the garbage collector.

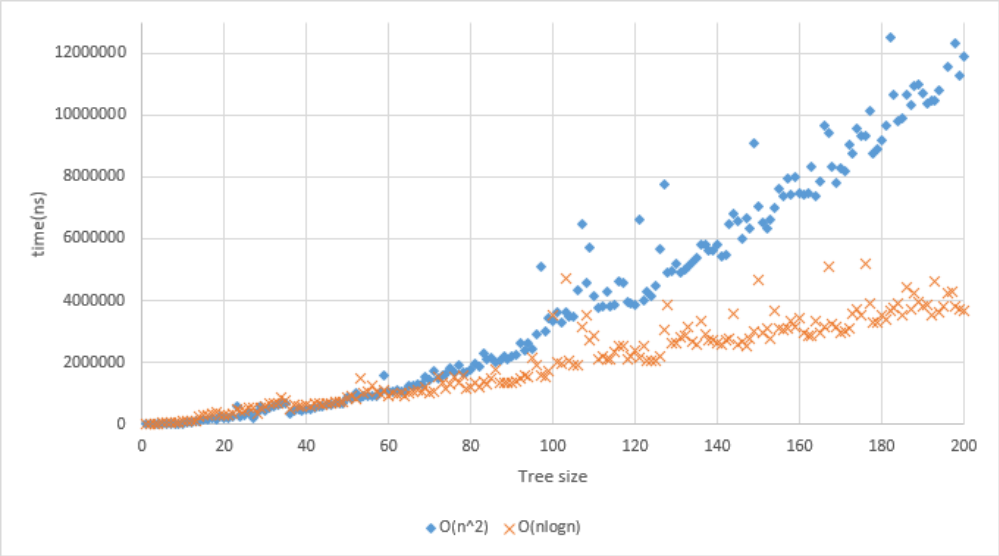

Figure 7.14: The runtime of the $O(n^2)$ algorithm and the $O(nlogn)$ algorithm given identical complete trees after having subtracted the runtime of the garbage collector.

