Stockholms universitet

# Tree-Editing via the $\ominus$-operator

Trädmodifiering med $\ominus$-operationen

Veronica Persson

Handledare: Marc Hellmuth
Examinator: Lars Arvestad
Inlämningsdatum: 31 May 2026



Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

**Abstract**

The aim of this paper is to construct an algorithm finding the fewest vertices we need to remove with the $\ominus$-operator in two directed rooted trees $T$ and $T'$, such that these trees become isomorphic. The $\ominus$-operator is a tool for removing vertices in directed acyclic graphs. Limitations and restrictions of this problem are discussed as well. The proposed method found to solve the stated problem is implemented in Python and it is an approximate algorithm, meaning it does not always in every case find an optimal solution. The algorithm is based on recursive subtree matching of tree structures combined with two different strategies for selecting which vertices to delete with the $\ominus$-operator. In relation to the Python code, a pseudocode for the most important parts of the program is provided with proofs of correctness, time complexity analysis and also discussion of technical details regarding the algorithm. It is also concluded by a closely examined existence analysis using star trees, that it always exists a solution to the problem for every pair of non-isomorphic trees. Finally we consider possible extensions of the presented approach, including generalizations to other types of graphs. As well potential improvements to the method in order to enhance its performance.

**Sammanfattning**

Syftet med denna uppsats är att konstruera en algoritm som hittar det minsta antalet noder som behöver tas bort med hjälp av $\ominus$-operatorn i två riktade rotade träd $T$ och $T'$, så att dessa träd blir isomorfiska. Denna $\ominus$-operator är en metod för att radera noder i riktade acykliska grafer. Uppsatsen behandlar även begränsningar och restriktioner av problemet. Den föreslagna metoden som har tagits fram för att lösa det givna problemet är implementerad i Python och detta är en approximativ algoritm, vilket betyder att den inte alltid i alla fall hittar en optimal lösning. Algoritmen bygger på rekursiv matchning av delträd i trädstrukturer, kombinerat med två olika strategier för att välja vilka noder som ska tas bort med $\ominus$-operatorn. Kopplat till Python koden presenteras en pseudokod för de viktigaste delarna av programmet tillsammans med korrekthetsbevis, tidskomplexitetsanalys, samt en diskussion av algoritmens tekniska detaljer. Genom en noggrann existensanalys baserat på stjärnträd visas det dessutom att det alltid existerar en lösning på problemet för varje par av icke-isomorfiska träd. Slutligen behandlas möjliga utvecklingsområden av det föreslagna tillvägagångssättet, inklusive generaliseringar till andra typer av grafer. Även potentiella förbättringar av själva algoritmen lyfts fram för att öka dess prestation.

# Contents

**1 Introduction** **4**

**2 Preliminaries** **6**
2.1 Graph basics . . . . . 6
2.1.1 Isomorphism . . . . . 8
2.2 The $\ominus$-operator . . . . . 8

**3 Results** **11**
3.1 Existence analysis . . . . . 11
3.2 Algorithm . . . . . 14
3.2.1 Structure of the implemented program . . . . . 16
3.2.2 Correctness proof . . . . . 20
3.2.3 Technical details . . . . . 22
3.2.4 Runtime . . . . . 23
3.3 Discussion . . . . . 23

**4 Summary and conclusion** **25**
4.1 AI Assistance . . . . . 26

**5 References** **27**

# 1 Introduction

Bioinformatics and computational biology has emerged and grown rapidly the last decades thanks to major advances in computational power and accessibility. The rapid progress the recent years in fields such as artificial intelligence, machine learning and deep learning has further boosted this area of research [6, 7]. The goal of bioinformatics and computational biology is to increase our understanding of health and genomic data. It plays an important role in the development and advances of personalized medicine, disease treatments and biological discoveries [7]. Thus new applications, technologies and methods for working with massive genomic datasets are constantly being developed and numerous softwares and programs are currently available for this purpose. An area of study within bioinformatics and computational biology is phylogenetic data. Phylogenetics studies the evolutionary relationships among organisms, based on their genetic information obtained through DNA and RNA sequencing. In the articles [5] and [6] tools for working with bioinformatic problems and phylogenetics are discussed. The first article is about the Biopython package, which is an open source international collaboration of volunteer developers, providing Python libraries for a wide range of bioinformatics problems. In the second article the Bio.Phylo toolkit is introduced. Bio.Phylo is included within Biopython and organizes phylogenetic trees as a primary data type, filling a previously underserved area of data handling within Biopython.

This thesis studies the $\ominus$-operator on directed rooted trees, which first was presented in [6] as a method called *collapse* in the Bio.Phylo package, but the $\ominus$-operator was first formally introduced in [2]. The $\ominus$-operator is a method for transforming DAGs, that mimics vertex suppression. In short we remove a vertex in a DAG and connect the children and the parents of this vertex.

The main purpose of this paper is to find an algorithm to determine the smallest sets of vertices in two directed rooted trees, $W$ in $T$ and $W'$ in $T'$, such that $T \ominus W$ and $T' \ominus W'$ become isomorphic. In relation to this we discuss if it is possible to find an exact algorithm or if a heuristic solution may be the best alternative, due to e.g the problems introduced by working with graph isomorphism and that one needs to set limitations and restrictions in order to avoid tackling too many problems at once, which could induce shallow, unclear and discontinuous results. We also discuss if a potential algorithm always will be able to provide a solution. That is, will it always for every instance of two non-isomorphic directed rooted trees $T$ and $T'$ exist vertices, which when deleted by the $\ominus$-operator results in two new isomorphic trees.

The structure of the thesis is as follows. Section 2 contains necessary definitions of graph theory and a thorough description of the $\ominus$-operator. In section 3 the results of the research topics are presented, which is divided into three parts. The first part addresses essential proofs. The second part analyzes an implemented algorithm in Python in relation to the central topics. A heuristic pseudocode is also provided for the main parts of the algorithm. In addition correctness proofs of the pseudocode and runtime complexity is briefly mentioned. In the last part the results of the algorithm are discussed. In section 4 the research topics and results are summarized. Future research and development possibilities are also addressed.

Python Code - `https://github.com/Croniii/BachelorThesis-ComputerSience`

# 2 Preliminaries

## 2.1 Graph basics

The definitions given in this section are based on [1] and [3].

