Universiteit
Leiden
The Netherlands

Efficient conversion between vtrees

Diego van Egmond

Supervisors:
Dr. Alfons Laarman
Lieuwe Vinkhuijzen, MSc

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

30/07/2021

Abstract

Binary trees play a major role in many aspects of computer science. A lot of research has been done in conversion between binary trees. Vtrees are full binary trees with unlabeled internal nodes and labeled leaves. A vtree can be used in combination with a Boolean function to obtain a canonical Sentential Decision Diagram (SDD). An SDD is a representation of a Boolean function. It can be used for problems such as knowledge compilation, symbolic verification methods and constraint programming. These problems are memory intensive, and memory is a bottleneck. This problem can be solved by changing the vtree reduce the size of the corresponding SDD. In this thesis, we consider the problem of converting one vtree to another using only swaps and rotations. We propose two algorithms for this task, which we call Algorithm 1 and Algorithm 2. Algorithm 1 converts the vtree to a left-linear tree, reorders the leaves and finally changes the structure of the vtree to the desired structure. Algorithm 2 improves on this by recursively solving subtrees and using subtree reduction. Experiments show that Algorithm 2 gives an average decrease in path length of at least 12.5% compared to Algorithm 1, for vtree pairs with shared subtrees. We also find that these shared subtrees occur in at least 15% of vtree pairs for any tree size. We characterize all situations in which Algorithm 2 finds a shorter path than Algorithm 1.

Contents

1 Introduction

Binary trees play a major role in many aspects of computer science. Converting from one tree to another is a key part of many algorithms, such as when balancing binary trees. Examples of tree balancing can be found in the paper of Baer and Schwab [BS77], and the paper of Pushpa and Vinod [PV07].

Much research has been done on conversion between binary trees. Rotation distance, which is the minimal number of operations needed in a conversion, is discussed by Cleary and St. John in [CS09] and [CS10], and by Pallo in [Pal87]. Tarjan, Sleator and Thurston [DDST88] also show visualizations of binary trees, such as triangulations. Culik and Wood [CIW82] look at tree similarity measures.

A variable tree (vtree) is a full binary tree with unlabeled internal nodes and labeled leaves. The labels of the leaves are equal to the set of variables used in a corresponding Boolean formula, and each variable occurs in the vtree exactly once. Vtrees are used to create different Sentential Decision Diagrams (SDDs) for a single Boolean function. For every combination of a Boolean function and a vtree, there is one canonical SDD. Choi and Darwiche [CD13] define the SDD as "a representation of a Boolean function".

Choi and Darwiche [CD13] give examples of applications for the SDD, such as knowledge compilation [CKD13] in the field of Artificial Intelligence. It was shown by Vinkhuijzen and Laarman [VL20] that the SDD can also be useful for symbolic model checking, while it can be a significantly more compact structure than the common Binary Decision Diagram (BDD). Another application of the SDD is constraint programming, which is shown by Kisa, van den Broeck, Choi, and Darwiche [KVdBCD14].

Algorithms that make use of SDDs often have the problem that the amount of used memory grows exponentially. This means that memory is often a bottleneck in these algorithms. Vinkhuijzen and Laarman [VL20] show that the size of an SDD can be changed by modifying the vtree. This can be seen in Figure 2 and Figure 3, which show two SDDs of different sizes for the same Boolean function, as well as the corresponding vtrees. By reducing the memory needed by individual SDDs we can run an algorithm with equal performance and less memory usage, or we can increase the number of SDDs we use without the need for more memory. We can decrease the SDD size by gradually changing the corresponding vtree. The SDD minimization algorithm of De Jong [dJ21] has this bottleneck in the vtree conversion.

In this thesis, we consider the problem of converting one vtree to another. The allowed operations are rotations and swaps, which are explained in Section 3.1. Cleary and St. John [CS10] have an algorithm for converting an one unlabeled binary tree to another. This algorithm converts the initial tree to a right-linear tree, and converts this right-linear tree to the desired target tree. However, the order of the leaves is not changed. Since vtrees have labeled leaves, the order should be changed as well.

We propose two algorithms to convert one vtree to another. These algorithms therefore can be directly implemented in the algorithm of De Jong **??**. The algorithms are inspired by the algorithm of Cleary and St. John [CS10]. We call them Algorithm 1 and Algorithm 2. Algorithm 1 starts by

converting the initial vtree to the left-linear tvree. After reaching the left-linear vtree, we reorder the leaves using adjacent swaps. Finally, we convert the left-linear vtree to the target vtree. The path from the left-linear vtree to the target vtree is found by converting the target vtree to the left-linear tree and subsequently inverting this conversion path.

Algorithm 2 improves on Algorithm 1 by adding recursion and subtree reduction. We find a set of variables for which there exists a subtree containing exactly those variables in both the initial vtree and the target vtree. We call these subtrees a shared subtree. Algorithm 2 recursively finds such shared subtrees, converts the shared subtree we find in the initial vtree to the shared subtree we find in the target vtree, and finally reduces this shared subtree to a single leaf. This reduction means we don't change the subtree anymore, and we treat it as if it is a single leaf.

The first version of the algorithm and the recursive variant are implemented in `C`, making use of the SDD package of Choi and Darwiche [CD18b]. To transform the initial vtree to a left-linear vtree, we use the method in Algorithm 1. This method does not change the order of variables. Subsequently, we use the method in Algorithm 2 to put the variables in the desired order. Finally, we reach the target vtree by finding the path from the target vtree to the left-linear vtree, reversing this path and following it with the tree we want to transform. The recursive variant looks for subtrees which can be solved separately. When these subtrees are correct, they are treated as single nodes.

The result is that a transformation can be found between any pair of vtrees that share the same variables. The recursive variant of the algorithm can find a subtree that can be solved independently in at least 12.5% of randomly generated vtree pairs. The average length of the transformation path is decreased in these cases by at least 15%. Both these statistics are greater for smaller vtrees, and they are expected to be higher in real-world examples.

We also characterize all situations in which Algorithm 2 finds a shorter path than Algorithm 1. This is shown in Lemma 8.

The thesis is structured as follows: Section 2 discusses previous work that is relevant to this thesis. Section 3 discusses the heuristics we propose to make the algorithm of Cleary and St. John [CS10] applicable on vtrees. Section 4 discusses the experimental setup we use to evaluate these heuristics. Section 5 discusses the results of our experiments. Section 6 discusses the conclusion we draw from the results. Finally, Section 7 proposes future work that is relevant to this research.

2 Related Work

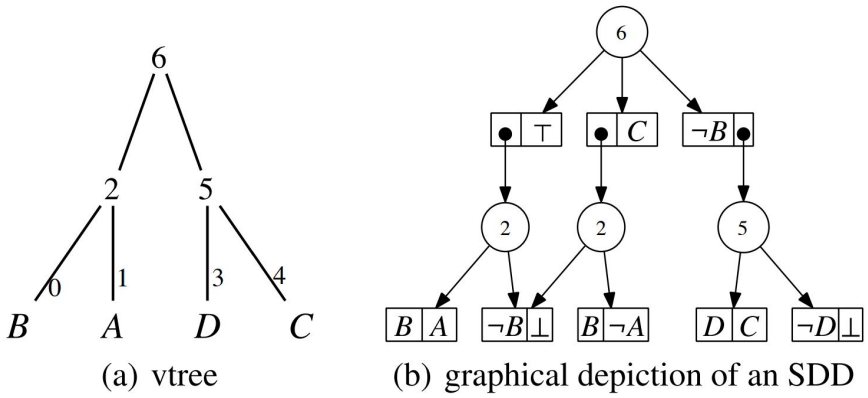
2.1 Vtrees and SDDs

An SDD is a canonical representation of a Boolean function given a so-called vtree. A vtree is a full binary tree with labeled leaves and unlabeled internal node. The labels of the leaves are equal to the set of variables used in the corresponding Boolean function. See Figure 1a for an example.

Figure 1b shows the SDD representing the function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. This SDD is created based on the vtree in Figure 1a. Each pair of squares represents an implication. The node containing B and A represents the implication $B \rightarrow A$. Similarly, the node containing $\neg B$ and $\bot$ represents the implication $\neg B \rightarrow \bot$. Each round node represents a logical conjunction of implications. The node containing 5 represents $(D \rightarrow C) \wedge (\neg D \rightarrow \bot)$. The number in the rounds node represents a subtree of the vtree. In the nodes below this round node, the variables in the left subtree of the corresponding vtree node are always on the left side of the implications, and the variables in the right subtree of the corresponding vtree node are always on the right side of the implications. For the round nodes containing 2, the variable B can only occur on the left side, and the variable A can only occur on the right side of the implications under these round nodes. The full SDD represents the following Boolean function:

$f(x) = (((B \rightarrow A) \wedge (\neg B \rightarrow \bot)) \rightarrow \top) \wedge (((\neg B \rightarrow \bot) \wedge (B \rightarrow \neg A)) \rightarrow C) \wedge (\neg B \rightarrow ((D \rightarrow C) \wedge (\neg D \rightarrow \bot)))$

This is equal to the function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.



(a) vtree (b) graphical depiction of an SDD

Figure 1: An SDD for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ and its corresponding vtree, taken from [CD13]

Vinkhuijzen and Laarman show that the size of an SDD depends on the chosen vtree. Figure 2 and Figure 3 show two SDDs which represent the same Boolean function, but which are constructed based on different vtrees. The balanced vtree in Figure 2 produces a slightly smaller SDD than the right-linear vtree in Figure 3.

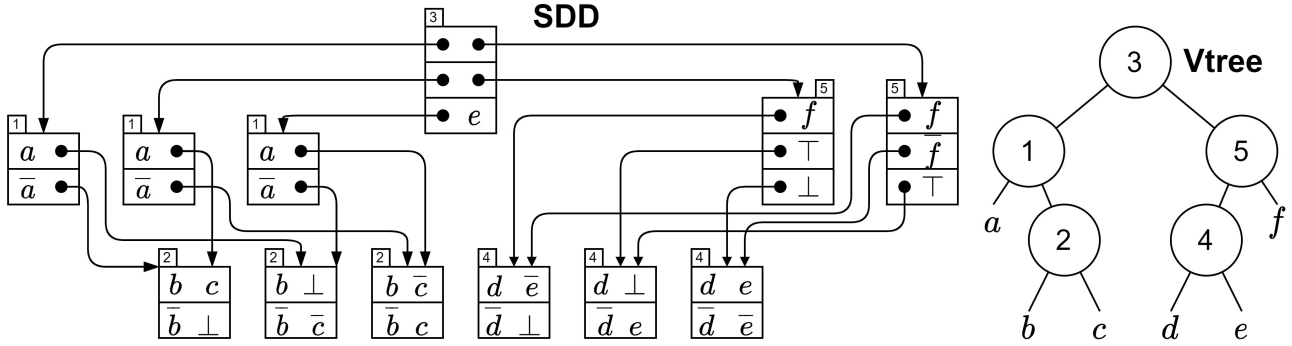


Figure 2: An SDD and the vtree it is normalised to. Rectangles are SDD nodes, labelled with the vtree node they are normalized to, taken from [VL20].

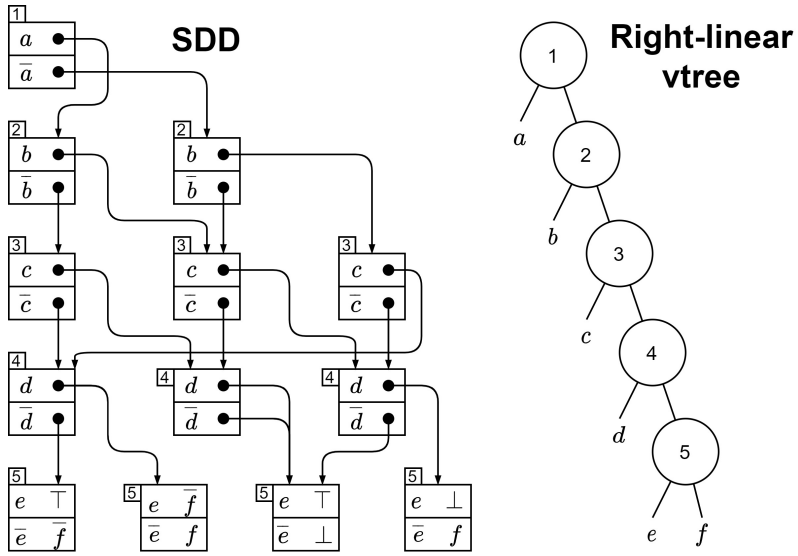


Figure 3: An SDD normalized to a right-linear vtree, taken from [VL20].

2.2 Swaps and rotations

Choi and Darwiche [CD13] define two operations on vtrees: rotation and swap. These operations can be applied to internal nodes of a vtree to change its structure. Figure 4 and Figure 5 are both from this paper.

2.2.1 Swaps

Applying a swap on a node means to swap its two children. This means that for this node, its left child and its right child are swapped. If one of these children is an internal node, the whole subtree is moved instead of just the single node. This subtree is the part of the tree, of which the root is the internal node that is being moved. A swap on node x can be seen in Figure 4.

2.2.2 Rotations

The rotation operation comes in two symmetric variants: the left rotation and the right rotation. The left rotation moves a node up in the tree, and moves the parent of this node down. The parent of the node that is moved up will become its left child. Its original right child will stay its right child, and its original left child will become the right child of the parent node that was moved down. To make sure this is possible, the node that is moved up in the left rotation needs to be a right child. Since the children of the node are moved during the operation, and we do not want our leaves to become internal nodes, we conclude that rotations can only be done on internal nodes.

The right rotation is the inverse function of the left rotation. However, it is not a mirror image of the left rotation. Since the right rotation on a node x is the inverse of a left rotation on node x, and x is moved up by the left rotation, this means x is moved down by the right rotation. The two variants of rotation can be seen in Figure 5. The operations `rr_vnode(`x`)` and `lr_vnode(`x`)` in Figure 5 are the right rotation and left rotation respectively.

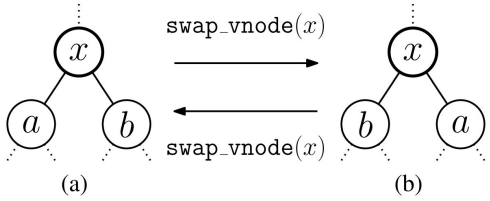


Figure 4: Swap on node x, taken from [CD13]

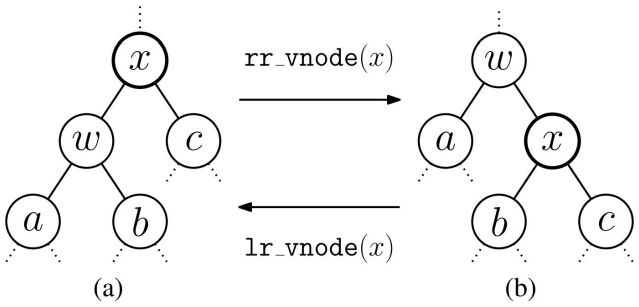


Figure 5: Rotation on node x, taken from [CD13]

2.3 Triangulations

A very useful property of binary trees, as explained by Sleator, Tarjan and Thurston [DDST88], is that it can be visually represented as a polygon that is partitionized into triangles. Such a partition of a polygon into triangles is called a triangulation. Figure 6 consists of four images.

The top left image shows a binary tree consisting of a root, the left subtree A of the root, and the right subtree B of the root. The top right image shows how this tree is represented as a triangulation. The root node is represented as an inverted triangle. The triangulation representing subtree A will be attached to the left side of this triangle, and the triangulation representing subtree B will be attached to the right side of this triangle. This will result in a triangulation with $|A| + 1$ sides on the left of the central triangle, and $|B| + 1$ sides on the right of the central triangle. $|A|$ and $|B|$ are the number of leaves in subtrees A and B respectively.