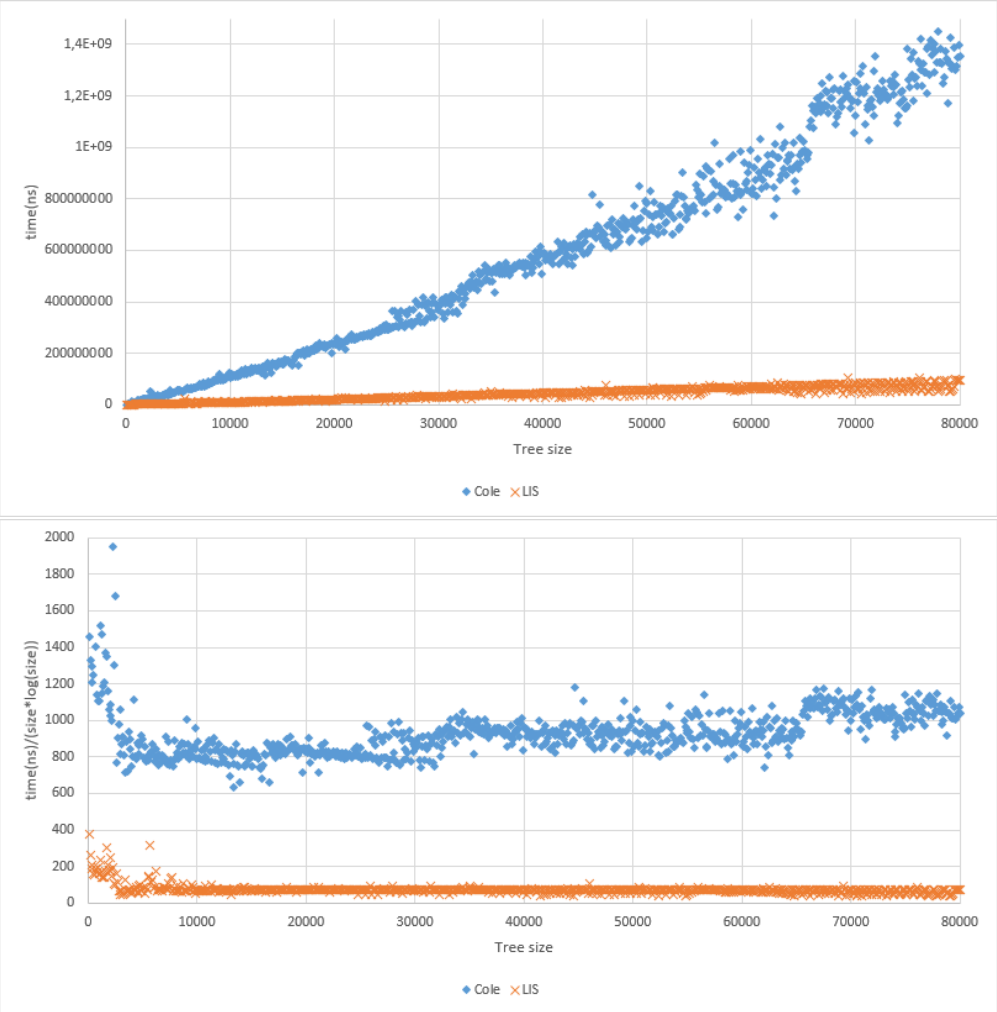

Figure 7.15: The runtime of the *nlogn* algorithm and the algorithm using MLIS given a special type of trees.

Chapter 8

Conclusion

The $O(n^2)$ algorithm turned out to be fairly simple to understand and implement, but has its limitations as a result. The algorithm is useful for smaller trees, but falls short otherwise. This is mainly because there seemingly is no way to guarantee that the entire size table won't be kept in memory during construction or backtracking of it. Our expectation was originally that this algorithm might outperform the $O(nlogn)$ algorithm in practice for realistically sized trees simply because of lesser algorithmic complexity and therefore possibly also overhead. However, that turned out to be false. The two algorithms seemed to perform reasonably similar for trees up to the size of ~60, from which point forward the complexity difference became apparent. The $O(nlogn)$ algorithm, which is currently the best theoretical solution to the problem in terms of complexity, took significant effort to implement. It requires some attention to detail when coding, since the complexity is quickly broken by small mistakes. As expected, it performs way better on large trees, and reasonable well on smaller trees also. Our small algorithm that uses a solution to the LIS problem, performs even better than the $O(nlogn)$ algorithm, but only works for trees where each internal node has a leaf as a child.

8.0.1 Future Work

Given more time, we would have liked to explore the following subjects further.

Distances: the MAST problem can quite easily be used as a measure of distance between two trees. One could just define the distance between two trees as the size difference between the input trees and the produced MAST. However, it could be interesting to see how such a distance measure might relate to other, faster, algorithms for tree distance.

Multiple Trees: The MAST problem is defined between two trees, but how, if possible, can one efficiently handle MAST calculation between multiple trees. Given a number of trees, a quick solution is to compute the MAST between the first two trees, prune the appropriate leaves from the next tree in line, and then run the algorithm on the MAST and this tree. This initially gives a complexity of $O(knlogn)$, where k denotes the number of trees.

Non-Binary Trees: While binary trees are most commonly used, it would be interesting to research how the $O(nlogn)$ algorithm might be expanded for trees of degree larger than two. The existence of an algorithm with complexity $O(n\sqrt{d}logn)$ for trees of degree d was conjectured in [1], and was later followed up and improved by [6].

Memory: The consumption, allocation and deallocation of memory is a major factor is the $O(nlogn)$ algorithm's running time. As such, we would like to investigate whether any data structures could improve the running time in practice, while the overall complexity stays the same.

Bibliography

[1] Richard Cole, Martin Farach-Colton, Ramesh Hariharan, Teresa Przytycka, and Mikkel Thorup. An o(nlogn) algorithm for the maximum agreement subtree problem for binary trees. *SIAM Journal on Computing*, 30(5):1385–1404, 2000.

[2] Michael L Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[3] Michael L. Fredman. Two applications of a probabilistic search technique: Sorting x+y and building balanced search trees. *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 240–244, May 1975.

[4] Wayne Goddard, Ewa Kubicka, Grezegorz Kubicki, and F. R. McMorris. The agreement metric for labeled binary trees. *Mathematical Biosciences*, 123(2):215–226, October 1994.

[5] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*, chapter 8. Cambridge University Press, May 1997.

[6] Ming Yang Kao, Tak Wah Lam, Wing Kin Sung, and Hing Fung Ting. An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *CoRR*, cs.CV/0101010, 2001.