**Definition 2.1** (*Directed Graph*)**:** A directed graph $G = (V, E)$ consists of a nonempty set of vertices $V(G) := V$ and an set of ordered edges $E(G) := E \subseteq (V \times V)$. We call the graph $G$ undirected if the edges in $E$ are unordered.

In a directed graph $G = (V, E)$ we define the in-degree of a vertex $v$ as $indeg(v) := |\{u \in V | (u, v) \in E\}|$ and the out-degree of a vertex $v$ as $outdeg(v) := |\{u \in V | (v, u) \in E\}|$.

A path in a graph refers to a sequence of distinct vertices $\langle v_0, v_1, ..., v_k \rangle$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, ..., k$. A cycle is a closed path, i.e a path with $v_0 = v_k$, meaning the path starts and ends at the same vertex, while ensuring no other vertices or edges are repeated. So an acyclic graph is a graph containing no cycles.

**Definition 2.2** (*Subgraph*)**:** The graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$.

**Definition 2.3** (*Connected Graph*)**:** If there exists a path between any two distinct vertices $u, v \in V$ in an undirected or directed graph $G = (V, E)$, the graph is said to be connected.

**Definition 2.4** (*Tree*)**:** An undirected or directed graph $T = (V, E)$ is a tree if and only if the graph is acyclic and connected.

**Lemma 2.1:** Definition 2.4 of a tree implicates the following: Any two distinct vertices $a, b$ in the tree $T$ are connected by exactly one unique path. A proof of this can be found on pages 1169-1170 in [3].

**Definition 2.5** (*DAG*)**:** A directed graph $G = (V, E)$ containing no cycles is called a directed acyclic graph or a DAG.

If $(v, u)$ is an edge in a DAG we say that $u$ is a child of $v$ and that $v$ is the parent of $u$. A leaf or external vertex is a vertex $v$ in a DAG without any children. We let the set of all leaves in a DAG be denoted by $L(G) \subseteq V(G)$. If we let $v, u \in V(G)$ and $v$ be the parent of $u$. Then the children vertices of $u$ are called grandchildren of $v$.

Any vertex $x$ in a DAG on a path from $v$ to $u$ is an ancestor of $u$ and so it follows if $x$ is ancestor of $u$, then $u$ is a descendant of $x$. We can define this relationship by $u \preceq_G x$.

**Definition 2.6** (*Directed Rooted Tree*)**:** If a directed tree $T = (V, E)$ has a special vertex $r \in V$ labelled as the "root" of the tree and from this node all other vertices $v \in V$ originate, so $indeg(r) = 0$, we say that $T$ is a directed rooted tree.

A directed rooted tree is essentially a DAG with the restriction that every node (except the root) has exactly one parent. From here on whenever a tree or rooted tree is mentioned it is assumed to take the form of a directed rooted tree.

**Definition 2.7** (*Subtree*)**:** The subtree $T'$ rooted at any vertex $r$ in a tree $T$, is the tree $T'$ derived by descendants of $r$, rooted at $r$.

**Definition 2.8** (*Star Tree*)**:** Let $S_k = (V, E)$ be a tree, where $k$ is the cardinality of $V$. We say that $S_k$ is a star tree of order $k$ if a single vertex $u \in V$ has $deg(u) = k - 1$ and the other $k - 1$ vertices $v \in V$ have $deg(v) = 1$. Star trees are also sometimes called star graphs. Figure 2.1 [4] displays different star trees.

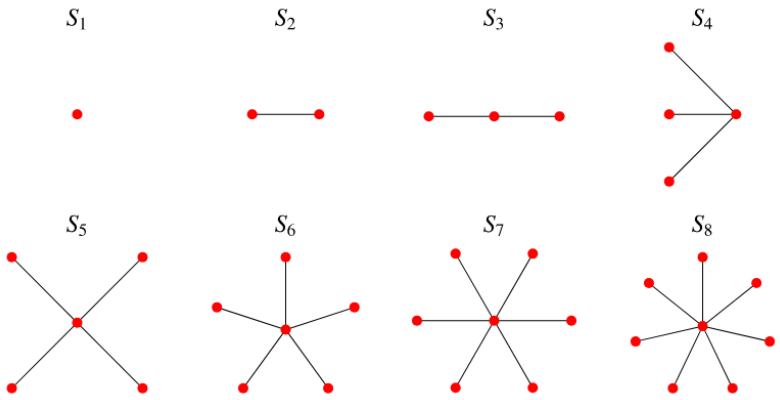



Figure 2.1: Eight star trees $S_k$ of order $k = 1, ..., 8$.

### 2.1.1 Isomorphism

**Definition 2.9** (*Isomorphism*)**:** Two directed graphs $G = (V, E)$ and $H = (V', E')$ are isomorphic if there exists a bijective map $\gamma : V \to V'$ such that $(u, v) \in E$ if and only if $(\gamma(u), \gamma(v)) \in E'$. Isomorphism between graphs is denoted by $G \cong H$.

The graph isomorphism problem is the problem of determining whether two finite graphs are isomorphic. The tree isomorphism problem is a restricted version of graph isomorphism problem, where the graphs are trees. Since trees have no cycles, the problem becomes much simpler than the general graph isomorphism problem.

The AHU (Aho, Hopcroft, Ullman) algorithm introduced in the 1970s is a well known algorithm for testing isomorphism of rooted trees and runs in time $O(n)$, where $n$ is equal to the number of vertices in a tree [8].

In this paper a variation of the AHU algorithm is implemented in the Python code which is based on [8, 9]. The runtime is $O(n \log n)$. This variant of AHU represents a tree using a unique string encoding. If the encoding-forms of two trees are identical, the trees are isomorphic.

## 2.2 The $\ominus$-operator

The $\ominus$-operator was introduced in the introduction. We now give a formal definition of this operator. Definitions and properties of the $\ominus$-operator are taken from [1] and [2].

**Definition 2.10** ($\ominus$-*operator*)**:** Let $G = (V, E)$ be a DAG and $v \in V$. Then $G \ominus v = (V', E')$ is the directed graph with the vertex set $V' = V \backslash \{v\}$ and edges $(p, q) \in E'$ precisely if $p, q \neq v$ and $(p, q) \in E$, or if $p, q \neq v$ and $(p, v) \in E$, $(v, q) \in E$.