The bottom left image shows a detailed binary tree, and the bottom right image shows the corresponding triangulation. The root node is node 6, so the label of the central triangle is 6. Node 6 has a left child, labeled node 4. Therefore, the central triangle with label 6 has a triangle attached to its left side, which is labeled 4. Node 4 has a left child, labeled node 3, and a right child, labeled node 5. Therefore, in the triangulation, the triangle with the label 4 has a triangle labeled 3 attached to its left side, and a triangle labeled 5 attached to its right side. The same process takes place for node 3 and node 2, and the final result is the equivalent triangulation.

The reason this visualization is useful, is because each rotation in a tree matches an operation done in the corresponding polygon. This operation on the polygon is to remove one of the internal edges, and replace it by the opposite diagonal of the now empty quadrilateral. One of these diagonal swaps is shown in Figure 7. The important thing to notice, is that the outer edges can never be changed or moved. Because of this, the numbers near the corners of the polygon are never changed, which indicates that rotations can never change the order of any nodes or leaves. However, the swap operation can change the order of the nodes and leaves. Since we want an algorithm to change vtrees rather than unlabeled binary trees, we will need to use swaps to change the variable order.

2.4 Conversion to left-linear trees

A left-linear tree is a tree in which every internal node is a left child, with the exception of the root node. Figure 8 shows that any full binary tree can be converted to a right-linear tree using only right rotations. Similarly, any tree can be converted to a left-linear tree using only left rotations, since the right rotation is the inverse function of the left rotation.

2.5 The algorithm of Cleary and St. John

Since all rotations are reversible operations, and every tree can be converted to a left-linear and right-linear tree using only rotations, we can conclude that every structure can be reached from the left-linear tree and from the right-linear tree. This idea is what inspired Cleary and St. John [CS10] to develop a linear-time algorithm which can find a path between two unlabeled binary trees. This algorithm first converts the initial tree to a right-linear tree. To find the path from this right-linear

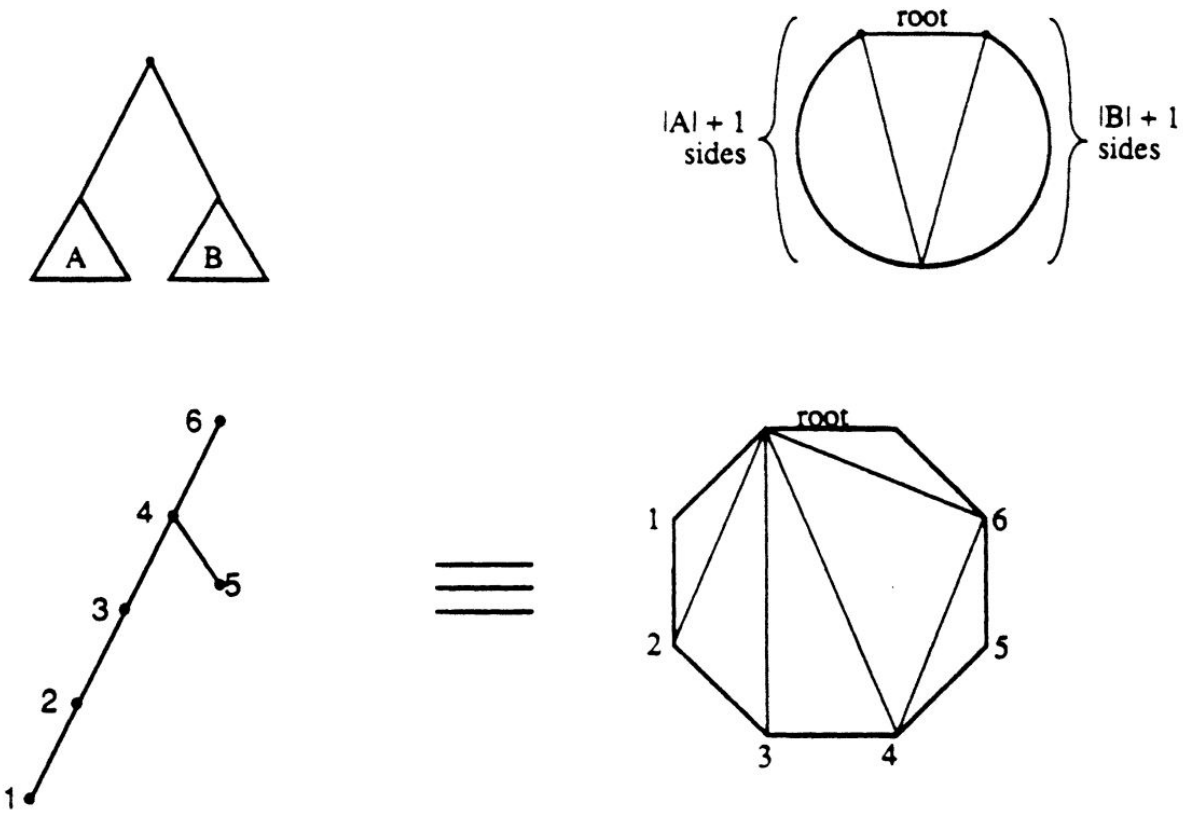


Figure 6: Binary trees and their triangulations, taken from [DDST88]

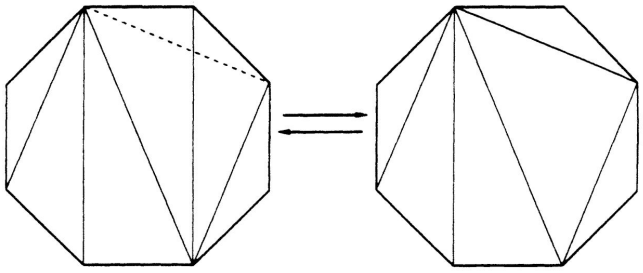


Figure 7: A diagonal swap, taken from [DDST88]

tree to the target tree, we instead search for a path from the target tree to the right-linear tree. We use the reversible property of tree rotations to reverse this path. This is now a path from the right-linear tree to the target tree. To get the full path from the initial tree to the target tree, we simply start by following the path from the initial tree to the right-linear tree, and subsequently the path from the right-linear tree to the target tree. Since vtrees are also full binary trees with unlabeled internal nodes, it is possible to use and expand on this idea. However, we need to add a function to put the variables in the right order.

2.6 Subtree reduction and chain reduction

Cleary and St. John [CS09] introduce the techniques of subtree reduction and chain reduction, which can be used to significantly reduce the size of unlabeled full binary trees, without changing the information it contains.

When one is looking for a path between two trees, and there exists a subtree that occurs in both trees, it is possible to collapse this subtree to a single leaf. An example of this is shown in Figure 9. Subsequently, it is easier to find a path between the two trees. Any path between the reduced trees that consists of only swaps and/or rotations is also a possible path between the original trees.

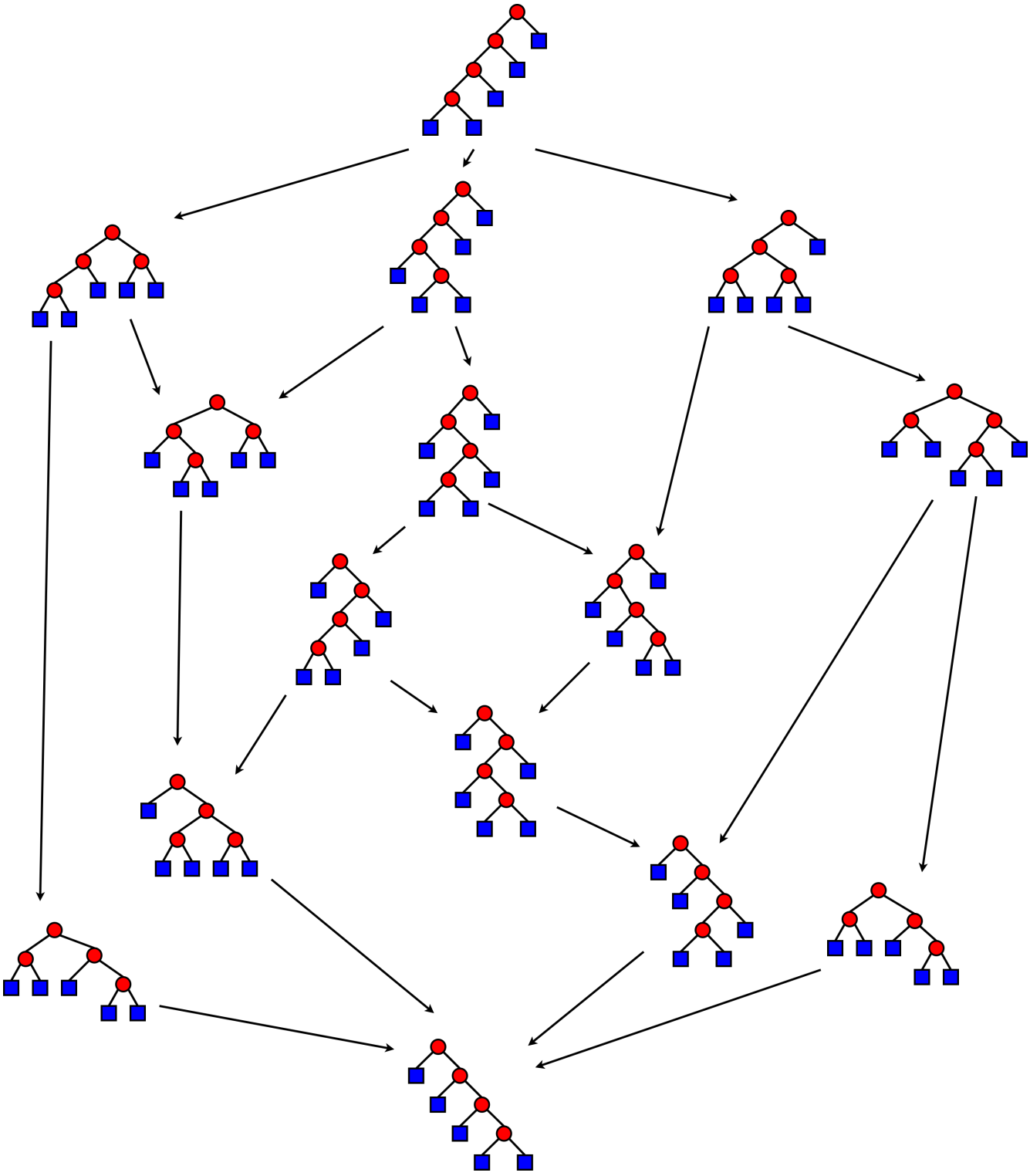


Figure 8: Tree transformation with only right rotations, showing all full binary trees with 5 leaves. For readability, some trees are duplicated.

Chain reduction, which is shown in Figure 10, is a way to represent an internal chain of nodes as an internal node. If this chain has a child on the left or right side, this child is attached to the new internal node. If there is not a child on one of the sided of the chain, a leaf is added to this side of the new internal node. In Figure 10, the chain only has a child on its right side, which means that the reduced form of the chain is an internal node with the original right child and an added left child. Any path between the reduced trees can also be applied to the original trees, if no swaps or rotations were done that would move the reduced chain.

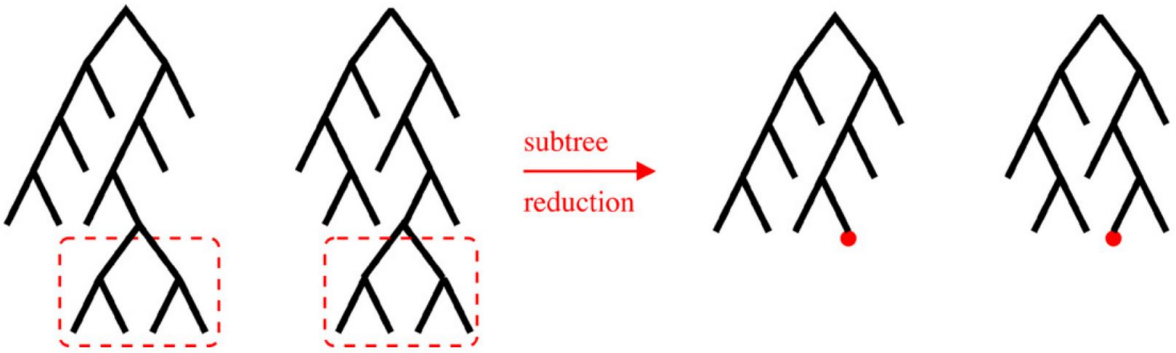


Figure 9: An example of subtree reduction, taken from [CS09]

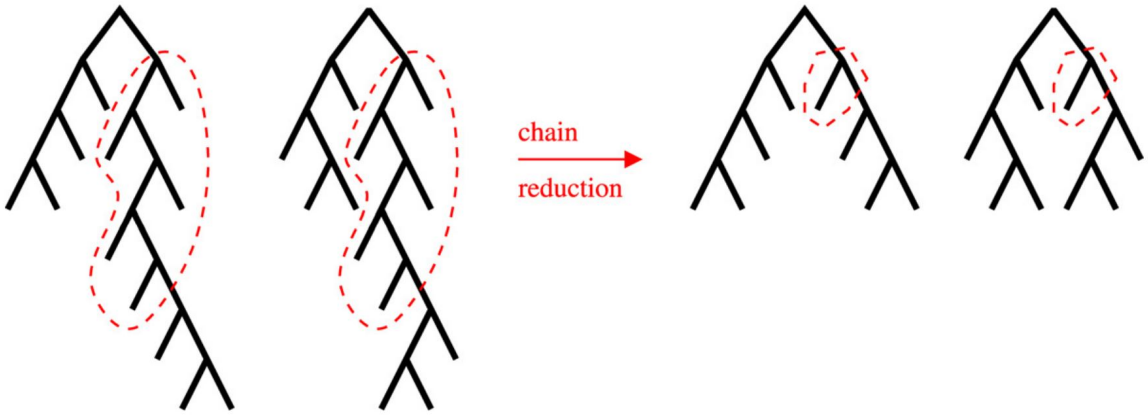


Figure 10: An example of chain reduction, taken from [CS09]

2.7 Internal changes

The paper "On the rotation distance in the lattice of binary trees" by J. Pallo [Pal87] includes an interesting example of how the order of steps does not always matter. This can be seen in Figure 11. In this figure, it is shown that a left rotation on the parent of $T2$ and $T3$, followed by a change within subtree $T2$ gives the same result as when these steps are done in the opposite order. The change within subtree $T2$ is implied by the change from $T2$ to $T2'$. What this shows us, is that rotations have no effect on the subtrees below it, and changes within subtrees also do not change the structure of the rest of the tree.

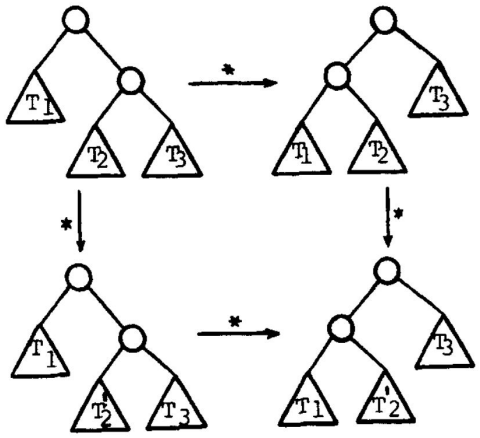


Figure 11: Different step orders give the same result, taken from [Pal87]

2.8 The SDD package

We use the SDD package [CD18b] to implement all our algorithms. This library can create, save, load and modify vtrees. The available operations to perform on vtrees are the left rotation, the right rotation and the swap. The library also has functions to navigate the tree, and to create a `.dot`-file corresponding to a vtree. This `.dot`-file can be used to create a `.png`-image. A more detailed overview of this package can be found in the beginner manual [CD18a].

3 Algorithms and Functionalities

The purpose of the desired algorithm is to find a path between two vtrees. This algorithm should take two vtrees as its input: the initial vtree and the target vtree. The output of the algorithm is the path from the initial vtree to the target vtree. A path is a sequence of steps, where each step is the application of one of the operations on an internal node. The operations we can use are described in Section 3.1.

We implement two algorithms for this task, called Algorithm 1 and Algorithm 2. Algorithm 2 is meant to improve on Algorithm 1. These algorithms are explained in Section 3.2.

To see the performance of the algorithms, we need a set of vtrees of different sizes to evaluate the performance, and how it is related to the tree size. We will also want to compare the difference between the two different algorithms. To do this, we require a set of random trees, and a set of trees which are expected to give Algorithm 2 a clear advantage. To compare the algorithms to each other, and see the effect of the tree size on the performance, we need a specific set of vtrees. The generation of these trees is explained in Section 3.3.

Lastly, we also want to know how big the chance is that the improvement in Algorithm 2 will be useful. To generate data to find this out, we use the function described in Section 3.4.

3.1 Operations

The operations we use in the algorithm are rotations and swaps. These are explained in Section 2.2. The swaps are part of the SDD Package, and these can easily be implemented. However, left and right rotations are only applicable in specific cases, and require a method to decide whether a left rotation or a right rotation is needed. To solve this problem, we replace left rotations and right rotations by up rotations.

The "Rotate up" operation rotates a given target node up. Specifically, a left rotation is applied to the target node if this target node is a right child, and a right rotation is applied to its parent if the target node is a left child. This operation is illustrated in Figure 12.

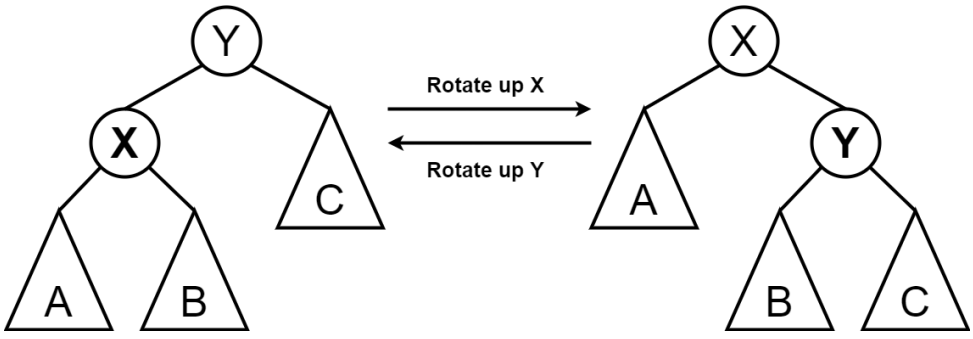


Figure 12: The "Rotate up" operation

3.2 Vtree Conversion Algorithms

3.2.1 Vtree Conversion Algorithm 1

The goal of the first algorithm is to simply find a path between a pair of two vtrees with the same set of variables. This algorithm can be split up into three basic functions.

As seen in Section 2.4, any binary tree structure can be converted to and from the left-linear tree using only rotations. We use this by converting both the initial vtree and the target vtree to the left-linear tree. The pseudo code for this function is given in Algorithm 1.

Algorithm 1 Convert to left-linear tree

```
1: x ← root node
2: while x is not a leaf do
3:     if right child of x is an internal node then
4:         x ← right child of x
5:         rotate up x
6:     else
7:         x ← left child of x
8:     end if
9: end while
```

This conversion to the left-linear tree will cost a maximum of $n-2$ rotations for a tree with n leaves. All the steps are saved, and for every rotation we also save whether the rotated node was originally a left child or a right child.

The second step is to reorder the variables. Since we want the tree to remain a left-linear tree during this sorting, we use the steps done in Figure 13. To swap B and C, we use three steps. First, we rotate up the parent of the lower leaf, which is node 1. Second, we swap the two variables we wanted to reorder. Finally, we reverse the change in the structure by rotating up node 2. The exception is sorting the two leftmost variables, since that only takes one swap. The pseudo code for this function is given in Algorithm 2. Since we can use only adjacent swaps, we will need a maximum of $(n-1)!$ swaps. Since every swap costs 3 operations with the exception of the first two variables, the maximum number of steps this reordering can take is $3 \cdot (n-1)! - 2$ steps.

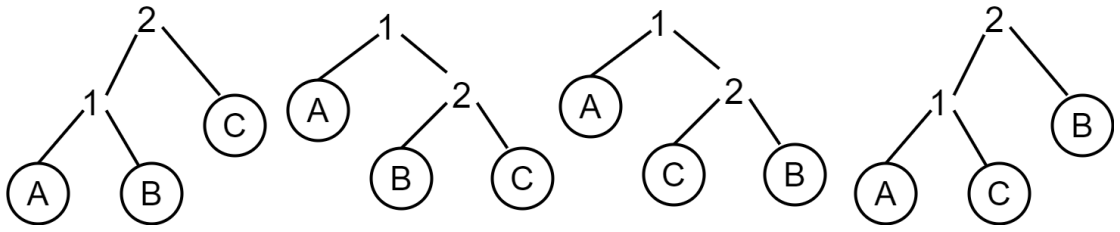


Figure 13: Swapping two variables in a left-linear tree

Algorithm 2 Reorder leaves in left-linear tree

```
1: i ← 0
2: while i < number of variables do
3:     x ← leaf that should be in position i
4:     while x is not in the right position do
5:         Swap x with its left neighbour                ▷ As seen in Figure 13
6:     end while
7:     i ← i + 1
8: end while
```

The third step is to take the path that was generated by the conversion of the target tree to a left-linear tree, and reverse this path. To achieve this reversed path, we need to know if the rotated nodes were left or right children. If a node we rotated was originally a left child, we can reverse this by rotating up its current right child and vice versa. This reverse path is now a path from the left-linear tree to the target tree.

Finally, we can concatenate the paths from these 3 steps to get the total path. The 3 paths represent:

1. Conversion to the left-linear tree
2. Reordering the variables
3. Conversion to the target tree

The output of the algorithm consists of the path. Every step consists of a type and a node id. The type can be either a swap, a rotation from the left and a rotation from the right. Rotations from the left and right are both up rotations, but the side implies whether the node was a left child or a right child before the rotation. The node id is the number given to the internal node. The internal nodes are numbered by inorder traversal so they stay consistent when rotations are done.

3.2.2 Vtree Conversion Algorithm 2

Based on Section 2.6 and Section 2.7, we know that it is possible to change subtrees without affecting the rest of the tree, and that we could pack equal subtrees together to simplify the problem. From these two conceptions we can deduce that it would be possible to solve subtrees first, and consequently marking these subtrees to be treated as single leaves.

A good example of the advantage of Algorithm 2 is the conversion from `vtree_2_5_01` to `vtree_2_5_02`. These vtrees are shown in Figure 14 and Figure 15. These vtrees have a shared subtree with the variables $\{B, C, E\}$.

When we want to find the path using Algorithm 1, we will end up with the following steps:

1. Rotate up node 5
2. Rotate up node 3

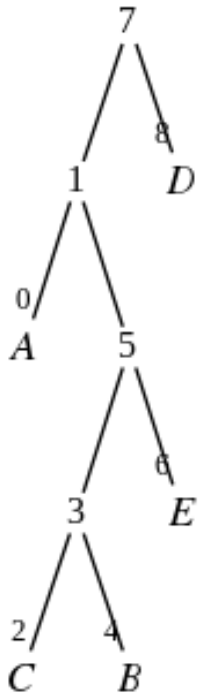


Figure 14: vtree_2_5_01

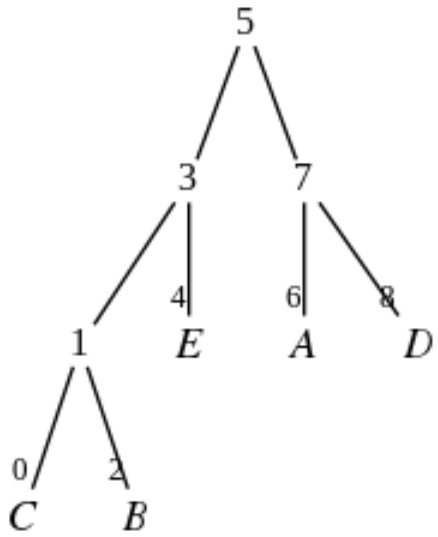


Figure 15: vtree_2_5_02

3. Swap on node 1
4. Rotate up node 1
5. Swap on node 3
6. Rotate up node 3
7. Rotate up node 3
8. Swap on node 5
9. Rotate up node 5
10. Rotate up node 5

In this path, the first two steps are to change the tree to a left-linear tree. Steps 3 moves the variable C to the left of A. Steps 4, 5 and 6 are to move the variable B to the left of A. Subsequently, we apply steps 7, 8 and 9 to move E to the left of A and reach the correct variable order. The last step is to rotate up node 5 to get the right structure. This path is 10 steps long.

When we want to find the path between these same vtrees using Algorithm 2, we instead get the following path:

1. Swap on node 1

2. Rotate up node 1

In this path, the shared subtree is already correct and does not need to be changed. Since the subtree is equal, it can be treated as if it's a single leaf. In essence, the the problem is now equivalent to Figure 16, which is a much simpler problem.

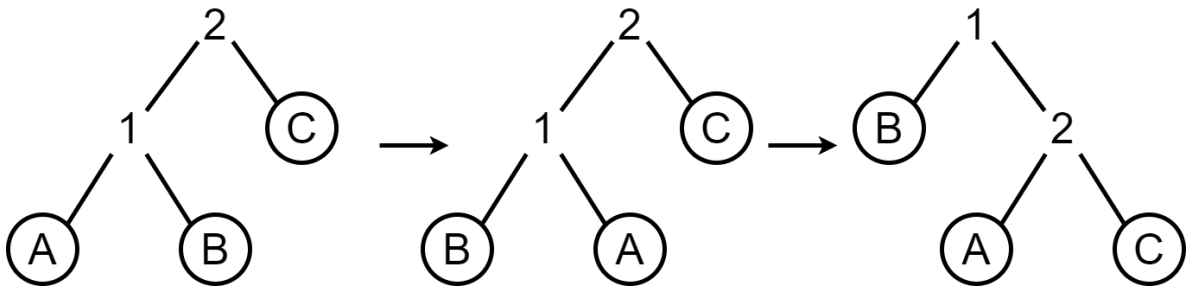


Figure 16: Equivalent conversion with the shared subtree represented as the variable B

Algorithm 2 works as follows. Until the initial vtree is equal to the target vtree, this second version of the algorithm repeatedly finds the first not yet solved shared subtree in the target vtree, solves this subtree and finally marks it as a solved subtree. In this context, a shared subtree is a subtree in the initial vtree, for which there exists a subtree in the target vtree that contains the same set of variables in its leaves. These subtrees are found by checking every subtree with postorder traversal using a recursive function. For every visited internal node, we check if the corresponding subtree is a shared subtree. The final subtree to be solved will always be the full vtree, since every tree of equal size shares the same set of variables.

We now give necessary and sufficient conditions which characterize the situations in which Algorithm 2 finds a shorter path than Algorithm 1, in Lemma 8. To this end, we first prove several lemmas.

Lemma 1. *If there exists no shared subtree in the vtree pair, Algorithm 1 and Algorithm 2 will produce the same conversion path.*

Proof. Algorithm 2 is a recursive implementation of Algorithm 1 that repeatedly finds the smallest shared subtree, converts this subtree in the initial vtree to the corresponding subtree in the target vtree, and reduces this subtree to a single leaf. If there exists no shared subtree in the vtree pair, Algorithm 2 will only convert the complete initial vtree to the target vtree. This means Algorithm 1 and Algorithm 2 will produce the same conversion path. □

Lemma 2. *The step to convert the an initial vtree to a left-linear vtree produces fewer operations in Algorithm 2 than in Algorithm 1, if there exists a shared subtree whose root is not on the left spine of the initial vtree.*

Proof. Assume we want to find a conversion path for a vtree pair, in which there exists a shared subtree whose root is not on the left spine of the initial vtree. This shared subtree has n leaves.

In Algorithm 1, the step to convert a vtree to a left-linear vtree will be done once. It will start at the top, and bring every internal node above the shared subtree to the left spine. We define the

number of operations this takes as x_1. After these operations, the root of the shared subtree will be the right child of a node on the left spine.
Subsequently, Algorithm 1 will bring the shared subtree to the left spine. Since the root of the shared subtree is a right child, rotating this node up will mean that both its children will both be right children of nodes on the left spine. This means that, to bring the complete shared subtree to the left spine, each of the internal nodes in this subtree will be rotated up. Any vtree with n leaves has $n-1$ internal nodes. Therefore, bringing a shared subtree with n leaves to the left spine will cost $n-1$ operations.
Finally, Algorithm 1 will bring the nodes below the shared subtree to the left spine. We define the number of operations this takes as x_2.
The step to convert a vtree to the left-linear vtree will produce a total of $x_1 + (n-1) + x_2$ operations in Algorithm 1.

In Algorithm 2, the step to convert a vtree to a left-linear vtree will be done twice. The first time we use this step, is when we independently convert the shared subtree to a left-linear vtree. It will cost at most $n-2$ operations to do this, because the shared subtree contains $n-2$ internal nodes under the root node. It will cost at least 0 operations, because it is possible for the subtree to already be a left-linear vtree.
The second time Algorithm 2 uses the step to convert a vtree to a left-linear vtree, is after the shared subtree has been reduced, to convert the complete initial vtree to a left-linear vtree. This step will start by bringing every internal node above the reduced shared subtree to the left spine. Just like in Algorithm 1, this takes x_1 operations. After these operations, the reduced shared subtree will be the right child of a node on the left spine. Since subtree reduction makes the algorithm treat the shared subtree as a single leaf, no operations are done on this shared subtree.
Finally Algorithm 2 will bring the nodes below the shared subtree to the left spine. Just like in Algorithm 1, this takes x_2 operations.

The step to convert an initial vtree to the left-linear vtree will produce a total of at least $0 + x_1 + x_2$ operations in Algorithm 2, and at most $(n-2) + x_1 + x_2$. This means that the step to convert the initial vtree to the left-linear vtree will produce at least 1 operation less, and at most $n-1$ operations less in Algorithm 2 than in Algorithm 1. □

Lemma 3. *The step to convert an initial vtree to a left-linear vtree produces equally as many operations in Algorithm 2 as in Algorithm 1, if there does not exist a shared subtree whose root is not on the left spine of the initial vtree.*

Proof. Lemma 1 shows that, if there exists no shared subtree in the vtree pair, Algorithm 1 and Algorithm 2 function the same and generate the same conversion path. This means that, if there exists no shared subtree in the vtree pair, the step to convert a vtree to a left-linear vtree will produce the same steps for both algorithms as well.

Assume we want to find a conversion path for a vtree pair, in which there exists a shared subtree whose root is on the left spine of the initial vtree.

In Algorithm 1, the step to convert a vtree to a left-linear vtree will be done once. It will start at the top, and bring every internal node above the shared subtree to the left spine. We define the

number of operations this takes as x_1. After these operations, the root of the shared subtree will be the left child of a node on the left spine.
Subsequently, Algorithm 1 will bring the shared subtree to the left spine. Since the root of the shared subtree is a left child, this will take the same number of operations as individually converting the shared subtree to a left-linear vtree. We define the number of operations this takes as x_2.
The step to convert a vtree to the left-linear vtree will produce a total of $x_1 + x_2$ operations in Algorithm 1.