In other words the graph $G \ominus v$ is obtained from $G = (V, E)$ by deleting $v$ and all its incident edges, while connecting each parent $p$ of $v$ with each child $q$ of $v$. If $v$ is a leaf or a root in $G$, then only $v$ together with all edges incident to $v$ are removed and no new edges are added. Thus no new vertices are introduced in $G \ominus v$ , i.e $V(G \ominus v) \subseteq V(G)$, but several new edges might be introduced in $G \ominus v$.

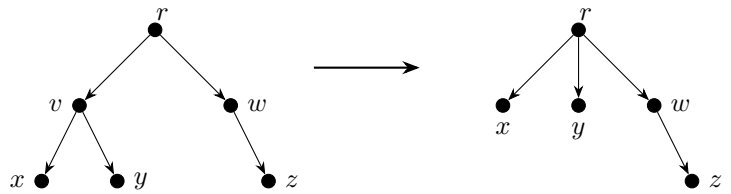
As stated in section 5 of [1] by construction, $G \ominus v$ preserves the ancestor relationship, that is $q \preceq_G p$ if and only if $q \preceq_{G \ominus v} p$ for all vertices $p, q \in V(G \ominus v)$ distinct from $v$. This implies that $G \ominus v$ remains a DAG.

Also as established in section 5 of [2] it's straightforward to check that $(G \ominus v) \ominus u = (G \ominus u) \ominus v$ holds for all $u, v \in V$ and $u \neq v$. Thus for a subset $W = \{w_1, w_2, ..., w_k\} \subset V$, we define the DAG $G \ominus W := (...((G \ominus w_1) \ominus w_2)...) \ominus w_k$.

We can now make the following observation:

**Observation 2.1:** If $G$ is a DAG and $W \subseteq V(G) \backslash L(G)$ is a non-empty subset. Then $G \ominus W$ is a DAG on $L(G)$, i.e leaves of $G$ remain leaves in $G \ominus W$ as long as the deleted vertices are not leaves.

We now consider two examples to further show how the collapse operator works. Looking at the first example in figure 2.2 the $\ominus$-operator is applied on the tree $T$ with the intention of deleting the inner vertex $v$. By the definition we then connect the children $x$ and $y$ of $v$ with the parent $r$ of $v$. On the right side of the root $r$ nothing changes.




$T$ $\qquad\qquad$ $T \ominus v$

Figure 2.2: Suppressing the vertex $v$ in $T$ with the $\ominus$-operator.

In the second example displayed in figure 2.3 we delete three vertices, $W = \{z, u, w\} \subset T(V)$ from the tree $T$ with the $\ominus$-operator, i.e $T \ominus \{z, u, w\}$. If we were to change the order in which we delete these vertices, the outcome would still be the same, as explained earlier.

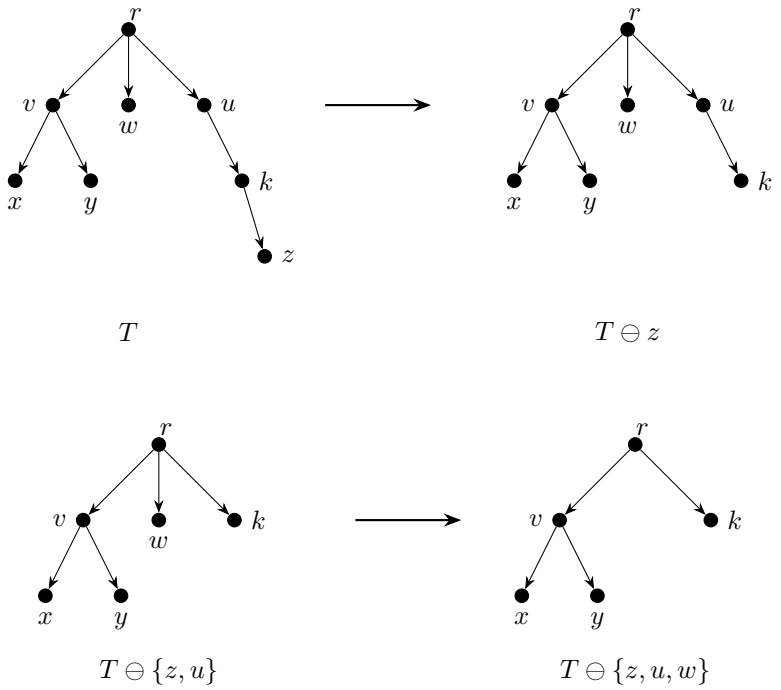



Figure 2.3: Suppressing the set of vertices $\{z, u, w\}$ of $T$ with the $\ominus$-operator.

# 3 Results

Here the results and discussions of the central questions presented in the introduction are provided. The first part of this section contains mathematical proofs of the existence of isomorphic transformations with the $\ominus$-operator. The second part explains and investigates a feasible heuristic algorithm whose goal is to find the smallest sets of vertices, $W \subset V(T)$ and $W' \subset V(T')$, such that $T \ominus W \cong T' \ominus W'$.

## 3.1 Existence analysis

A question we want to ask ourselves is: Will a possible algorithm always provide a solution given a correct input consisting of two rooted trees? That is, will it always exist some sets of vertices $W \subset V(T)$ and $W' \subset V(T')$ such that $T \ominus W$ and $T' \ominus W'$ become isomorphic. Or is it impossible sometimes to transform the trees to be isomorphic? This question is of importance, because it will affect how we construct, build and set up our algorithm. For example what kind of base cases and errors we have to consider and take into regard.

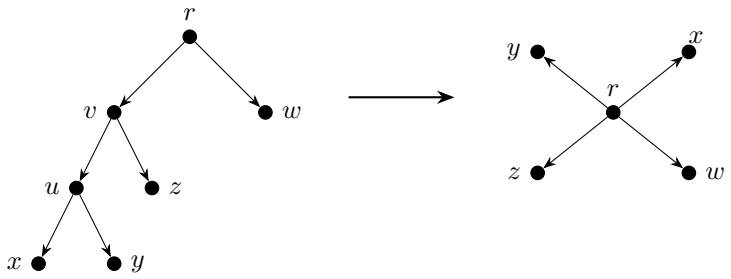
We start with the following lemma:

**Lemma 3.1:** Let $T$ be a tree with the root $r$. If we define the set $W$ as $W := V(T) \backslash (L(T) \cup \{r\})$, then $T \ominus W$ is a star tree.

*Proof:* Removing all vertices in $W$ from the tree $T$, means we remove all internal vertices in $T$ except the root $r$. So after this $V(T \ominus W)$ will only consist of the root $r$ and the leaves $L(T)$ from the original tree $T$. From lemma 2.1 in section 2.1 we know that every leaf $x \in L(T)$ is connected to $r$ by a unique path in $T$. All internal vertices on these paths are in $W$. Following the definition of the $\ominus$-operator (definition 2.10 in section 2.2), we have that each parent $p$ of the suppressed vertex $v$ is connected to each child $q$ of $v$. So removing all vertices in $W$ shortcuts the paths into single edges $(r, x)$. This implies each leaf $x \in L(T)$ will be connected to the root $r$ with a single unique edge $(r, x)$. Thus $T$ always collapses into a star tree $T \ominus W$ with center $r$ and outer vertices $L(T)$.

□

Below in figure 3.1 an example of how a star tree can be obtained is shown. Let $T = (V, E)$ be a tree with $V(T) = \{r, v, w, u, z, x, y\}$ and the set $W = \{u, v\}$ of internal vertices, except the root. After applying the collapse operator on the tree together with the set $W$, we clearly see that the tree collapses into a star tree $S_5$.




$T$ $\qquad\qquad$ $T \ominus \{v, u\}$

Figure 3.1: Transformation of $T$ into a star tree $S_5 = T \ominus \{v, u\}$

**Proposition 3.1:** Given two trees $T$ and $T'$, there always exists two sets $W \subset V(T)$ and $W' \subset V(T')$ such that

$$T \ominus W \cong T' \ominus W'.$$

*Proof:* We will divide this proof into two parts based on whether $T$ and $T'$ have the same number of leaves. Let's start with the case where both trees have the same number of leaves.

**|L(T) | = |L(T')|:** If both of the given trees $T$ and $T'$ already are star trees they automatically are isomorphic, since the trees have the same number of leaves and there is only one possible star graph for any fixed number of leaves. So in this scenario both sets $W$ and $W'$ are empty because nothing needs to be changed for the trees to be isomorphic.

Assuming neither of the given trees are star trees, one could just collapse both trees into star trees by setting $W = V(T)\backslash(L(T) \cup \{r\})$ and $W' = V(T')\backslash(L(T') \cup \{r'\})$ according to lemma 3.1. Seeing that no leaves are deleted during the $\ominus$-operation, we will work with the same leaf set all the time according to section 2.2 and since $|L(T)| = |L(T')|$, the star trees $T \ominus W$ and $T' \ominus W'$ obtained from the trees $T$ and $T'$ will also have the same number of leaves, which means they have an identical graph structure and therefore are isomorphic.

In the case only one of the trees already is a star tree, then one of the sets $W$ or $W'$ is going to be empty. We only need to transform the tree which is not a star tree into a star tree to make the trees isomorphic.

**|L(T)| $\neq$ |L(T')|:** Assuming neither of the given trees are star trees, we follow the exact same procedure as in the former case with an equal number of leaves. But because in this case the trees $T$ and $T'$ don't have the same number of leaves, the resulting star trees will also not have the same number of leaves, which means the number of vertices are going to differ and the star trees are not isomorphic. Then one of the star trees $T \ominus W$ or $T' \ominus W'$ must have more leaves than the other. Let's assume $L(T \ominus W) > L(T' \ominus W')$. Now we have to remove leaves from the star tree with a higher cardinality of leaves, i.e $T \ominus W$, until $L(T \ominus W) = L(T' \ominus W')$. So we will add vertices to $W$ and then eventually the star trees will become isomorphic. If instead $L(T' \ominus W') > L(T \ominus W)$ we proceed in the same way, but by adding vertices to $W'$ instead of $W$.

In the case both the given trees $T$ and $T'$ already are star trees, we can just do as in the second part of the text section above. In short we remove leaves from the star tree with a higher leaf cardinality until both star trees have the same number of leaves and therefore are isomorphic. Then one of the sets $W$ or $W'$ will be empty and the other will consist of leaves only.
If only one of the trees already is a star tree, we first have to transform the non-star

tree into a star tree and then carry out the same steps as in the case with both $T$ and $T'$ being star trees. Depending on which of $T$ and $T'$ has more leaves, either one of the sets $W$ or $W'$ is going to be empty, or none of the sets will be empty.

So there will always exist two sets $W$ and $W'$ (that can be empty) which together with the $\ominus$-operator will transform the trees $T$ and $T'$ so they eventually become isomorphic.

□

## 3.2 Algorithm

I have in Python created a program which approximates a solution to the problem of finding the smallest sets of vertices, $W \subset T$ and $W' \subset T'$, such that $T \ominus W \cong T' \ominus W'$. The entire code and some simulations can be found on Github - `https://github.com/Croniii/BachelorThesis-ComputerSience`.

To solve this problem the implemented algorithm needs to take two non-isomorphic rooted trees as input. Within the procedure of the algorithm with as few collapse operations as possible make these trees isomorphic. The output should hence be the two new modified isomorphic trees, a list containing all deleted vertices (i.e the sets $W$ and $W'$) and the total number of removed vertices.

My algorithm is heuristic and the idea is to make two trees isomorphic through recursive subtree-matching and collapse operations (applied when structural differences occur) performed during tree traversal. The sets $W$ and $W'$ are selected based on the amount of children and grandchildren tree nodes at the same depth in the two different trees have. Correctness of existence follows from proposition 3.1 stating two trees can always be transformed into isomorphic star trees using the collapse operator. The AHU algorithm (see section 2.1.1) is also implemented at the end of the procedure to verify that the code has worked correctly, meaning we check if the two modified trees returned really are isomorphic.

Now we continue with an visual illustration of how the program works on two trees to aid the reader in understanding the concept. The example in figure 3.2 is taken from simulation 2 in the testfile on Github. The upper part shows two non-isomorphic trees $T$ and $T'$ representing the input to the algorithm. The lower part displays the output from the algorithm, i.e the two modified trees $T \ominus \{y_3\}$ and $T' \ominus \{z_6\}$, which now are isomorphic after removing exactly two nodes with the $\ominus$-operator.