In Algorithm 2, the step to convert a vtree to a left-linear vtree will be done twice. The first time we use this step, is when we independently convert the shared subtree to a left-linear vtree. This will take x_2 operations.
The second time Algorithm 2 uses the step to convert a vtree to a left-linear vtree, is after the shared subtree has been reduced, to convert the complete initial vtree to a left-linear vtree. This step will start by bringing every internal node above the reduced shared subtree to the left spine. Just like in Algorithm 1, this takes x_1 operations. After these operations, the reduced shared subtree will be the leftmost child of the complete initial vtree, and the complete initial vtree will be considered a left-linear vtree.
The step to convert an initial vtree to the left-linear vtree will produce a total of $x_2 + x_1$ operations in Algorithm 2. This means that the step to convert the initial vtree to the left-linear vtree will produce equally as many operations in Algorithm 2 as in Algorithm 1. □

Lemma 4. *The step to convert an initial vtree to a left-linear vtree produces fewer operations in Algorithm 2 than in Algorithm 1, iff there exists a shared subtree whose root is not on the left spine of the initial vtree.*

Proof. Lemma 2 shows that, if there exists a shared subtree whose root is not on the left spine of the initial vtree, the step to convert the a vtree to a left-linear vtree produces fewer operations in Algorithm 2 than in Algorithm 1.
Lemma 3 shows that, if there does not exist a shared subtree whose root is not on the left spine of the initial vtree, the step to convert a vtree to a left-linear vtree produces equally as many operations in Algorithm 2 as in Algorithm 1.
This means that the step to convert an initial vtree to a left-linear vtree only produces fewer operations in Algorithm 2 than in Algorithm 1, if there exists a shared subtree whose root is not on the left spine of the initial vtree. □

Lemma 5. *The number of operations we use to convert a left-linear vtree to a target vtree is equal to the number of operations we use to convert this target vtree to a left-linear vtree.*

Proof. We obtain the conversion path from a left-linear vtree to a target vtree by determining the path from the target vtree to the left-linear vtree, and reversing this path. We reverse a path by taking the left-linear vtree and the conversion path, reversing the order of the operations and finally replacing each operation by the operation with the inverse effect. This means that, if we find a conversion path from a target vtree T_T to a left-linear vtree T_{LL}, whose path length is x, the conversion path we find from the left-linear vtree T_{LL} to the target vtree T_T will also consist of x operations. □

Lemma 6. *The step to convert a left-linear vtree to a target vtree produces fewer operations in Algorithm 2 than in Algorithm 1, iff there exists a shared subtree whose root is not on the left spine of the target vtree.*

Proof. Lemma 4 shows that, in Algorithm 2, the step to convert an initial vtree to a left-linear vtree produces fewer operations than Algorithm 1 iff there exists a shared subtree whose root is not on the left spine of the initial vtree.
Lemma 5 shows that a conversion path from a left-linear vtree to a target tree consists of equally as many operations as its reverse path.
These lemma's show us that the step to convert a left-linear vtree to a target vtree also produces fewer operations in Algorithm 2 than in Algorithm 1, iff there exists a shared subtree whose root is not on the left spine of the target vtree. □

Lemma 7. *The step to reorder leaves in a left-linear vtree produces equally as many operations in Algorithm 2 as in Algorithm 1, if there exists no shared subtree whose root node is not on the left spine of both the initial vtree and the target vtree.*

Proof. Lemma 1 shows that, if there exists no shared subtree in the vtree pair, Algorithm 1 and Algorithm 2 function the same and generate the same conversion path. This means that, if there exists no shared subtree in the vtree pair, the step to reorder leaves in a left-linear vtree will produce the same steps for both algorithms as well.

Assume we want to find a conversion path for a vtree pair, in which there exists a shared subtree with n leaves, whose root is on the left spine of both the initial vtree and the target vtree.

In Algorithm 1, the step to reorder leaves in a left-linear vtree will be done once. It will start by finding the desired order of the variables, based on the target vtree. Algorithm 1 will repeatedly take the leftmost variable from this ordered list of variables which is not yet in the correct position, and find the corresponding leaf. Since the root of the shared subtree is on the left spine of the target vtree, the first n variables in the desired variable order are the variables of the shared subtree. We define the number of operations produced to put the first n leaves in the correct position as x_1. Since the shared subtree also has its root on the left spine of the initial vtree, no adjacent swaps are done with any variables outside of the shared subtree until this point.
After these first n leaves are moved to the correct position, Algorithm 1 continues by moving all remaining leaves to the correct position. We define the number of operations produced to reorder these remaining leaves as x_2.
If there exists a shared subtree whose root is on the left spine of both the initial vtree and the target vtree, the step to reorder leaves in a left-linear vtree will produce a total of $x_1 + x_2$ operations in Algorithm 1.

In Algorithm 2, the step to reorder leaves in a left-linear vtree will be done twice. The first time we use this step, is when we independently reorder the variables of the shared subtree. This will produce x_1 operations.
The second time Algorithm 2 uses the step to reorder leaves in a left-linear vtree, is after the shared subtree has been reduced, to reorder the leaves in the complete initial vtree. Since the reduced

shared subtree is treated as a single leaf which is already in the correct position, reordering the variables will produce x_2 operations.
If there exists a shared subtree whose root is on the left spine of both the initial vtree and the target vtree, the step to reorder leaves in a left-linear vtree will produce a total of $x_1 + x_2$ operations in Algorithm 2.

If we want to find a conversion path for a vtree pair, in which there is no shared subtree, the step to reorder leaves in a left-linear vtree will produce the same steps for Algorithm 1 and Algorithm 2.

If we want to find a conversion path for a vtree pair, in which there is a shared vtree whose root is on the left spine of both the initial vtree and the target vtree, the step to reorder leaves in a left-linear vtree will produce $x_1 + x_2$ operations for both Algorithm 1 and Algorithm 2.

Therefore, if we want to find a conversion path for a vtree pair, in which there is not a shared vtree whose root is not on the left spine of both the initial vtree and the target vtree, the step to reorder leaves in a left-linear vtree will produce equally as many operations in Algorithm 2 as in Algorithm 1. □

Lemma 8. *Algorithm 2 produces a shorter path than Algorithm 1, iff there exists at least one shared subtree, whose root is not on both the left spine of the initial vtree, and the left spine of the target vtree.*

Proof. Both Algorithm 1 and Algorithm 2 only use the steps to convert an initial vtree to a left-linear vtree, to reorder variables in a left-linear vtree and to convert a left-linear vtree to a target vtree.

Assume we want to find a conversion path for a vtree pair, in which there exists a shared subtree whose root is not on the left spine of both the initial vtree and the target vtree.
We can conclude that at least one of the following is true. The initial vtree contains a shared subtree whose root is not on the left spine, or the target vtree contains a shared subtree whose root is not on the left spine.
Lemma 4 shows us that, if the initial vtree contains a shared subtree whose root is not on the left spine, Algorithm 2 will find a way to reduce the full conversion path compared to Algorithm 1.
Lemma 6 shows us that, if the target vtree contains a shared subtree whose root is not on the left spine, Algorithm 2 will find a way to reduce the full conversion path compared to Algorithm 1.
Therefore, we can conclude that if there exists a shared subtree whose root is not on the left spine of both the initial vtree and the target vtree, Algorithm 2 will find a way to reduce the full conversion path compared to Algorithm 1.

Assume we want to find a conversion path for a vtree pair, in which there does not exist a shared subtree. Lemma 1 shows us that, in this situation, Algorithm 2 produces the same conversion path as Algorithm 1.

Assume we want to find a conversion path for a vtree pair, in which there exists a shared subtree whose root node is on the left spine of both the initial vtree and the target vtree.

Lemma 4 shows us that, in this situation, the step to convert an initial vtree to a left-linear vtree produces equally as many operations for Algorithm 1 and Algorithm 1.
Lemma 7 shows us that, in this situation, the step to reorder variables in a left-linear vtree produces equally as many operations for Algorithm 1 and Algorithm 1.
Lemma 6 shows us that, in this situation, the step to convert a left-linear vtree to a target vtree produces equally as many operations for Algorithm 1 and Algorithm 1.
Since both algorithms consist of only these three steps, Algorithm 1 and Algorithm 2 produce conversion paths consisting of an equal number of operations, when there exists a shared subtree whose root is on the left spine of both the initial vtree and the target vtree.

From these arguments, we can conclude that Algorithm 2 reduces the length of the conversion path compared to Algorithm 1, iff there exists at least one shared subtree, whose root is not on both the left spine of the initial vtree, and the left spine of the target vtree. □

3.3 Vtree generation

Since we will need a good data set to test the performance of our algoritms, as well as a single data set to make sure all results are reliably consistent, we have to create a set of vtrees that can be reused. To generate these vtrees, we have the function Mktrees 1. To make sure we can compare the first and second version of the algorithm, we also want a separate set of vtrees which are expected to give a better path when processed by the second algorithm. These vtrees are generated by Mktrees 2, Mktrees 3 and Mktrees 4.

3.3.1 Mktrees 1

The first tree generation function creates random vtrees using the random vtree generation function of the SDD Package. The downside of this function is that it uses the time in seconds as a seed for its random vtree generation. This means that when we want to generate many vtrees quickly, many of these vtrees will be generated in the same second, and thus be equal. To solve this problem, a function is added to check if a vtree is equal to one of the already generated vtrees. If this is the case, the generation function will repeatedly be used until a new vtree is created. This means that this tree generation can roughly create 1 vtree per second. Since we want only unique trees, the limit of the number of generated trees is limited to the amount of unique possibilities.

3.3.2 Mktrees 2

To test the optimal conditions for the second vtree conversion algorithm, we use the function Mktrees 2 to generate vtree pairs that share a subtree which is equal in structure and variable order. To do this, we first generate a random vtree. In this vtree, we find the lowest leaf and move up 2 nodes from this. To avoid any problems with small trees, the smallest tree generated by this function has 5 leaves. This subtree is then marked, and 1000 random rotations or swaps are done on random internal nodes that are not part of the chosen subtree. This way we end up with a new tree that is completely different except for this subtree. Every tree that is generated to be random, except for the first subtree of a size, is generated by picking a randomly chosen vtree from the

set of previously created vtrees, and applying the 1000 random operations without having to stay outside of a certain subtree.

3.3.3 Mktrees 3

The Mktrees 3 function is a small change to Mktrees 2. In this function, the marked subtree is also changed, but the set of variables in this subtree is kept the same. This means that two corresponding vtrees will have the a shared subtree that contains the same set of variables, whereas the structure of the subtree or the order of the variables aren't necessarily equal.

3.3.4 Mktrees 4

Mktrees 4 is very similar to Mktrees 3, but with one big difference. The subtree for which we want to keep the same variable set, is now not determined by finding the lowest leaf and going up by 2 nodes from there. Instead, we use a random number generator to pick any internal node other than the root. This allows us to also get bigger shared subtrees, or subtrees in other parts of the vtree.

3.4 Shared subtree occurrence

3.4.1 Mktrees 5

Mktrees 5 is a function which works very similar to the previous 4 vtree generation functions. However, no subtrees are selected to be kept, and every vtree but the first is generated by picking a random previous vtree and applying 1000 randomly chosen operations on random internal nodes. Instead of saving the generated vtrees to a file, Mktrees 5 instead checks for every pair of 2 vtrees if they contain a shared subtree with the same variable set. For every number of variables, the number of found shared subtrees is saved. After the function has been executed, a list will be given with how often a shared subtree was found for each of the tree sizes.

4 Experimental setup

The algorithms and all additional functionalities from Section 3 are implemented in C, making use of the SDD Package by A. Choi and A. Darwiche. The source code can be found in the GitHub repository [vE21]. This code is used to test the performance of our algorithms, and to generate vtrees for testing and evaluation.

4.1 Comparing the algorithms

4.1.1 Random vtrees

In order to test how well our algorithms perform on a set of random vtrees, we need to generate such vtrees. To find out how this performance correlates to the number of variables in the vtrees, we want to generate vtrees for multiple different variable counts. We use the Mktrees 1 function from Section 1 to generate these random vtrees. We divide the set of generated vtrees into vtree pairs. In each of these pairs, the first vtree will be used as the initial vtree, and the second vtree will be used as the target vtree.

Since every pair consists of two vtrees with the same size, every vtree size should have an even number of generated vtrees. Because the vtree generation only gives us unique vtrees, this means that smaller vtrees will have fewer examples, and are therefore less test data.

The following amounts vtrees are created by Mktrees 1 for the different numbers of leaves:

Number of leaves	Number of vtrees
2	2
3	4
4	10
5	10
10	20
20	40
30	60

Table 1: Number of vtrees generated by Mktrees 1

These vtrees are divided into vtree pairs, and for evaluation every pair gives the following output:

- Tree size, the number of leaves/variables in the vtrees
- Tree type, the function by which the vtree was generated
- Algorithm, the algorithm that processed the vtree pair
- Number of steps, the number of operations done to change the initial vtree to the target vtree

The number of steps is the evaluation metric for the performance of the algorithm, and a better algorithm is expected to give us shorter paths (i.e. a lower number of steps). Since we process each pair with both algorithms, we can compare the resulting path lengths between the algorithms. From this comparison, we also find a difference and a difference ratio for each pair.

The difference is the number of steps Algorithm 2 does more than Algorithm 1. Since Algorithm 2 is an improvement, we expect this value to be between 0 and a small negative number. The difference ratio is the difference divided by the path length from Algorithm 1. This value is expected to be between 0 and -1, since we expect Algorithm 2 to improve, but not to perform twice as well as Algorithm 1.

4.1.2 Designed vtrees

The reason we also want to test on special types of vtrees, is because Algorithm 2 is expected to be equal to Algorithm 1 in the general case, but improve on it in special cases. These special cases are vtree pairs which have a shared subtree. A shared subtree means that the initial vtree has a subtree with a set of variables, and another subtree in the target vtree can be found which has the same set of variables. The structure and variable order of this subtree are not required to be equal, only the set of variables. Three different types of such special vtree pairs can be generated using the functions Mktrees 2, Mktrees 3 and Mktrees 4 from Section 3.3.2, Section 3.3.3 and Section 3.3.4. To represent these types equally, each of the corresponding functions will create the same number of vtrees for every vtree size. Since Mktrees 2 and Mktrees 3 can't consistently create shared subtrees in vtrees with less than 5 variables, the lowest number of variables to be used here is 5. The synthestetic examples generated by these functions are probably worse than real-world scenarios, which means that the actual performance of the tree manipulation algorithms is expected to be better than the experimental results show. Table 2 shows the number of generated vtrees.

Number of variables	Number of vtrees
5	2
10	20
20	40
30	60

Table 2: Number of vtrees generated by Mktrees 2, Mktrees 3 and Mktrees 4 each

Just like in Section 4.1.1, the vtrees are changed into pairs, and each pair is evaluated based on the path length, the difference among algorithms and the difference ratio. This gives us 61 vtree pairs for every one of the vtree types.

4.2 Shared subtree occurrence

Algorithm 2 is designed to improve on Algorithm 1 in the cases that the two vtrees have a shared subtree. To find out what the difference between the two algorithms is, we don't only want to know how much better Algorithm 2 would perform in these special cases, but also what the probability is that a random vtree pair is one of these special cases.