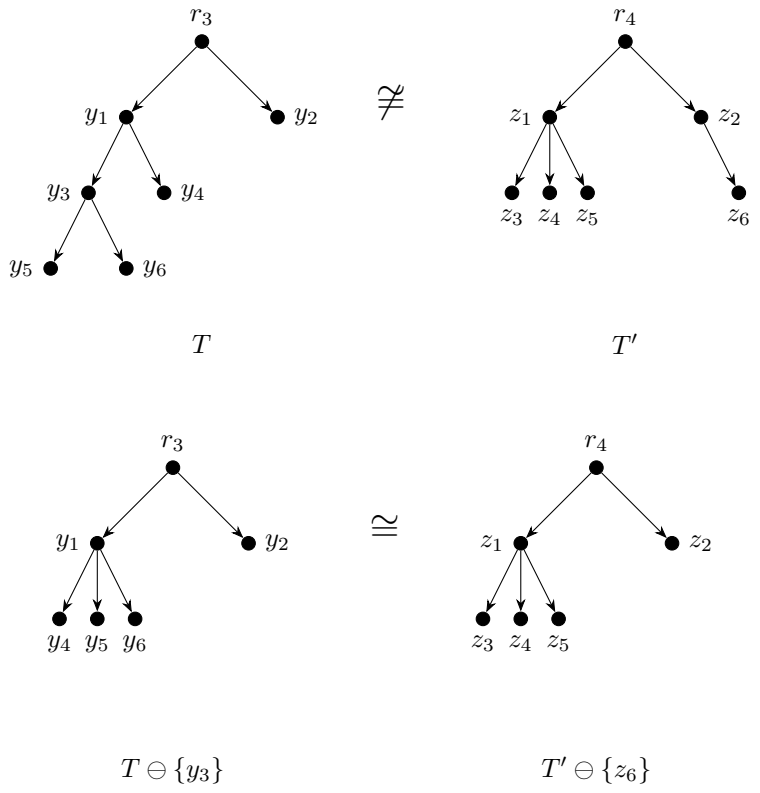



Figure 3.2: The implemented algorithm applied on two trees $T$ and $T'$

### 3.2.1 Structure of the implemented program

The program is object orientated and consists of four different classes. The first class *"TreeNode"* is used to build and implement a directed rooted tree. To build a tree we need to create *"TreeNode"* objects for all nodes which will be in the tree and connect them to each other with help of properties and functions of the class. So one single node will contain its only parent and a list of all its children and thus a node also refers to all its descendants. Hence the root node will contain the entire tree structure.

The second class *"EditNodes"* is responsible for transforming two non-isomorphic trees isomorphic and therefore contains all methods and attributes to do this. The last two classes *"MaxChildrenChooseVictimStrategy"* and *"SingleNodeChooseVictimStrategy"* implement two different kind of strategies for deciding which nodes should be deleted with the collapse operator. So these classes select which node we want to apply the $\ominus$-operator on. During execution only one of these can be used. Via the attribute *ChooseVictimStrategy* in *"EditNodes"* the user can select which strategy/class to apply.

There are five functions which make up the core functionality of the program and they all intersect which each other and work recursively. A pseudocode for each one of these functions is provided below. For further details see the code files on Github.

A list containing all the childnode objects of the node $v$ is denoted $children[v]$. $TotalChildren[v]$ refers to the total number of children of the node $v$. $IsomorphicNode$ is an attribute in the *"TreeNode"* class, which matches nodes across trees, i.e the attribute is used to keep track of nodes that have already been mapped togheter.

Algorithm $1-3$ are functions belonging to the *"EditNodes"* class. Algorithm 1 is called *MakeIsomorphic* and takes as input two *"TreeNode"*objects and returns nothing.

**Algorithm 1** : Makes two non-isomorphic trees isomorphic

```
1: function MakeIsomorphic(v, v')
2:     Connect v and v' via the attribute IsomorphicNode
3:     while TotalChildren[v] ≠ TotalChildren[v'] do
4:         Call the function ChooseVictim(v, v')
5:     if TotalChildren[v] ≠ 0 then
6:         Call the function MappingChildren(v, v')
7:     return                                              ▷
```

Algorithm 2 is called *MappingChildren* and takes two *"TreeNode"* objects as input and returns nothing. This function assumes both the input nodes have the same amount of children, which will always be the case because we only call upon *MappingChildren* from *MakeIsomorphic* after its while-loop has terminated. The while-loop only stops running if two nodes have the same number of children.

**Algorithm 2** : Decides which nodes should be mapped across two trees

1: **function** MAPPINGCHILDREN($v, v'$)
2: Create a empty list $grandchildrenList = [\quad]$
3: **for** all $u$ in copy(children[$v$]) **do**
4: **for** all $u'$ in copy(children[$v'$]) **do**
5: **if** TotalChildren[$u'$] = TotalChildren[$u$] and $u'.IsomorphicNode$ = None **then**
6: Add $u'$ to $grandchildrenList$
7: **if** $grandchildrenList$ not empty **then**
8: Call the function $MakeIsomorphic(u, grandchildrenList[0])$
9: Reset $grandchildrenList$ to an empty list.
10:
11: Create a list $childrenList1$ containing the childnodes $u$ of $v$, which have not yet been matched, i.e the $IsomorphicNode$ attribute is empty. Then sort the list in descending order based on TotalChildren[$u$].
12:
13: Create a list $childrenList2$ containing the childnodes $u'$ of $v'$, which have not yet been matched, i.e the $IsomorphicNode$ attribute is empty. Then sort the list in descending order based on TotalChildren[$u'$].
14: **if** $childrenList1$ not empty **then**
15: **for** ($i = 0, ...,$length of $childrenList1$) **do**
16: Call the function $MakeIsomorphic(childrenList1[i], childrenList2[i])$
17: **return** ▷

Algorithm 3 is called *ChooseVictim* and takes two *"TreeNode"* objects as input and returns nothing. The function uses the *"EditNodes"* class attribute $ChooseVictimStrategy$, which is set to one of the two classes *"MaxChildrenChooseVictimStrategy"* or *"SingleNodeChooseVictimStrategy"*. The $ChooseVictimStrategy$ attribute is set by the user in advance.

**Algorithm 3** : Chooses one node to be removed in a tree, i.e the $Victim$ and then modifies the tree using collapse to remove the $Victim$

**function** ChooseVictim($v, v'$)
    $Victim$=$ChooseVictimStrategy.Action1(v, v')$ or $Victim$=$ChooseVictimStrategy.Action2(v, v')$ depending on which class the user has chosen. ( $Action1$ and $Action2$ are functions given in the pseudocode below.)

    Call the function $Collapse(Victim)$. (This function can be found in the codefile "CodeThesis.py" on Github)

Algorithm 4 is called $Action1$ and is a function of the *"MaxChildrenChooseVictim-Strategy"* class. The function takes two *"TreeNode"* objects as input and returns one single *"TreeNode"* object".

**Algorithm 4** : Decides which node to apply collapse on

**function** Action1($v, v'$)
    $Victim$ = None
    $ParentVictim$ = None
    **if** TotalChildren[$v$] > TotalChildren[$v'$] **then**
        $ParentVictim = v$
    **else**
        $ParentVictim = v'$
    **for** all $u$ in children[$ParentVictim$] **do**
        **if** $Victim$= None **then**
            $Victim = u$
        **if** TotalChildren[$u$] < TotalChildren[$Victim$] **then**
            $Victim = u$
    **return** $Victim$

Algorithm 5 is called *Action*2 and is a function of the *"SingleNodeChooseVictimStrategy"* class. The function takes two *"TreeNode"* objects as input and returns one single *"TreeNode"* object.

**Algorithm 5** : Decides which node to apply collapse on

```
1:  function ACTION2(v, v')
2:      Victim = None
3:      ParentVictim = None
4:      if TotalChildren[v] < TotalChildren[v'] then
5:          ParentVictim = v
6:      else
7:          ParentVictim = v'
8:
9:      ChildDifference = |TotalChildren[v]−TotalChildren[v']|
10:     for all u in children[ParentVictim] do
11:         if TotalChildren[u] = ChildDifference +1 then
12:             Victim = u
13:         if Victim ≠ None then
14:             return Victim
15:
16:     Call upon Action1 from the class "MaxChildrenChooseVictimStrategy" and sets
        the output to Victim.
17:     return Victim
```

We end this subsection with another result from the simulations in the testfile given on Github. In figure 3.2 the *"SingleNodeChooseVictimStrategy"* class strategy was used in the simulation. In the next figure 3.3 the same simulation is shown, but for the *"MaxChildrenChooseVictimStrategy"* class. As we can see the output here is different from the output in figure 3.2. In total 4 nodes are removed with the $\ominus$-operator this time, resulting in an other set of isomorphic trees. This result is inferior because removing more nodes means we lose more information. Also in this case we notice, the algorithm has not solved the problem of removing as few vertices as possible to make the trees $T$ and $T'$ isomorphic. From the previous results, we know the same trees can be made isomorphic by removing just two vertices.

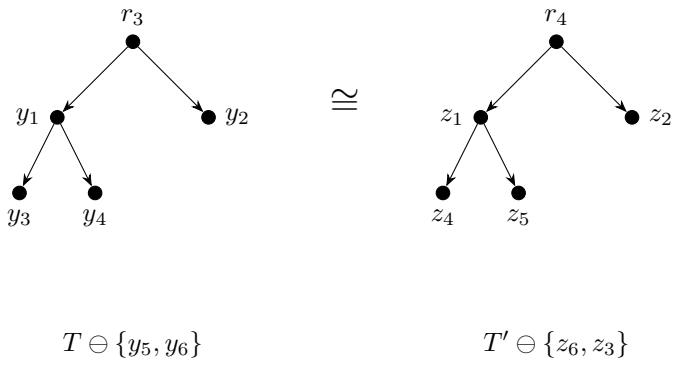



Figure 3.3: The implemented algorithm applied on two trees $T$ and $T'$ using the strategy "MaxChildrenChooseVictimStrategy"

### 3.2.2 Correctness proof

In this part proofs of correctness are given for all five core functions presented in the pseudocodes in section 3.2.1.

**Lemma 3.1:** When given two *"TreeNode"* objects $v, v'$ as input, *Action1* correctly returns one single *"TreeNode"*object called $Victim$ which should be a child of either $v$ or $v'$.

*Proof:* The if-else statement handles the cases when TotalChildren[$v$] > TotalChildren[$v'$] and when TotalChildren[$v$] < TotalChildren[$v'$]. Due to the fact that this algorithm only will be initialized by *MakeIsomorphic* via the while-loop condition, the case when $v$ and $v'$ have the same number of children will never occur. So the variable $ParentVictim$ correctly stores the node $v$ or $v'$, with the most children. This ensures the value of $ParentVictim$ never remains empty. The for-loop will always have at least one iteration because of the condition in the same while-loop as mentioned above in the function*MakeIsomorphic* and during the first iteration of the loop, the $Victim$ variable is set to the first child $u$ of $ParentVictim$ . The loop ends when all children $u$ of $ParentVictim$ have been processed. Then $Victim$ is the child $u$ with the min(TotalChildren[$u$]) among all children of $ParentVictim$. So *Action1* correctly returns a *"TreeNode"* object being a child of $v$ or $v'$.

**Lemma 3.2:** When given two *"TreeNode"* objects $v, v'$ as input, *Action2* correctly returns one single *"TreeNode"* object called $Victim$ which should be a child of either $v$ or $v'$.

*Proof: Action2* follows the same structure and steps as in *Action1* for the lines $1 - 7$, except the variable $ParentVictim$ correctly stores the node $v$ or $v'$, with minimum

children instead of the maximum. On line 10 the for-loop will run at least once, due to the same reasons as discussed in the proof of lemma 3.1. The loop searches among the children $u$ of $ParentVictim$ for a $u$ whose number of total children equals $|\text{TotalChildren}[v] - \text{TotalChildren}[v']| + 1$. If such a node exists the $Victim$ variable is set to this $u$ and the loop breaks and the function will stop and return $Victim$. Otherwise the loop ends when all children of $ParentVictim$ have been processed and the function then falls back onto *Action1*, which we know is correct. Thus *Action2* correctly returns a *"TreeNode"* object being a child of $v$ or $v'$.