To find these results, we use the function Mktrees 5 from Section 3.4.1 to create 20000 vtrees for each of the following variable counts: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, and 50. The vtrees will be divided into pairs, and instead of saving these vtrees, the function will give us the number of pairs that contain a shared subtree. Since keeping 20000 vtrees in memory is a lot, we instead use the Mktrees 5 function 10 times with 2000 vtrees each, and sum up the total for each vtree size. Finally, this number is divided by 10000 to estimate the probability of any random vtree pair having a shared subtree with the same set of variables.

5 Results

5.1 Comparing the algorithms

The results of the experiments can be found in Appendix A. The algorithms failed to produce a path for some of the vtrees, but these missing data values appeared to occur at random vtree pairs. Since we use all values to calculate averages, and most missing values appear on vtree types and sizes that have many examples, the remaining data for these vtree sizes and types is still enough to get an accurate estimate. However, since we do not have a lot of examples of vtrees with 5 variables or less, the averages for these vtrees may be less accurate.
For every vtree pair with path length x_1 from Algorithm 1, and path length x_2 from Algorithm 2, we calculate the difference and difference ratio as follows:

The difference is $d = x_2 - x_1$.
The difference ratio is $r = \frac{d}{x_1} = \frac{x_2 - x_1}{x_1}$.

For the path lengths, the difference and the difference, the values are sorted by tree type and the average is calculated for each tree size. This average value is plotted in Figure 17, Figure 19 and Figure 22.

5.1.1 Random trees

The type 1 vtrees are the vtrees that were generated by Mktrees 1. When we compare the performance of the algorithms in Figure 17, the we see that the black line is slightly below the gray line. This means that, on average, the path found by Algorithm 2 is slightly shorter than the path found by Algorithm 1. What we can also see, is that the spread of data points increases as the vtree we test grows bigger.

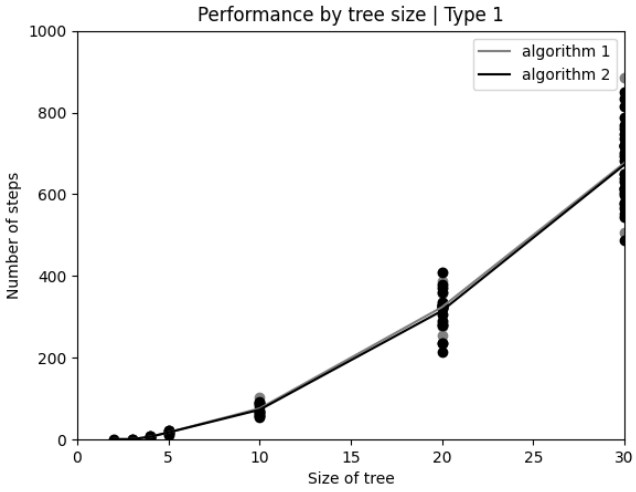


Figure 17: Performance of the algorithms for vtrees of type 1

In Figure 18, we can see that the performance of the two algorithms is equal for the vtrees of 5 or fewer variables. As we can see in the right graph, Algorithm 2 performs slightly better with the vtrees of size 10, but this advantage of the second algorithm slightly decreases between the tree sizes 10 and 30.

We see that Algorithm 2 performs slightly better than Algorithm 1. However, there was no improvement on trees with 5 variables or less. This can be explained by two things. Firstly, it is impossible for a vtree with 1 or 2 variables to contain a shared subtree, since the smallest shared subtree can only occur in a vtree with 3 variables. A second explanation is that Algorithm 2 does not necessarily perform better if the vtree pair has a shared subtree.

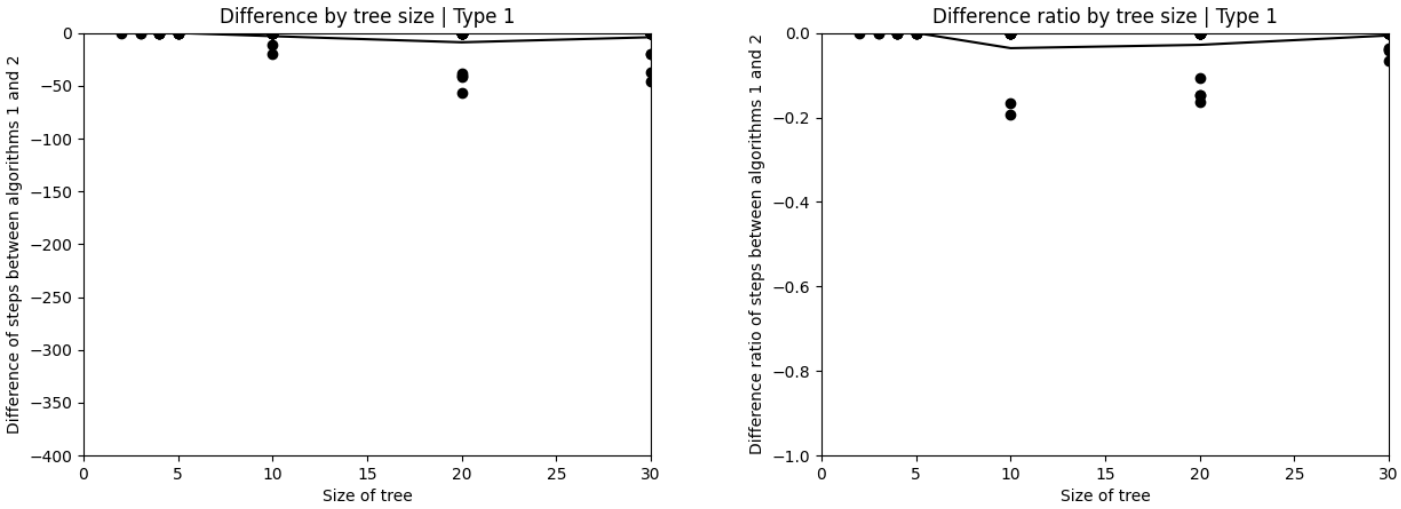


Figure 18: Difference (left) and difference ratio (right) of performance between the algorithms for vtrees of type 1

5.1.2 Designed vtrees

The type 2, type 3 and type 4 vtrees are those generated by Mktrees 2, Mktrees 3 and Mktrees 4 respectively.

When we look at the performance of the algorithms on the type 2 and type 3 vtrees in Figure 19, the main thing we see is that Algorithm 2 gives us a significantly shorter path than Algorithm 1. We also see that the performance of Algorithm 1 in these figures is roughly the same as its performance on the type 1 vtrees, as seen in Figure 17.

When looking at the average difference between the algorithms in Figure 20, we see that it appears to have a linear relation with the number of variables. Bigger trees lead to a bigger difference in the generated path length. However, when looking at the average difference ratio between the algorithms in Figure 21, the line through the average ratios looks like a reciprocal function whose limit is around 15%. The advantage of Algorithm 2 relative to the path length decreases, but the

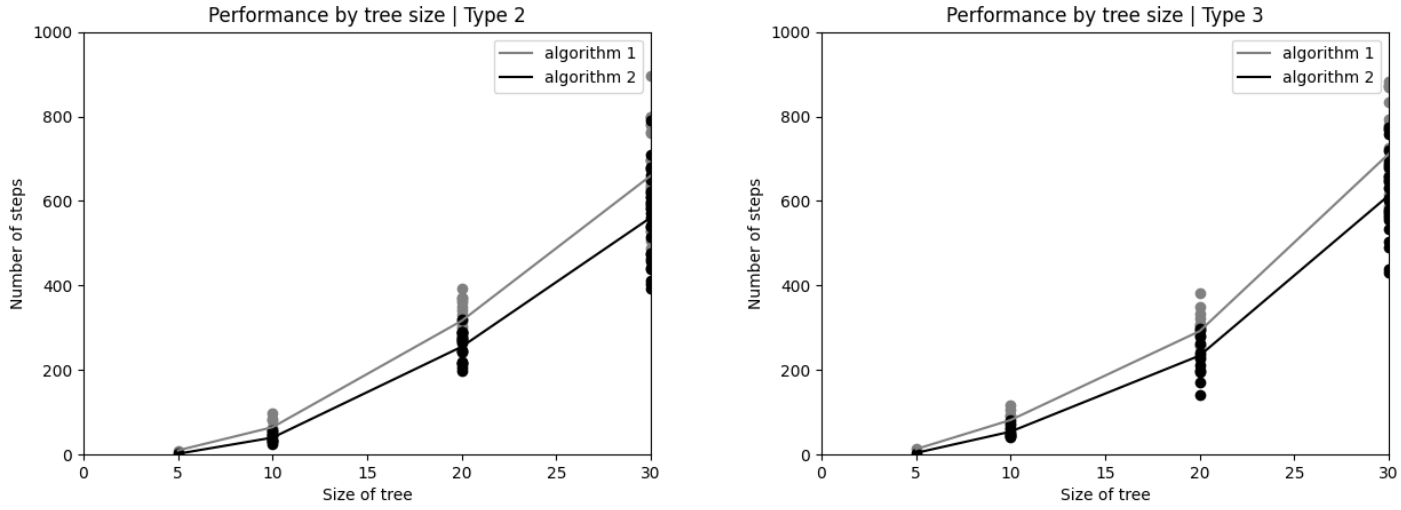


Figure 19: Performance between the algorithms for vtrees of type 2 (left) and type 3 (right)

step by which it decreases also gets exponentially smaller.

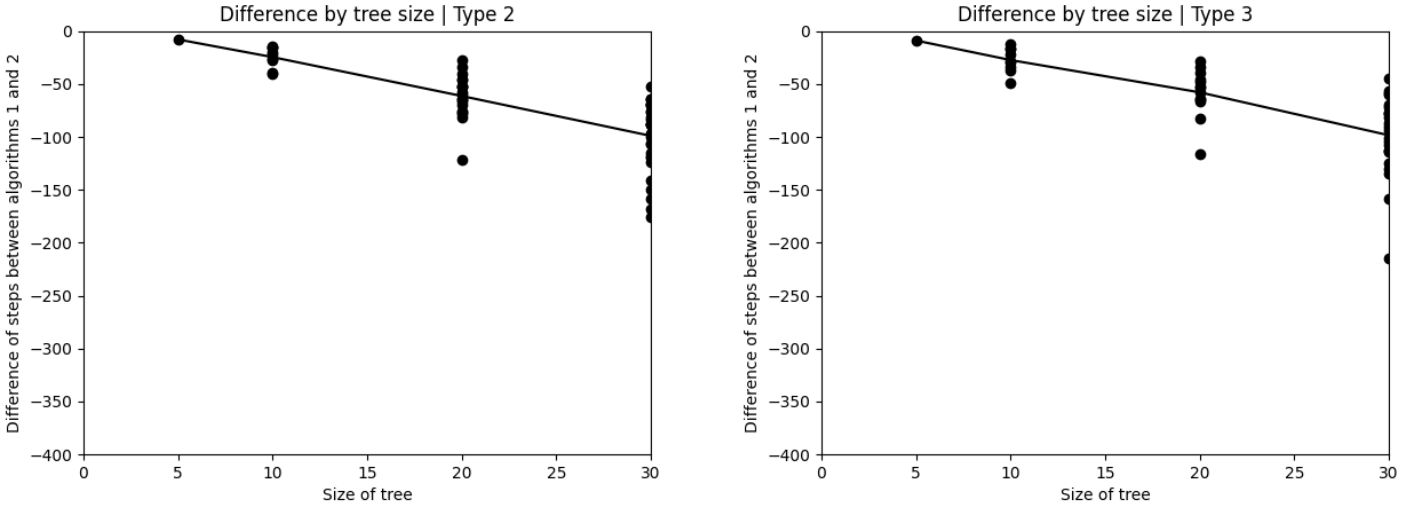


Figure 20: Difference in performance between the algorithms for vtrees of type 2 (left) and type 3 (right)

Algorithm 2 also performs better on type 4 vtrees, similar to the type 2 and type 3 vtrees. This can be seen in Figure 22. However, the advantage of Algorithm 2 is slightly greater for this type of vtree. This can also be seen in Figure 23, as the lines through the means are lower. The limit of the line in the right graph of Figure 23 is greater than the 15% we have seen in Figure 21.

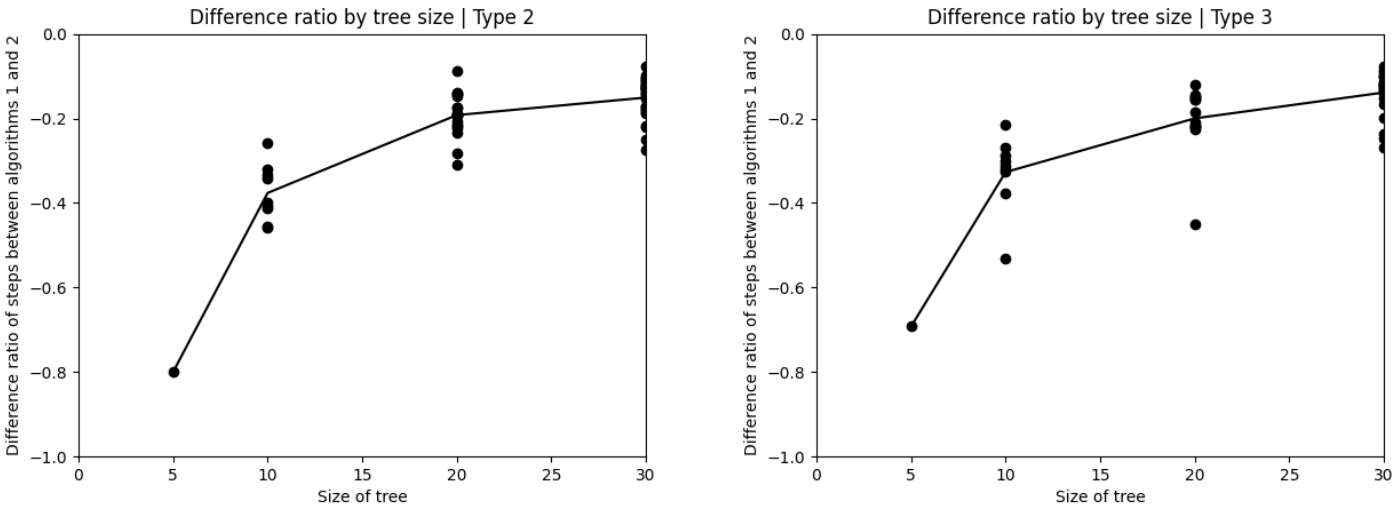


Figure 21: Difference ratio for performance between the algorithms for vtrees of type 2 (left) and type 3 (right)

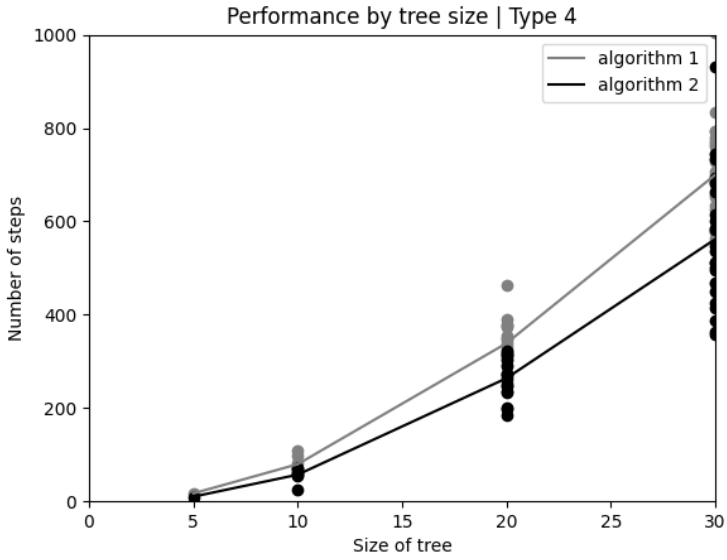


Figure 22: Performance of the algorithms for vtrees of type 4

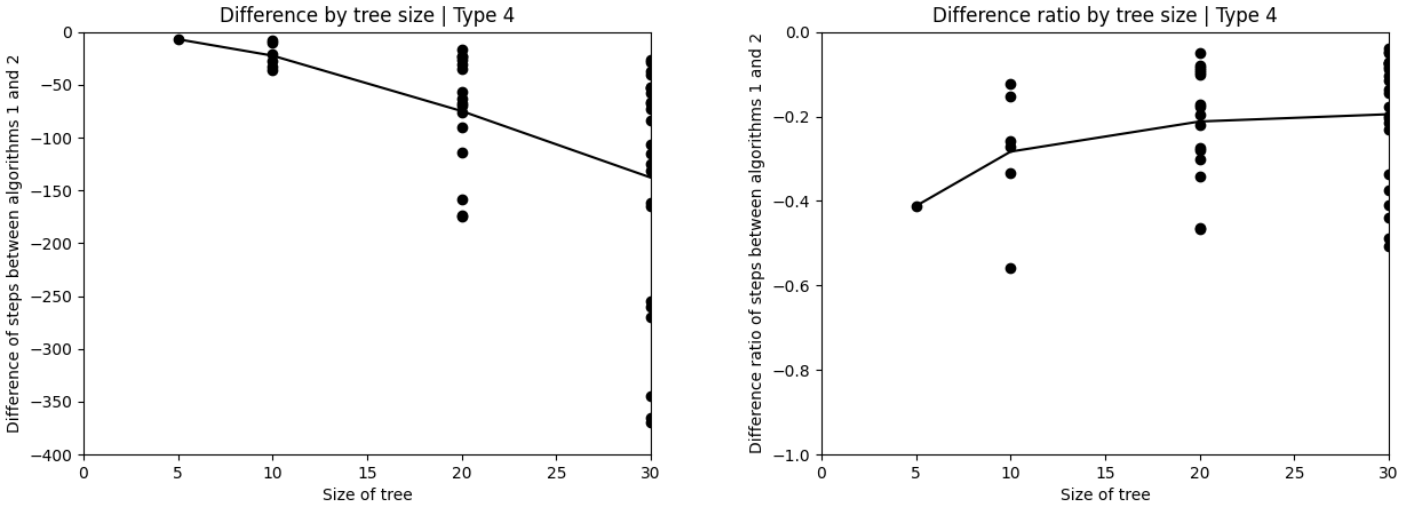


Figure 23: Difference (left) and difference ratio (right) of performance between the algorithms for vtrees of type 4

5.2 Shared subtree occurrence

Figure 24 shows the proportion of vtree pairs from Mktrees 5 that share a subtree with the same set of variables. The values that are shown in this plot can be found in Table 5 from Appendix A.2. In this figure, we see that this proportion is 0% for the trees with 1 or 2 variables, around 33% for trees with 3 or 4 variables and slowly appears to approach a limit around 12.5%.

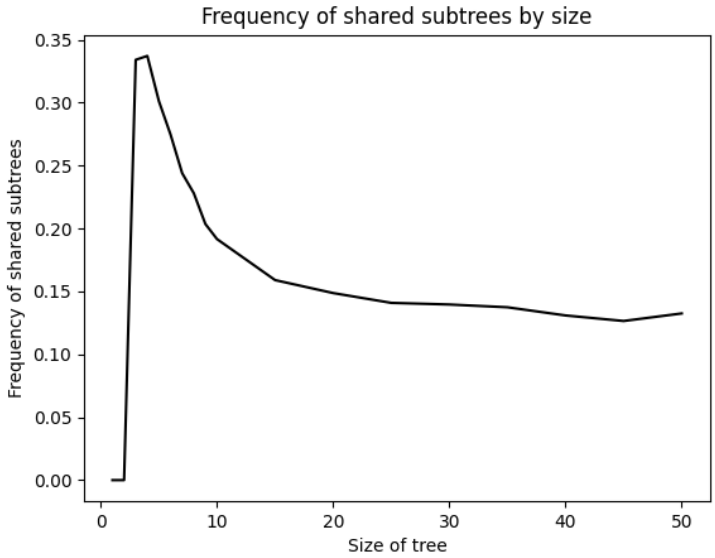


Figure 24: Probability of a shared subtree occurring in random vtree pairs

6 Conclusion

In this thesis, we have proposed two algorithms to convert one vtree to another. Algorithm 1 converts a vtree to a left-linear tree, reorders the variables and finally converts the vtree to the target vtree. Algorithm 2 adds recursion by solving shared subtrees first, and using subtree reduction to represent these subtrees as leaves.

We have shown that a path between 2 vtrees can always be found, because every vtree can be converted to a left-linear vtree, and every variable order can be achieved by the variable reordering step. We have also shown that the probability a shared subtree occurs in a pair of 2 randomly generated vtrees is always greater than 12.5%, and that the average improvement of Algorithm 2 compared to Algorithm 1 in these cases is always greater than 15%. We can conclude this, because both Figure 24 and the graphs in Figure 21 and Figure 23 showing the difference ratio approach a non-zero limit around these values. However, since we expect vtree pairs we want to solve to be more similar than 2 random vtrees, and also have bigger shared subtrees, we expect these percentages to be higher in reality. Finally, we have shown in Lemma 8 that Algorithm 2 produces a shorter path than Algorithm 1, iff there exists at least one shared subtree, for which the root is not on both the left spine of the initial vtree, and the left spine of the target vtree.

7 Future work

To improve on this thesis, or to validate the conclusions, it would be good to repeat some of the experiments with a larger set of vtrees. In these experiments, it could also be useful to make a distinction between the number of rotations and the number of swaps. This information would be useful in a situation where the cost of swaps and rotations is different.

Something that would be interesting to try, is to implement Algorithm 2 to recursively solve each subtree twice: one time with the left-linear tree and another time with the right-linear tree. The shortest of these two paths could be used as the solution of this subtree, and the final path could then be a combination of the two methods.

It would also be possible to improve the variable reordering step of the algorithms. An example is to iteratively select a variable from right to left in the desired variable order, find this variable in the vtree we want to reorder, and move this variable into the desired position. With this method, we take more advantage of the fact that, when we reorder variables in a left-linear tree, an adjacent swap on the two leftmost variables costs 2 operations less than an adjacent swap in any other position.

Another possible way to improve the algorithm, is by testing if a conversion from a vtree a to another vtree b produces a different number of steps than a conversion from vtree b to vtree a, and how this knowledge could be used to increase the efficiency.

Finally, it would also be a good idea to change the algorithm by using a different evaluation metric than the number of steps. An example of another possible metric would be the size of the largest SDD corresponding to the vtrees. This metric may give different paths for the same pair of vtrees, if the Boolean function to be represented by the SDD is different.

References

[BS77] J.-L. Baer and B. Schwab. A comparison of tree-balancing algorithms. *Commun. ACM*, 20(5):322–330, May 1977.

[CD13] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1), June 2013.

[CD18a] Arthur Choi and Adnan Darwiche. *SDD Beginning-User Manual*. Automated Reasoning Group, UCLA, January 2018.

[CD18b] Arthur Choi and Adnan Darwiche. The sdd package. http://reasoning.cs.ucla.edu/sdd, Jan 2018.

[CIW82] Karel Culik II and Derick Wood. A note on some tree similarity measures. *Information Processing Letters*, 15(1):39–42, 1982.

[CKD13] Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In Linda C. van der Gaag, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 121–132, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CS09] Sean Cleary and Katherine St. John. Rotation distance is fixed-parameter tractable. *Information Processing Letters*, 109(16):918–922, 2009.

[CS10] Sean Cleary and Katherine St. John. A linear-time approximation algorithm for rotation distance. *Journal of Graph Algorithms and Applications*, 14(2):385–390, 2010.

[DDST88] Robert E. Tarjan Daniel D. Sleator and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *J. Amer. Math. Soc.*, 1(3), July 1988.

[dJ21] Rachel G. de Jong. A genetic algorithm for minimizing sentential decision diagrams (unpublished), 2021.

[KVdBCD14] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[Pal87] Jean Pallo. On the rotation distance in the lattice of binary trees. *Information Processing Letters*, 25(6):369–373, 1987.

[PV07] Suri Pushpa and Prasad Vinod. Binary search tree balancing methods: A critical study. *IJCSNS International Journal of Computer Science and Network Security*, 7(8):237–243, 2007.

[vE21] Diego van Egmond. Bachelorproject. https://git.liacs.nl/s2243458/bachelorproject, June 2021.

[VL20] Lieuwe Vinkhuijzen and Alfons Laarman. Symbolic model checking with sentential decision diagrams. In Jun Pang and Lijun Zhang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 124–142, Cham, 2020. Springer International Publishing.

A Results

A.1 Algorithm output

A.1.1 Output of Algorithm 1

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
10	vtree_1_10_01.vt	vtree_1_10_02.vt	1	1	68
10	vtree_1_10_03.vt	vtree_1_10_04.vt	1	1	88
10	vtree_1_10_05.vt	vtree_1_10_06.vt	1	1	66
10	vtree_1_10_07.vt	vtree_1_10_08.vt	1	1	59
10	vtree_1_10_09.vt	vtree_1_10_10.vt	1	1	92
10	vtree_1_10_11.vt	vtree_1_10_12.vt	1	1	57
10	vtree_1_10_13.vt	vtree_1_10_14.vt	1	1	91
10	vtree_1_10_15.vt	vtree_1_10_16.vt	1	1	66
10	vtree_1_10_17.vt	vtree_1_10_18.vt	1	1	71
10	vtree_1_10_19.vt	vtree_1_10_20.vt	1	1	103
20	vtree_1_20_01.vt	vtree_1_20_02.vt	1	1	357
20	vtree_1_20_03.vt	vtree_1_20_04.vt	1	1	329
20	vtree_1_20_05.vt	vtree_1_20_06.vt	1	1	326
20	vtree_1_20_07.vt	vtree_1_20_08.vt	1	1	329
20	vtree_1_20_09.vt	vtree_1_20_10.vt	1	1	322
20	vtree_1_20_11.vt	vtree_1_20_12.vt	1	1	305
20	vtree_1_20_13.vt	vtree_1_20_14.vt	1	1	255
20	vtree_1_20_15.vt	vtree_1_20_16.vt	1	1	379
20	vtree_1_20_17.vt	vtree_1_20_18.vt	1	1	278
20	vtree_1_20_19.vt	vtree_1_20_20.vt	1	1	335
20	vtree_1_20_21.vt	vtree_1_20_22.vt	1	1	291
20	vtree_1_20_23.vt	vtree_1_20_24.vt	1	1	388
20	vtree_1_20_25.vt	vtree_1_20_26.vt	1	1	408
20	vtree_1_20_27.vt	vtree_1_20_28.vt	1	1	328
20	vtree_1_20_29.vt	vtree_1_20_30.vt	1	1	361
20	vtree_1_20_31.vt	vtree_1_20_32.vt	1	1	283
20	vtree_1_20_33.vt	vtree_1_20_34.vt	1	1	370
20	vtree_1_20_35.vt	vtree_1_20_36.vt	1	1	277
20	vtree_1_20_37.vt	vtree_1_20_38.vt	1	1	322
20	vtree_1_20_39.vt	vtree_1_20_40.vt	1	1	237
2	vtree_1_2_01.vt	vtree_1_2_02.vt	1	1	1

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_1_30_01.vt	vtree_1_30_02.vt	1	1	567
30	vtree_1_30_03.vt	vtree_1_30_04.vt	1	1	746
30	vtree_1_30_05.vt	vtree_1_30_06.vt	1	1	720
30	vtree_1_30_07.vt	vtree_1_30_08.vt	1	1	736
30	vtree_1_30_09.vt	vtree_1_30_10.vt	1	1	692
30	vtree_1_30_11.vt	vtree_1_30_12.vt	1	1	613
30	vtree_1_30_13.vt	vtree_1_30_14.vt	1	1	507
30	vtree_1_30_15.vt	vtree_1_30_16.vt	1	1	760
30	vtree_1_30_17.vt	vtree_1_30_18.vt	1	1	572
30	vtree_1_30_19.vt	vtree_1_30_20.vt	1	1	787
30	vtree_1_30_21.vt	vtree_1_30_22.vt	1	1	631
30	vtree_1_30_23.vt	vtree_1_30_24.vt	1	1	770
30	vtree_1_30_25.vt	vtree_1_30_26.vt	1	1	616
30	vtree_1_30_27.vt	vtree_1_30_28.vt	1	1	544
30	vtree_1_30_29.vt	vtree_1_30_30.vt	1	1	886
30	vtree_1_30_31.vt	vtree_1_30_32.vt	1	1	651
30	vtree_1_30_33.vt	vtree_1_30_34.vt	1	1	577
30	vtree_1_30_35.vt	vtree_1_30_36.vt	1	1	835
30	vtree_1_30_37.vt	vtree_1_30_38.vt	1	1	683
30	vtree_1_30_39.vt	vtree_1_30_40.vt	1	1	605
30	vtree_1_30_41.vt	vtree_1_30_42.vt	1	1	598
30	vtree_1_30_43.vt	vtree_1_30_44.vt	1	1	615
30	vtree_1_30_45.vt	vtree_1_30_46.vt	1	1	579
30	vtree_1_30_47.vt	vtree_1_30_48.vt	1	1	717
30	vtree_1_30_49.vt	vtree_1_30_50.vt	1	1	814
30	vtree_1_30_51.vt	vtree_1_30_52.vt	1	1	700
30	vtree_1_30_53.vt	vtree_1_30_54.vt	1	1	639
30	vtree_1_30_55.vt	vtree_1_30_56.vt	1	1	720
30	vtree_1_30_57.vt	vtree_1_30_58.vt	1	1	696
30	vtree_1_30_59.vt	vtree_1_30_60.vt	1	1	717
3	vtree_1_3_01.vt	vtree_1_3_02.vt	1	1	1
3	vtree_1_3_03.vt	vtree_1_3_04.vt	1	1	1
4	vtree_1_4_01.vt	vtree_1_4_02.vt	1	1	6
4	vtree_1_4_03.vt	vtree_1_4_04.vt	1	1	9
4	vtree_1_4_05.vt	vtree_1_4_06.vt	1	1	6
4	vtree_1_4_07.vt	vtree_1_4_08.vt	1	1	4
4	vtree_1_4_09.vt	vtree_1_4_10.vt	1	1	9
5	vtree_1_5_01.vt	vtree_1_5_02.vt	1	1	18
5	vtree_1_5_03.vt	vtree_1_5_04.vt	1	1	11
5	vtree_1_5_05.vt	vtree_1_5_06.vt	1	1	21
5	vtree_1_5_07.vt	vtree_1_5_08.vt	1	1	17
5	vtree_1_5_09.vt	vtree_1_5_10.vt	1	1	21

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
10	vtree_2_10_01.vt	vtree_2_10_02.vt	2	1	98
10	vtree_2_10_03.vt	vtree_2_10_04.vt	2	1	62
10	vtree_2_10_05.vt	vtree_2_10_06.vt	2	1	50
10	vtree_2_10_07.vt	vtree_2_10_08.vt	2	1	85
10	vtree_2_10_09.vt	vtree_2_10_10.vt	2	1	63
10	vtree_2_10_11.vt	vtree_2_10_12.vt	2	1	45
10	vtree_2_10_13.vt	vtree_2_10_14.vt	2	1	44
10	vtree_2_10_17.vt	vtree_2_10_18.vt	2	1	55
10	vtree_2_10_19.vt	vtree_2_10_20.vt	2	1	82
20	vtree_2_20_01.vt	vtree_2_20_02.vt	2	1	315
20	vtree_2_20_05.vt	vtree_2_20_06.vt	2	1	370
20	vtree_2_20_07.vt	vtree_2_20_08.vt	2	1	284
20	vtree_2_20_09.vt	vtree_2_20_10.vt	2	1	319
20	vtree_2_20_11.vt	vtree_2_20_12.vt	2	1	330
20	vtree_2_20_13.vt	vtree_2_20_14.vt	2	1	264
20	vtree_2_20_15.vt	vtree_2_20_16.vt	2	1	278
20	vtree_2_20_17.vt	vtree_2_20_18.vt	2	1	304
20	vtree_2_20_19.vt	vtree_2_20_20.vt	2	1	297
20	vtree_2_20_21.vt	vtree_2_20_22.vt	2	1	272
20	vtree_2_20_23.vt	vtree_2_20_24.vt	2	1	361
20	vtree_2_20_25.vt	vtree_2_20_26.vt	2	1	341
20	vtree_2_20_27.vt	vtree_2_20_28.vt	2	1	277
20	vtree_2_20_29.vt	vtree_2_20_30.vt	2	1	367
20	vtree_2_20_31.vt	vtree_2_20_32.vt	2	1	240
20	vtree_2_20_33.vt	vtree_2_20_34.vt	2	1	371
20	vtree_2_20_35.vt	vtree_2_20_36.vt	2	1	350
20	vtree_2_20_37.vt	vtree_2_20_38.vt	2	1	281
20	vtree_2_20_39.vt	vtree_2_20_40.vt	2	1	393
30	vtree_2_30_03.vt	vtree_2_30_04.vt	2	1	764
30	vtree_2_30_05.vt	vtree_2_30_06.vt	2	1	647
30	vtree_2_30_07.vt	vtree_2_30_08.vt	2	1	617
30	vtree_2_30_09.vt	vtree_2_30_10.vt	2	1	640
30	vtree_2_30_11.vt	vtree_2_30_12.vt	2	1	528
30	vtree_2_30_13.vt	vtree_2_30_14.vt	2	1	799
30	vtree_2_30_15.vt	vtree_2_30_16.vt	2	1	487
30	vtree_2_30_17.vt	vtree_2_30_18.vt	2	1	699
30	vtree_2_30_19.vt	vtree_2_30_20.vt	2	1	626
30	vtree_2_30_21.vt	vtree_2_30_22.vt	2	1	506
30	vtree_2_30_23.vt	vtree_2_30_24.vt	2	1	674
30	vtree_2_30_25.vt	vtree_2_30_26.vt	2	1	536
30	vtree_2_30_27.vt	vtree_2_30_28.vt	2	1	679
30	vtree_2_30_29.vt	vtree_2_30_30.vt	2	1	782

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_2_30_31.vt	vtree_2_30_32.vt	2	1	696
30	vtree_2_30_33.vt	vtree_2_30_34.vt	2	1	799
30	vtree_2_30_35.vt	vtree_2_30_36.vt	2	1	697
30	vtree_2_30_37.vt	vtree_2_30_38.vt	2	1	475
30	vtree_2_30_39.vt	vtree_2_30_40.vt	2	1	630
30	vtree_2_30_41.vt	vtree_2_30_42.vt	2	1	798
30	vtree_2_30_43.vt	vtree_2_30_44.vt	2	1	762
30	vtree_2_30_45.vt	vtree_2_30_46.vt	2	1	635
30	vtree_2_30_47.vt	vtree_2_30_48.vt	2	1	591
30	vtree_2_30_49.vt	vtree_2_30_50.vt	2	1	780
30	vtree_2_30_51.vt	vtree_2_30_52.vt	2	1	528
30	vtree_2_30_53.vt	vtree_2_30_54.vt	2	1	678
30	vtree_2_30_55.vt	vtree_2_30_56.vt	2	1	896
30	vtree_2_30_57.vt	vtree_2_30_58.vt	2	1	538
30	vtree_2_30_59.vt	vtree_2_30_60.vt	2	1	651
5	vtree_2_5_01.vt	vtree_2_5_02.vt	2	1	10
10	vtree_3_10_01.vt	vtree_3_10_02.vt	3	1	63
10	vtree_3_10_03.vt	vtree_3_10_04.vt	3	1	118
10	vtree_3_10_05.vt	vtree_3_10_06.vt	3	1	73
10	vtree_3_10_07.vt	vtree_3_10_08.vt	3	1	106
10	vtree_3_10_09.vt	vtree_3_10_10.vt	3	1	77
10	vtree_3_10_13.vt	vtree_3_10_14.vt	3	1	92
10	vtree_3_10_15.vt	vtree_3_10_16.vt	3	1	59
10	vtree_3_10_17.vt	vtree_3_10_18.vt	3	1	56
10	vtree_3_10_19.vt	vtree_3_10_20.vt	3	1	92
20	vtree_3_20_05.vt	vtree_3_20_06.vt	3	1	258
20	vtree_3_20_07.vt	vtree_3_20_08.vt	3	1	322
20	vtree_3_20_11.vt	vtree_3_20_12.vt	3	1	201
20	vtree_3_20_13.vt	vtree_3_20_14.vt	3	1	260
20	vtree_3_20_17.vt	vtree_3_20_18.vt	3	1	303
20	vtree_3_20_19.vt	vtree_3_20_20.vt	3	1	257
20	vtree_3_20_21.vt	vtree_3_20_22.vt	3	1	228
20	vtree_3_20_27.vt	vtree_3_20_28.vt	3	1	292
20	vtree_3_20_29.vt	vtree_3_20_30.vt	3	1	350
20	vtree_3_20_31.vt	vtree_3_20_32.vt	3	1	332
20	vtree_3_20_33.vt	vtree_3_20_34.vt	3	1	382
20	vtree_3_20_35.vt	vtree_3_20_36.vt	3	1	305
20	vtree_3_20_39.vt	vtree_3_20_40.vt	3	1	308

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_3_30_01.vt	vtree_3_30_02.vt	3	1	873
30	vtree_3_30_03.vt	vtree_3_30_04.vt	3	1	773
30	vtree_3_30_07.vt	vtree_3_30_08.vt	3	1	616
30	vtree_3_30_09.vt	vtree_3_30_10.vt	3	1	718
30	vtree_3_30_11.vt	vtree_3_30_12.vt	3	1	657
30	vtree_3_30_13.vt	vtree_3_30_14.vt	3	1	662
30	vtree_3_30_15.vt	vtree_3_30_16.vt	3	1	560
30	vtree_3_30_21.vt	vtree_3_30_22.vt	3	1	882
30	vtree_3_30_23.vt	vtree_3_30_24.vt	3	1	659
30	vtree_3_30_25.vt	vtree_3_30_26.vt	3	1	579
30	vtree_3_30_27.vt	vtree_3_30_28.vt	3	1	677
30	vtree_3_30_29.vt	vtree_3_30_30.vt	3	1	573
30	vtree_3_30_31.vt	vtree_3_30_32.vt	3	1	630
30	vtree_3_30_35.vt	vtree_3_30_36.vt	3	1	645
30	vtree_3_30_37.vt	vtree_3_30_38.vt	3	1	874
30	vtree_3_30_39.vt	vtree_3_30_40.vt	3	1	654
30	vtree_3_30_41.vt	vtree_3_30_42.vt	3	1	792
30	vtree_3_30_45.vt	vtree_3_30_46.vt	3	1	588
30	vtree_3_30_47.vt	vtree_3_30_48.vt	3	1	770
30	vtree_3_30_49.vt	vtree_3_30_50.vt	3	1	834
30	vtree_3_30_51.vt	vtree_3_30_52.vt	3	1	780
30	vtree_3_30_53.vt	vtree_3_30_54.vt	3	1	689
30	vtree_3_30_55.vt	vtree_3_30_56.vt	3	1	869
30	vtree_3_30_57.vt	vtree_3_30_58.vt	3	1	725
5	vtree_3_5_01.vt	vtree_3_5_02.vt	3	1	13
10	vtree_4_10_03.vt	vtree_4_10_04.vt	4	1	66
10	vtree_4_10_05.vt	vtree_4_10_06.vt	4	1	59
10	vtree_4_10_09.vt	vtree_4_10_10.vt	4	1	81
10	vtree_4_10_11.vt	vtree_4_10_12.vt	4	1	65
10	vtree_4_10_15.vt	vtree_4_10_16.vt	4	1	108
10	vtree_4_10_17.vt	vtree_4_10_18.vt	4	1	99
20	vtree_4_20_03.vt	vtree_4_20_04.vt	4	1	377
20	vtree_4_20_05.vt	vtree_4_20_06.vt	4	1	373
20	vtree_4_20_07.vt	vtree_4_20_08.vt	4	1	255
20	vtree_4_20_09.vt	vtree_4_20_10.vt	4	1	462
20	vtree_4_20_11.vt	vtree_4_20_12.vt	4	1	273
20	vtree_4_20_13.vt	vtree_4_20_14.vt	4	1	272
20	vtree_4_20_15.vt	vtree_4_20_16.vt	4	1	389
20	vtree_4_20_17.vt	vtree_4_20_18.vt	4	1	349
20	vtree_4_20_19.vt	vtree_4_20_20.vt	4	1	294

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
20	vtree_4_20_23.vt	vtree_4_20_24.vt	4	1	348
20	vtree_4_20_25.vt	vtree_4_20_26.vt	4	1	303
20	vtree_4_20_27.vt	vtree_4_20_28.vt	4	1	331
20	vtree_4_20_29.vt	vtree_4_20_30.vt	4	1	378
20	vtree_4_20_35.vt	vtree_4_20_36.vt	4	1	323
20	vtree_4_20_37.vt	vtree_4_20_38.vt	4	1	354
20	vtree_4_20_39.vt	vtree_4_20_40.vt	4	1	339
30	vtree_4_30_01.vt	vtree_4_30_02.vt	4	1	735
30	vtree_4_30_03.vt	vtree_4_30_04.vt	4	1	1003
30	vtree_4_30_05.vt	vtree_4_30_06.vt	4	1	690
30	vtree_4_30_07.vt	vtree_4_30_08.vt	4	1	657
30	vtree_4_30_09.vt	vtree_4_30_10.vt	4	1	760
30	vtree_4_30_11.vt	vtree_4_30_12.vt	4	1	793
30	vtree_4_30_13.vt	vtree_4_30_14.vt	4	1	729
30	vtree_4_30_15.vt	vtree_4_30_16.vt	4	1	580
30	vtree_4_30_17.vt	vtree_4_30_18.vt	4	1	772
30	vtree_4_30_19.vt	vtree_4_30_20.vt	4	1	626
30	vtree_4_30_21.vt	vtree_4_30_22.vt	4	1	833
30	vtree_4_30_25.vt	vtree_4_30_26.vt	4	1	552
30	vtree_4_30_31.vt	vtree_4_30_32.vt	4	1	680
30	vtree_4_30_35.vt	vtree_4_30_36.vt	4	1	765
30	vtree_4_30_37.vt	vtree_4_30_38.vt	4	1	745
30	vtree_4_30_39.vt	vtree_4_30_40.vt	4	1	792
30	vtree_4_30_43.vt	vtree_4_30_44.vt	4	1	699
30	vtree_4_30_45.vt	vtree_4_30_46.vt	4	1	467
30	vtree_4_30_47.vt	vtree_4_30_48.vt	4	1	564
30	vtree_4_30_49.vt	vtree_4_30_50.vt	4	1	708
30	vtree_4_30_53.vt	vtree_4_30_54.vt	4	1	781
30	vtree_4_30_55.vt	vtree_4_30_56.vt	4	1	634
30	vtree_4_30_57.vt	vtree_4_30_58.vt	4	1	652
30	vtree_4_30_59.vt	vtree_4_30_60.vt	4	1	577
5	vtree_4_5_01.vt	vtree_4_5_02.vt	4	1	17

Table 3: Output of Algorithm 1

A.1.2 Results of Algorithm 2

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
10	vtree_1_10_01.vt	vtree_1_10_02.vt	1	2	68
10	vtree_1_10_03.vt	vtree_1_10_04.vt	1	2	88
10	vtree_1_10_05.vt	vtree_1_10_06.vt	1	2	66
10	vtree_1_10_07.vt	vtree_1_10_08.vt	1	2	59
10	vtree_1_10_09.vt	vtree_1_10_10.vt	1	2	92
10	vtree_1_10_11.vt	vtree_1_10_12.vt	1	2	57
10	vtree_1_10_13.vt	vtree_1_10_14.vt	1	2	91
10	vtree_1_10_15.vt	vtree_1_10_16.vt	1	2	55
10	vtree_1_10_17.vt	vtree_1_10_18.vt	1	2	71
10	vtree_1_10_19.vt	vtree_1_10_20.vt	1	2	83
20	vtree_1_20_01.vt	vtree_1_20_02.vt	1	2	319
20	vtree_1_20_03.vt	vtree_1_20_04.vt	1	2	329
20	vtree_1_20_05.vt	vtree_1_20_06.vt	1	2	326
20	vtree_1_20_07.vt	vtree_1_20_08.vt	1	2	329
20	vtree_1_20_09.vt	vtree_1_20_10.vt	1	2	322
20	vtree_1_20_11.vt	vtree_1_20_12.vt	1	2	305
20	vtree_1_20_13.vt	vtree_1_20_14.vt	1	2	213
20	vtree_1_20_15.vt	vtree_1_20_16.vt	1	2	379
20	vtree_1_20_17.vt	vtree_1_20_18.vt	1	2	278
20	vtree_1_20_19.vt	vtree_1_20_20.vt	1	2	335
20	vtree_1_20_21.vt	vtree_1_20_22.vt	1	2	291
20	vtree_1_20_23.vt	vtree_1_20_24.vt	1	2	331
20	vtree_1_20_25.vt	vtree_1_20_26.vt	1	2	408
20	vtree_1_20_27.vt	vtree_1_20_28.vt	1	2	328
20	vtree_1_20_29.vt	vtree_1_20_30.vt	1	2	361
20	vtree_1_20_31.vt	vtree_1_20_32.vt	1	2	283
20	vtree_1_20_33.vt	vtree_1_20_34.vt	1	2	370
20	vtree_1_20_35.vt	vtree_1_20_36.vt	1	2	236
20	vtree_1_20_37.vt	vtree_1_20_38.vt	1	2	322
20	vtree_1_20_39.vt	vtree_1_20_40.vt	1	2	237
2	vtree_1_2_01.vt	vtree_1_2_02.vt	1	2	1

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_1_30_01.vt	vtree_1_30_02.vt	1	2	567
30	vtree_1_30_03.vt	vtree_1_30_04.vt	1	2	746
30	vtree_1_30_05.vt	vtree_1_30_06.vt	1	2	720
30	vtree_1_30_07.vt	vtree_1_30_08.vt	1	2	736
30	vtree_1_30_09.vt	vtree_1_30_10.vt	1	2	692
30	vtree_1_30_11.vt	vtree_1_30_12.vt	1	2	613
30	vtree_1_30_13.vt	vtree_1_30_14.vt	1	2	487
30	vtree_1_30_15.vt	vtree_1_30_16.vt	1	2	760
30	vtree_1_30_17.vt	vtree_1_30_18.vt	1	2	552
30	vtree_1_30_19.vt	vtree_1_30_20.vt	1	2	787
30	vtree_1_30_21.vt	vtree_1_30_22.vt	1	2	631
30	vtree_1_30_23.vt	vtree_1_30_24.vt	1	2	770
30	vtree_1_30_25.vt	vtree_1_30_26.vt	1	2	616
30	vtree_1_30_27.vt	vtree_1_30_28.vt	1	2	544
30	vtree_1_30_29.vt	vtree_1_30_30.vt	1	2	849
30	vtree_1_30_31.vt	vtree_1_30_32.vt	1	2	651
30	vtree_1_30_33.vt	vtree_1_30_34.vt	1	2	577
30	vtree_1_30_35.vt	vtree_1_30_36.vt	1	2	835
30	vtree_1_30_37.vt	vtree_1_30_38.vt	1	2	683
30	vtree_1_30_39.vt	vtree_1_30_40.vt	1	2	605
30	vtree_1_30_41.vt	vtree_1_30_42.vt	1	2	598
30	vtree_1_30_43.vt	vtree_1_30_44.vt	1	2	615
30	vtree_1_30_45.vt	vtree_1_30_46.vt	1	2	579
30	vtree_1_30_47.vt	vtree_1_30_48.vt	1	2	717
30	vtree_1_30_49.vt	vtree_1_30_50.vt	1	2	814
30	vtree_1_30_51.vt	vtree_1_30_52.vt	1	2	700
30	vtree_1_30_53.vt	vtree_1_30_54.vt	1	2	639
30	vtree_1_30_55.vt	vtree_1_30_56.vt	1	2	720
30	vtree_1_30_57.vt	vtree_1_30_58.vt	1	2	650
30	vtree_1_30_59.vt	vtree_1_30_60.vt	1	2	717
3	vtree_1_3_01.vt	vtree_1_3_02.vt	1	2	1
3	vtree_1_3_03.vt	vtree_1_3_04.vt	1	2	1
4	vtree_1_4_01.vt	vtree_1_4_02.vt	1	2	6
4	vtree_1_4_03.vt	vtree_1_4_04.vt	1	2	9
4	vtree_1_4_05.vt	vtree_1_4_06.vt	1	2	6
4	vtree_1_4_07.vt	vtree_1_4_08.vt	1	2	4
4	vtree_1_4_09.vt	vtree_1_4_10.vt	1	2	9
5	vtree_1_5_01.vt	vtree_1_5_02.vt	1	2	18
5	vtree_1_5_03.vt	vtree_1_5_04.vt	1	2	11
5	vtree_1_5_05.vt	vtree_1_5_06.vt	1	2	21
5	vtree_1_5_07.vt	vtree_1_5_08.vt	1	2	17
5	vtree_1_5_09.vt	vtree_1_5_10.vt	1	2	21

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
10	vtree_2_10_01.vt	vtree_2_10_02.vt	2	2	58
10	vtree_2_10_03.vt	vtree_2_10_04.vt	2	2	46
10	vtree_2_10_05.vt	vtree_2_10_06.vt	2	2	34
10	vtree_2_10_07.vt	vtree_2_10_08.vt	2	2	46
10	vtree_2_10_09.vt	vtree_2_10_10.vt	2	2	37
10	vtree_2_10_11.vt	vtree_2_10_12.vt	2	2	30
10	vtree_2_10_13.vt	vtree_2_10_14.vt	2	2	24
10	vtree_2_10_17.vt	vtree_2_10_18.vt	2	2	33
10	vtree_2_10_19.vt	vtree_2_10_20.vt	2	2	54
20	vtree_2_20_01.vt	vtree_2_20_02.vt	2	2	269
20	vtree_2_20_05.vt	vtree_2_20_06.vt	2	2	288
20	vtree_2_20_07.vt	vtree_2_20_08.vt	2	2	244
20	vtree_2_20_09.vt	vtree_2_20_10.vt	2	2	291
20	vtree_2_20_11.vt	vtree_2_20_12.vt	2	2	266
20	vtree_2_20_13.vt	vtree_2_20_14.vt	2	2	218
20	vtree_2_20_15.vt	vtree_2_20_16.vt	2	2	218
20	vtree_2_20_17.vt	vtree_2_20_18.vt	2	2	246
20	vtree_2_20_19.vt	vtree_2_20_20.vt	2	2	245
20	vtree_2_20_21.vt	vtree_2_20_22.vt	2	2	220
20	vtree_2_20_23.vt	vtree_2_20_24.vt	2	2	291
20	vtree_2_20_25.vt	vtree_2_20_26.vt	2	2	277
20	vtree_2_20_27.vt	vtree_2_20_28.vt	2	2	199
20	vtree_2_20_29.vt	vtree_2_20_30.vt	2	2	291
20	vtree_2_20_31.vt	vtree_2_20_32.vt	2	2	206
20	vtree_2_20_33.vt	vtree_2_20_34.vt	2	2	319
20	vtree_2_20_35.vt	vtree_2_20_36.vt	2	2	274
20	vtree_2_20_37.vt	vtree_2_20_38.vt	2	2	215
20	vtree_2_20_39.vt	vtree_2_20_40.vt	2	2	271
30	vtree_2_30_03.vt	vtree_2_30_04.vt	2	2	596
30	vtree_2_30_05.vt	vtree_2_30_06.vt	2	2	583
30	vtree_2_30_07.vt	vtree_2_30_08.vt	2	2	541
30	vtree_2_30_09.vt	vtree_2_30_10.vt	2	2	464
30	vtree_2_30_11.vt	vtree_2_30_12.vt	2	2	459
30	vtree_2_30_13.vt	vtree_2_30_14.vt	2	2	649
30	vtree_2_30_15.vt	vtree_2_30_16.vt	2	2	403
30	vtree_2_30_17.vt	vtree_2_30_18.vt	2	2	599
30	vtree_2_30_19.vt	vtree_2_30_20.vt	2	2	538
30	vtree_2_30_21.vt	vtree_2_30_22.vt	2	2	442
30	vtree_2_30_23.vt	vtree_2_30_24.vt	2	2	622
30	vtree_2_30_25.vt	vtree_2_30_26.vt	2	2	439
30	vtree_2_30_27.vt	vtree_2_30_28.vt	2	2	591
30	vtree_2_30_29.vt	vtree_2_30_30.vt	2	2	676

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_2_30_31.vt	vtree_2_30_32.vt	2	2	572
30	vtree_2_30_33.vt	vtree_2_30_34.vt	2	2	679
30	vtree_2_30_35.vt	vtree_2_30_36.vt	2	2	609
30	vtree_2_30_37.vt	vtree_2_30_38.vt	2	2	393
30	vtree_2_30_39.vt	vtree_2_30_40.vt	2	2	560
30	vtree_2_30_41.vt	vtree_2_30_42.vt	2	2	710
30	vtree_2_30_43.vt	vtree_2_30_44.vt	2	2	621
30	vtree_2_30_45.vt	vtree_2_30_46.vt	2	2	477
30	vtree_2_30_47.vt	vtree_2_30_48.vt	2	2	515
30	vtree_2_30_49.vt	vtree_2_30_50.vt	2	2	662
30	vtree_2_30_51.vt	vtree_2_30_52.vt	2	2	413
30	vtree_2_30_53.vt	vtree_2_30_54.vt	2	2	608
30	vtree_2_30_55.vt	vtree_2_30_56.vt	2	2	790
30	vtree_2_30_57.vt	vtree_2_30_58.vt	2	2	474
30	vtree_2_30_59.vt	vtree_2_30_60.vt	2	2	581
5	vtree_2_5_01.vt	vtree_2_5_02.vt	2	2	2
10	vtree_3_10_01.vt	vtree_3_10_02.vt	3	2	46
10	vtree_3_10_03.vt	vtree_3_10_04.vt	3	2	81
10	vtree_3_10_05.vt	vtree_3_10_06.vt	3	2	51
10	vtree_3_10_07.vt	vtree_3_10_08.vt	3	2	72
10	vtree_3_10_09.vt	vtree_3_10_10.vt	3	2	48
10	vtree_3_10_13.vt	vtree_3_10_14.vt	3	2	43
10	vtree_3_10_15.vt	vtree_3_10_16.vt	3	2	42
10	vtree_3_10_17.vt	vtree_3_10_18.vt	3	2	44
10	vtree_3_10_19.vt	vtree_3_10_20.vt	3	2	62
20	vtree_3_20_05.vt	vtree_3_20_06.vt	3	2	142
20	vtree_3_20_07.vt	vtree_3_20_08.vt	3	2	283
20	vtree_3_20_11.vt	vtree_3_20_12.vt	3	2	172
20	vtree_3_20_13.vt	vtree_3_20_14.vt	3	2	212
20	vtree_3_20_17.vt	vtree_3_20_18.vt	3	2	237
20	vtree_3_20_19.vt	vtree_3_20_20.vt	3	2	199
20	vtree_3_20_21.vt	vtree_3_20_22.vt	3	2	194
20	vtree_3_20_27.vt	vtree_3_20_28.vt	3	2	228
20	vtree_3_20_29.vt	vtree_3_20_30.vt	3	2	296
20	vtree_3_20_31.vt	vtree_3_20_32.vt	3	2	280
20	vtree_3_20_33.vt	vtree_3_20_34.vt	3	2	299
20	vtree_3_20_35.vt	vtree_3_20_36.vt	3	2	241
20	vtree_3_20_39.vt	vtree_3_20_40.vt	3	2	262

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
30	vtree_3_30_01.vt	vtree_3_30_02.vt	3	2	759
30	vtree_3_30_03.vt	vtree_3_30_04.vt	3	2	686
30	vtree_3_30_07.vt	vtree_3_30_08.vt	3	2	556
30	vtree_3_30_09.vt	vtree_3_30_10.vt	3	2	646
30	vtree_3_30_11.vt	vtree_3_30_12.vt	3	2	580
30	vtree_3_30_13.vt	vtree_3_30_14.vt	3	2	605
30	vtree_3_30_15.vt	vtree_3_30_16.vt	3	2	490
30	vtree_3_30_21.vt	vtree_3_30_22.vt	3	2	774
30	vtree_3_30_23.vt	vtree_3_30_24.vt	3	2	600
30	vtree_3_30_25.vt	vtree_3_30_26.vt	3	2	534
30	vtree_3_30_27.vt	vtree_3_30_28.vt	3	2	573
30	vtree_3_30_29.vt	vtree_3_30_30.vt	3	2	438
30	vtree_3_30_31.vt	vtree_3_30_32.vt	3	2	505
30	vtree_3_30_35.vt	vtree_3_30_36.vt	3	2	568
30	vtree_3_30_37.vt	vtree_3_30_38.vt	3	2	659
30	vtree_3_30_39.vt	vtree_3_30_40.vt	3	2	564
30	vtree_3_30_41.vt	vtree_3_30_42.vt	3	2	679
30	vtree_3_30_45.vt	vtree_3_30_46.vt	3	2	430
30	vtree_3_30_47.vt	vtree_3_30_48.vt	3	2	692
30	vtree_3_30_49.vt	vtree_3_30_50.vt	3	2	721
30	vtree_3_30_51.vt	vtree_3_30_52.vt	3	2	650
30	vtree_3_30_53.vt	vtree_3_30_54.vt	3	2	607
30	vtree_3_30_55.vt	vtree_3_30_56.vt	3	2	768
30	vtree_3_30_57.vt	vtree_3_30_58.vt	3	2	632
5	vtree_3_5_01.vt	vtree_3_5_02.vt	3	2	4
10	vtree_4_10_03.vt	vtree_4_10_04.vt	4	2	56
10	vtree_4_10_05.vt	vtree_4_10_06.vt	4	2	26
10	vtree_4_10_09.vt	vtree_4_10_10.vt	4	2	60
10	vtree_4_10_11.vt	vtree_4_10_12.vt	4	2	57
10	vtree_4_10_15.vt	vtree_4_10_16.vt	4	2	72
10	vtree_4_10_17.vt	vtree_4_10_18.vt	4	2	72
20	vtree_4_20_03.vt	vtree_4_20_04.vt	4	2	202
20	vtree_4_20_05.vt	vtree_4_20_06.vt	4	2	199
20	vtree_4_20_07.vt	vtree_4_20_08.vt	4	2	185
20	vtree_4_20_09.vt	vtree_4_20_10.vt	4	2	304
20	vtree_4_20_11.vt	vtree_4_20_12.vt	4	2	247
20	vtree_4_20_13.vt	vtree_4_20_14.vt	4	2	249
20	vtree_4_20_15.vt	vtree_4_20_16.vt	4	2	313
20	vtree_4_20_17.vt	vtree_4_20_18.vt	4	2	314
20	vtree_4_20_19.vt	vtree_4_20_20.vt	4	2	271

Tree size	Filename 1	Filename 2	Tree type	Algorithm	Number of steps
20	vtree_4_20_23.vt	vtree_4_20_24.vt	4	2	317
20	vtree_4_20_25.vt	vtree_4_20_26.vt	4	2	236
20	vtree_4_20_27.vt	vtree_4_20_28.vt	4	2	274
20	vtree_4_20_29.vt	vtree_4_20_30.vt	4	2	264
20	vtree_4_20_35.vt	vtree_4_20_36.vt	4	2	233
20	vtree_4_20_37.vt	vtree_4_20_38.vt	4	2	291
20	vtree_4_20_39.vt	vtree_4_20_40.vt	4	2	322
30	vtree_4_30_01.vt	vtree_4_30_02.vt	4	2	683
30	vtree_4_30_03.vt	vtree_4_30_04.vt	4	2	930
30	vtree_4_30_05.vt	vtree_4_30_06.vt	4	2	664
30	vtree_4_30_07.vt	vtree_4_30_08.vt	4	2	387
30	vtree_4_30_09.vt	vtree_4_30_10.vt	4	2	694
30	vtree_4_30_11.vt	vtree_4_30_12.vt	4	2	735
30	vtree_4_30_13.vt	vtree_4_30_14.vt	4	2	359
30	vtree_4_30_15.vt	vtree_4_30_16.vt	4	2	496
30	vtree_4_30_17.vt	vtree_4_30_18.vt	4	2	512
30	vtree_4_30_19.vt	vtree_4_30_20.vt	4	2	501
30	vtree_4_30_21.vt	vtree_4_30_22.vt	4	2	468
30	vtree_4_30_25.vt	vtree_4_30_26.vt	4	2	511
30	vtree_4_30_31.vt	vtree_4_30_32.vt	4	2	425
30	vtree_4_30_35.vt	vtree_4_30_36.vt	4	2	600
30	vtree_4_30_37.vt	vtree_4_30_38.vt	4	2	614
30	vtree_4_30_39.vt	vtree_4_30_40.vt	4	2	685
30	vtree_4_30_43.vt	vtree_4_30_44.vt	4	2	537
30	vtree_4_30_45.vt	vtree_4_30_46.vt	4	2	414
30	vtree_4_30_47.vt	vtree_4_30_48.vt	4	2	449
30	vtree_4_30_49.vt	vtree_4_30_50.vt	4	2	363
30	vtree_4_30_53.vt	vtree_4_30_54.vt	4	2	744
30	vtree_4_30_55.vt	vtree_4_30_56.vt	4	2	582
30	vtree_4_30_57.vt	vtree_4_30_58.vt	4	2	584
30	vtree_4_30_59.vt	vtree_4_30_60.vt	4	2	548
5	vtree_4_5_01.vt	vtree_4_5_02.vt	4	2	10

Table 4: Output of Algorithm 2

A.2 Average probability of shared subtree occurrence

Number of variables	Proportion of vtrees
1	0.0
2	0.0
3	0.3341
4	0.3372
5	0.3012
6	0.2747
7	0.244
8	0.2281
9	0.2036
10	0.1916
15	0.159
20	0.1487
25	0.1409
30	0.1396
35	0.1374
40	0.1309
45	0.1265
50	0.1325

Table 5: Proportion of vtrees that contains at least one shared subtree, by tree size