**Lemma 3.3:** When given two *"TreeNode"* objects $v, v'$ as input, *ChooseVictim* correctly invokes the function $Collapse$ on one of the children of $v$ or $v'$ called $Victim$.

*Proof:* This function calls either *Action1* or *Action2* with the given inputs $v$ and $v'$ as parameters and saves the output from either of the algorithms in the variable $Victim$, which is a "TreeNode" object. Correctness proofs for these algorithms ensures $Victim$ will always be of correct form and thus the function $Collapse(Victim)$ can be invoked successfully with a valid parameter.

**Lemma 3.4:** When given two *"TreeNode"* objects $v, v'$ with exactly the same number of children as input, *MappingChildren* correctly traverses through the trees $v$, $v'$ and makes valid recursive calls to the function *MakeIsomorphic* mapping the nodes.

*Proof:* The first part of the algorithm begins with a nested for-loop and iterates over copies of all children of $v$ and $v'$, to avoid errors which may be caused otherwise when mutating a list we are iterating over. If there exists some grandchildren of $v'$ fulfilling the if-condition on line 5, these grandchildren are stored in a list called $grandchildrenList$. The if-condition ensures the elements in $grandchildrenList$ are unmapped and that only structurally compatible nodes are mapped later in the code. The second loop ends when all children of $v'$ have been processed. Now if $grandchildrenList$ is nonempty, *MakeIsomorphic* is correctly handled, since both input parameters are *"TreeNode"* objects obtained from the for-loops and the proof of lemma 3.5 ensures *MakeIsomorphic* works correctly. Regardless whether the if-condition on line 7 is satisfied or not, $granchildrenList$ is reset to be empty after the if-condition, ensuring no invalid mappings are created. Finally the first loop ends when all children of $v$ have been passed. In the next part, line $11-17$, remaining unmatched nodes are processed. The situation when we have unmatched nodes left, implicates the lists $childrenList1$ and $childrenList2$ are non-empty and a for-loop is executed. The for-loop iterates over all elements in $childrenList1$ and terminates since the list is finite. Inside the loop *MakeIsomorphic* is executed with parameters from the lists, which we know are *"TreeNode"* objects. These lists will always have the same length (arising from the input conditions and the first part of the algorithm), so indexing is safe and pairing is well defined. Thus line 16 is handled correctly, such that larger subtrees are matched first minimizing structural mismatch. Consequenlty loops and recursion are over finite structures, so *MappingChildren* properly traverses through the trees $v$ and $v'$ and successfully recursively calls *MakeIsomorphic*.

**Lemma 3.5:** When given two *"TreeNode"* objects $v, v'$ as input, *MakeIsomorphic* correctly invokes *MappingChildren* and *ChooseVictim*, which ensures *MakeIsomorphic* constructs a proper isomorphism between the trees $v$ and $v'$.

*Proof:* The algorithm begins with correctly mapping the nodes $v$ and $v'$. At the beginning of two trees, this guarantees that the root correspondence is properly established. The while-loop runs as long as the number of children of $v$ and $v'$ differs. Inside the loop, $ChooseVictim(v, v')$ is correctly invoked since its parameters is the original inputs $v$ and $v'$. *ChooseVictim* collapses nodes to reduce the children of the input nodes and therefore the loop will eventually terminate when $v$ and $v'$ have the same number of children. Thus *MappingChildren* will be invoked successfully in the if-statement. When $v, v'$ are leafs the function stops. So *MakeIsomorphic*is correct with respect to tree isomorphism under the correctly defined *MappingChildren* and *ChooseVictim* functions.

Regarding the lemmas $3.3 - 3.5$ the recursion always terminates because the trees are finite and each recursive call will work on a strictly smaller structure.

### 3.2.3 Technical details

To reduce the complexity of the problem and focus on the structural aspects of tree isomorphism, some assumptions and restrictions have been made. One specification is that I have not used the Bio.Phylo package mentioned in the introduction for the $\ominus$-operator. Instead I have made a simplified implementation of the collapse method, customized for only directed rooted trees (since that is what we are working with) and fitted to work with the functions and classes in the code.

Another practical detail is that two resulting trees transformed by the program, may still be printed with different child/subtree ordering despite being structurally isomorphic. Since the order of children/subtrees does not affect tree isomorphism, two equivalent trees may appear visually different when printed. This is therefore only a representation issue and does not affect the correctness of the algorithm. Its easy to verify this by flipping the subtrees or children and it is also verified by the AHU algorithm like mentioned earlier.

Also an very important detail is that the order of which the children are presented in the *"TreeNode"* objects, will influence the solution. So two simulations with exactly the same trees, but with different implemented order of their children/subtrees, may result in different isomorphic trees. This aspect originates in the *MappingChildren* function.

### 3.2.4 Runtime

We will shortly mention some aspects regarding the runtime. The implemented program has runtime $O(n^2)$ in worst case, based on the pseudocode. Here $n$ denotes the total number of vertices in the larger of the two input trees. This runtime can be derived from the *MakeIsomorphic* function, where the dominant cost arises from *MappingChildren*. The most expensive part in *MappingChildren* is the nested for-loop iterating over the children of the input nodes.

When starting the process of making two trees isomorphic, the implemented program as a first step checks if the two given trees already are isomorphic with the AHU algorithm. We do this since invoking the other parts of the program, which means we traverse through two entire trees, when nothing is to be changed or removed anyway, will only result in needless iterations. So the best scenario in the program would be two isomorphic trees as input, because then only the AHU algorithm is initialized and the other parts are never executed. The AHU algorithm has time complexity $O(n \log n)$, which means this is the best possibly runtime achievable.

If we have two non-isomorphic trees the code will always need to traverse through the entire tree structures of these two trees in order to track nodes across the trees and find out where the trees differs, to now which vertices to remove. This regardless how similar the trees are. So the runtime cannot really be improved in this part, i.e in the *MappingChildren* function. What affects the runtime is how many nodes one needs to remove. The ideal in the case of non-isomorphic trees would be if we just have to remove a single node. In the worst case we have two completely different trees. Then the algorithm needs to make many deletions with the collapse operator, which will raise the runtime of the program.

Also the *"SingleNodeChooseVictimStrategy"* class will probably take more time to run, because if this strategy fails, it will default back onto the *"MaxChildrenChooseVictimStrategy"* strategy.

## 3.3 Discussion

From the results presented in this section we can make some observations. In figure 3.2 and 3.3 we notice that for two identical pairs of non-isomorphic trees, the resulting isomorphic trees are different depending on which strategy class ( *"MaxChildrenChooseVictimStrategy"* or *"SingleNodeChooseVictimStrategy"*) is used. The same pattern follows through all the other simulations in the testfile in Python. Sometimes the first strategy works better and at other times the second strategy, depending on how the tree structure looks like and how the trees are is implemented.

In the code simulations we observe sometimes one of the sets $W$ or $W'$ are empty, when the given trees are quite similar. Usually both of the sets contain at least one node. In the case the given trees already are isomorphic both sets will be empty and then there is no point in making $\ominus$-operations, since no vertices needs to be removed, which the AHU establishes.

The star tree solution given in section 3.1 was at first considered as solution to the central question stated in the introduction and as Python implementation, but this method completely ignores all internal tree structure. The approach is not very good if we care about preserving evolutionary or hierarchical information. In very few cases the star tree solution removes the fewest vertices possible. One realizes often it removes much more vertices than necessary.

# 4 Summary and conclusion

The conclusion which can be drawn based on the results is that we could not find an exact algorithm determining the smallest sets of vertices in two directed rooted trees, $W$ in $T$ and $W'$ in $T'$, such that $T \ominus W$ and $T' \ominus W'$ became isomorphic. However an heuristic algorithm was found. This means the implemented algorithm is not an exact or maybe the most optimal solution to the problem. At each step, the algorithm makes local decisions regarding which vertices to collapse based on predefined strategies (*"MaxChildrenChooseVictimStrategy"* and *"SingleNodeChooseVictimStrategy"*). Since these decisions are irreversible and no backtracking is performed, the algorithm cannot guarantee that the resulting sets $W$ and $W'$ are minimal in all cases. In contrast we confirmed with help of startrees, that it will always be possible to transform two non-isomorphic trees isomorphic with the $\ominus$-operator.

So based on this we notice there are areas of improvement. One upgrade would be to analyze both the input trees before applying the algorithms turning the trees isomorphic. Right now local decisions are made regarding which vertices to collapse during the tree traversal. If we build up an overview of how the trees look before starting to modify them it becomes easier to determine which modifications are necessary. We can first identify the nodes/subtrees being identical across the two trees and ignore them in the *MakeIsomorphic* algorithm. The program could then focus on finding the best possible ways to collapse nodes to minimize the loss and preserve as much of the original tree structure as possible.
Also we realize using the same strategy to choose which nodes to remove throughout the entire process of making two trees isomorphic is not the most optimal idea. Combining multiple strategies would probably be the best alternative. To find the most optimal strategy in each deletion case, a potential approach could be the earlier discussed procedure of analyzing trees in advance.

This study contributes to the progress of the Bio.Phylo package and the extension of the $\ominus$-operator, which adds to the research of bioinformatics and computational biology. The current work can be further developed in multiple directions. Such as not restricting ourselves to directed rooted trees. Instead possibly evolve the study to directed graphs, undirected graphs or DAGs. There is also opportunities for extension of the algorithm itself, in different ways to improve the simulation results.

## 4.1 AI Assistance

The AI tool *ChatGPT (OpenAI)* has been used in this thesis for support with grammar and text structure. Also to increase understanding of materials and concepts. In the coding part in Python, AI aided with explanations of implemented methods, such as the *filter()* function and different copying mechanisms in Python. Additionally it has been used to a small extent for detecting errors in the written code.

AI has not been used generate code, write text or in some other ways solve the central questions. All the solutions and work in this project have been made by the author with help from supervisor Marc Hellmuth and by discussions with course mates and friends. All content has been reviewed and revised by the author.

# 5 References


[1] Lindeberg, A., Hellmuth, M. *"Simplifying and Characterizing DAGs and Phylogenetic Networks via Least Common Ancestor Constraints"*. Bull Math Biol 87, 44 (2025). https://doi.org/10.1007/s11538-025-01419-z

[2] Shanavas, A.V., Changat, M., Hellmuth, M., Stadler, P.F. (2024). *"Unique Least Common Ancestors and Clusters in Directed Acyclic Graphs"*. In: Kalyanasundaram, S., Maheshwari, A. (eds) *Algorithms and Discrete Applied Mathematics*. CALDAM 2024. Lecture Notes in Computer Science, vol 14508. Springer, Cham. https://doi.org/10.1007/978-3-031-52213-0_11

[3] Thomas H. Cormen, Charles E.Leserson, Ronald L. Rivest, Clifford Stein, *"Introduction to Algortihms"*, The MIT Press, ISBN 9780262046305, (2022)

[4] Weisstein, Eric.W. *"Star Graph."* From MathWorld – A Wolfram resource. Retrieved from https://mathworld.wolfram.com/StarGraph.html (Accessed: 07. May. 2026)

[5] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, Michiel J. L. de Hoon, *"Biopython: freely available Python tools for computational molecular biology and bioinformatics"*, Bioinformatics, Volume 25, Issue 11, June 2009, Pages 1422–1423, https://doi.org/10.1093/bioinformatics/btp163

[6] Talevich, E., Invergo, B.M., Cock, P.J. et al. *"Bio.Phylo: A unified toolkit for processing, analyzing and visualizing phylogenetic trees in Biopython"*. BMC Bioinformatics 13, 209 (2012).https://doi.org/10.1186/1471-2105-13-209

[7] Si, Tong, and Haijun Gong. 2026. *"Challenges and Advances in Bioinformatics and Computational Biology"* Current Issues in Molecular Biology 48, no. 2: 185. https://doi.org/10.3390/cimb48020185

[8] Florain Ingels. 2023. *"Revisiting Tree Isomorphism: An Algorithm Bric-à-Brac"*. ⟨hal-04232137v2⟩. pages 1-7.

[9] Campbell, Douglas M., and David Radford. 1991. *"Tree Isomorphism Algorihtms: Speed vs. Clarity"*. Mathematics Magazine, vol. 64, no. 4, pp. 252-61. JSTOR, https://doi.org/10.2307/2690833

Datalogi
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm