Treewidth based algorithms for Tree Containment in phylogenetics

by

Robbert V. Huijsman

To obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday January 31, 2023 at 10:00 AM.

Student number:	4490797	
Project Duration:	April, 2022 - Januari, 2023	
Faculty:	Faculty of Applied Mathematics, Delft	
Thesis committee:	Dr.ir. L.J.J. van Iersel	TU Delft, supervisor
	Dr. M.E.L. Jones Jones	TU Delft, supervisor
	Prof.Dr. M.M. de Weerdt	TU Delft

TU Delft

1 Abstract

TreeContainment is a well-known problem within phylogenetics, which asks whether a binary phylogenetic tree is embedded in a binary phylogenetic network. For this problem, Jones, Weller and van Iersel (2022) [1] have created an algorithm that uses dynamic programming on tree-decompositions to achieve a running time that is exponential in the tree-width parameter instead of in the number of reticulations. However, due to the implicit formulations of two crucial steps in this algorithm, this running time cannot be achieved in practice without finding ways to generate the required structures explicitly. This paper provides two new sub-algorithms that can do that. To further improve the performance of the algorithm, I introduce a number of criteria and other methods that can be used to reduce the number of structures generated. Additionally, I describe a way to manipulate the nice tree-decompositions to create a more favorable order of bags for the dynamic programming. These sub-algorithms and improvements are used by two new algorithms twitch and pitch, whose implementations are compared to a brute force algorithm and a new branching cherry-picking algorithm named botch. The latter has an FPT running time that is exponential in the number of vertices that have only reticulations as children. The comparisons show that the implementations of the dynamic programming algorithms twitch and pitch are slower in practice than the brute force algorithm, despite their numerous improvements. Of the four new implementations, botch has the best test results and it is shown to be fast in practice.

2 Introduction

FPT (fixed parameter tractable) algorithms have a time complexity where all exponential terms are dependant on a certain parameter. Hence, if this parameter is constant, the time complexity is no longer exponential but instead polynomial. A popular parameter for FPT algorithms is the *treewidth* of a graph. This parameter shows how similar the graph is to a tree and can be very small for graphs that are close to trees.

Phylogenic trees and networks are used to represent the evolutionary history of DNA. TreeContainment is a well-known problem within phylogenetics, which asks whether a phylogenetic tree is embedded in a phylogenetic network. This problem can arise when a newly generated network representing the evolutionary history of genomes needs to be confirmed using existing phylogenetic trees. The former can be a network instead of a tree (for which the problem would be easy) due to genetic hybridization events that frequently occur in for example fungi or flora. These hybridization events are represented using reticulations, which are vertices that have multiple incoming arcs but only one outgoing arc.

This problem is closely related to NetworkContainment and HybridizationNumber. The former asks whether an input network is contained within another input network, while the latter has a set of input trees and asks whether there exists a network with at most k reticulations that contains them. Notably, verifying a solution of a HybridizationNumber instance can be done by solving multiple TreeContainment instances. Development of new methods for TreeContainment may in the long term contribute to the development of new methods for these harder problems, for which no FPT algorithms with reasonable dependence on the parameter exist.

2.1 Related literature

Kanj et al. (2008) [2] have shown that the TreeContainment problem is unfortunately NP-hard for general binary phylogenetic networks. One approach to deal with this is by trying to find specific restrictive types of input networks that do permit polynomial-time algorithms. The following are some examples of network types for which this approach has been successful: Tree-child networks, where Janssen and Murakami (2021) [3] have created an algorithm with linear time complexity, which can even solve the more general problem NetworkContainment. Genetically stable networks (which includes tree-child networks), where Gambette et al (2016) [4] have created an algorithm with a quadratic time complexity. Nearly stable networks and reticulation visible networks, where Weller (2017) [5] has created an algorithm with a linear time complexity.

A different approach is to try to find an FPT algorithm for the general problem where the parameter in which it is exponential is limited. Some examples of this approach are as follows: Kanj et al. (2008) [2] have created an algorithm with a time complexity of $\mathcal{O}(2^{\frac{k}{2}}|N|^2)$, where k is the number of reticulations.

For large values of k, this is an improvement over the well-known brute force algorithm (also shown in section 9.1), which has a $\mathcal{O}(2^k|N|)$ time complexity. Weller (2017) [5] further improved this to a running time of $\mathcal{O}(3^{t^*}|N|^2)$ where t^* is the maximum number of "unstable component-roots". Note that while the base of the exponent is worse, t^* is considered a much smaller exponent. Finally, Jones et al (2022) [1] have created a new dynamic programming algorithm that this paper is largely based on. It solves TreeContainment in $(2^{\mathcal{O}(t^2)} \cdot |N|)$ time complexity, where t is the tree-width of N.

2.2 Motivation

The aforementioned dynamic programming algorithm from Jones et al (2022) [1] is interesting for multiple reasons. First of all, its running time is exponential in the tree-width which has some useful properties. For example, Kelk et al (2018) [6] have shown that the square root of the *level* (largest number of reticulations in a biconnected component) is an upper bound for the tree-width. Seeing the effects of this running time in practice is quite interesting, especially given the lack of other dynamic programming algorithm implementations on tree decompositions in phylogenetics. Furthermore, the popularity of tree-width based fixed parameter tractable algorithms outside of phylogenetics has supported the development and accessibility of algorithms that can find tree-decompositions. One of these is the well-known minimum degree heuristic mentioned by Bodlaender (1993) [7] that is used in one of the implementations in this paper. Lastly, Jones et al have given an implicit description in some parts of the algorithm, which does not guarantee the possibility of an implementation. Two steps in the algorithm in particular require generating all structures for which certain rules hold. However, the approach of generating all possible structures and removing those that do not follow the specified rules for this would violate the theoretical running time. Hence, creating the implementation also proves the practical feasibility of this algorithm, which might encourage the creation of similar algorithms for other problems in phylogenetics.

2.3 My contributions

In this paper I introduce two new explicitly stated sub-algorithms that can replace the aforementioned implicit steps from the algorithm from Jones et al (2022) [1], which makes the implementation of their dynamic programming concept feasible. These sub-algorithms are used in two new algorithms twitch and pitch, for which I have also created a series of improvements that increase their speed. Additionally, this paper introduces a new branching cherry-picking algorithm botch that can solve TreeContainment in $\mathcal{O}(2^t \cdot |N|)$ time, where t is the number of nodes that have only reticulations as children. I test implementations of these algorithms against a brute force algorithm to see their running times in practice. From these tests, I conclude that the botch algorithm is quite fast in practice and that the twitch and pitch algorithms are quite slow in practice, despite their theoretical FPT running times.

3 Preliminaries

An overview of all definitions in this paper is shown in Figure 1. This section contains all green and blue colored definitions from it.

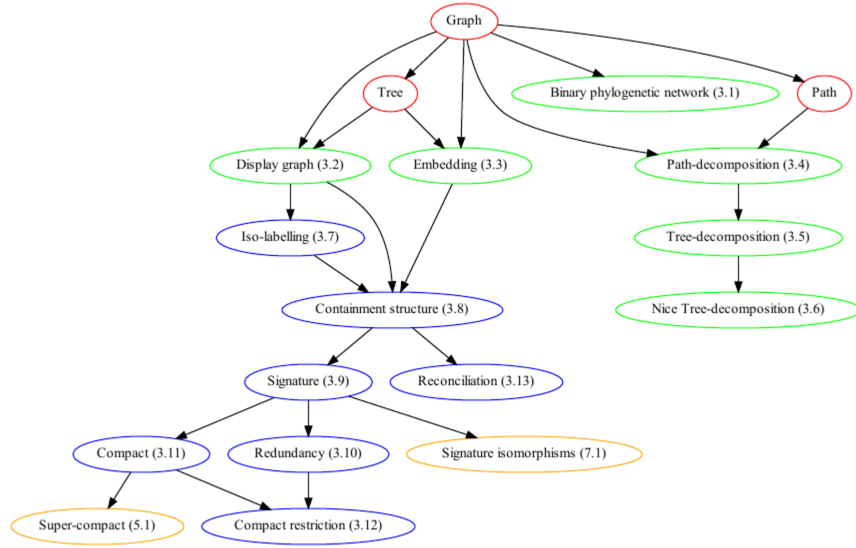


Figure 1: Definitions colored red are well known and expected to be familiar to the reader. Definitions colored green are less well known and described in this section. Definitions colored blue are algorithm-specific and are introduced by Jones et al. (2022) [1]. Though the formulations in this paper may differ slightly, they are functionally equivalent aside from iso-labellings. Definitions colored orange are new and defined in this paper where needed. An arrow from a definition a to a definition b shows that b requires prior knowledge of a. The numbers next to the definition names match the numbers of the definitions in the paper.

Definition 3.1. A *phylogenetic network* is a directed acyclic graph where every node either has indegree 1 or outdegree 1 but not both. A phylogenetic network is *binary* if its nodes have indegree and outdegree ≤ 2. The nodes are referred to by their following types:

1. Nodes with indegree 0 and outdegree 1 are *roots*. In a phylogenetic network, there may only be one of these.

2. Nodes with indegree 1 and outdegree 0 are *leaves*.

3. Nodes with indegree 1 and outdegree 2 are *tree nodes*.

4. Nodes with indegree 2 and outdegree 1 are *reticulation nodes*.

Additionally, a *phylogenetic tree* is a phylogenetic network without reticulation nodes.

In the rest of this paper, all trees and networks are assumed to be binary phylogenetic trees and binary phylogenetic networks respectively. Nodes and vertices may be used interchangeably.

Definition 3.2. Given a network N and a tree T on the same set of leaves, a *display graph* $D(N, T)$ is a graph that is the union of all nodes and arcs from both N and T.

Images of display graphs in this paper will always have the network drawn on top, the leaves drawn in the middle and the tree drawn on the bottom. Additionally, leaves are always drawn as squares, whereas non-leaves are drawn as ellipses. See Figure 14 for an example of this.

Definition 3.3. An *embedding* is a function $\delta : T \to N$ that maps every vertex from T to a vertex of N and every arc of T to a path in N with the following properties:

1. If $v \in T \cap N$, $\delta(v) = v$.
2. If $uv \in A(T)$, then $\delta(uv)$ must be an path from $\delta(u)$ to $\delta(v)$ in N. Furthermore, the paths $\delta(uv)$ for $uv \in A(T)$ must be arc-disjoint.

In the rest of this paper, a node $u \in N$ that is in $\delta(V(T))$ is often called an *embedding node*, while the corresponding tree node $\delta^{-1}(u)$ is sometimes referred to as the *embedded node*. Similarly, arcs in $\delta(A(T))$ are referred to as *embedding arcs* with corresponding *embedded arcs*.

Definition 3.4. Let N be a network. A *path decomposition* $X(N)$ of N is a path whose nodes, referred to as *bags*, are sets $S_i \subseteq V(N)$ for $i \in \{1, 2, \ldots, |X(N)|\}$. These bags have the following conditions:

1. For every node $a \in V(N)$, there is an i such that $a \in S_i$.
2. For every arc $ab \in A(N)$, there is an i such that $a, b \in S_i$.
3. For any vertex $a \in V(N)$, the set of all bags that contain it must form a path.

The number of elements in the largest bag of the decomposition minus 1 is called the *width*. The minimal width value of a path decomposition for a given network N is referred to as the *pathwidth* or $pw(N)$. Subtracting 1 is done to ensure that the pathwidth of a path is always 1.

The following is a definition of a similar decomposition where the path is replaced by a tree.

Definition 3.5. Let N be a network. A *tree decomposition* $X(N)$ of N is a tree whose nodes, referred to as *bags*, are sets $S_i \subseteq V(N)$ for $i \in \{1, 2, \ldots |X(N)|\}$. These bags have the following conditions:

1. For every node $a \in V(N)$, there is an i such that $a \in S_i$.
2. For every arc $ab \in A(N)$, there is an i such that $a, b \in S_i$.
3. For any vertex $a \in V(N)$, the set of all bags that contain it must form a tree.

Again, the number of elements in the largest bag of the decomposition minus 1 is called the *width* and the minimal width value of a tree decomposition for a given network N is referred to as the *treewidth* or $tw(N)$. Subtracting 1 ensures that the treewidth of a tree is always 1.

The following definition can apply to either of the two aforementioned decompositions:

Definition 3.6. A decomposition $X(N)$ on a network N with bags S_i for $i \in \{1, 2, \ldots |X(N)|\}$ is called *nice* if its bags are each of one of the following types:

1. Leaf bags S_a where $S_a = \emptyset$
2. Introduce bags S_a with 1 child bag S_b where $a \in V(S)$ and $S_a = S_b \cup a$
3. Forget bags S_a with 1 child bag S_b where $a \in V(S)$ and $S_a = S_b \setminus a$
4. Join bags S_a with 2 child bags S_b and S_c where $S_a = S_b = S_c$.

3.1 Algorithm specific preliminaries

Definition 3.7. Given a bag S and a set of labels Y, an *iso-labelling* is a function $\iota : V(D(N,T)) \to S \cup Y$ where the co-domain value that is assigned to the node is called the *iso-label*. It is bijective on S in the sense that no two nodes in $D(N,T)$ can be given the same labelling from S and every label from S should be given to some node in $D(N,T)$. However, any number of nodes may be given labellings from Y. Aditionally, nodes with an iso-label from Y may only be adjacent nodes with the same iso-label or nodes with an iso-label from S.

The goal of the iso-labelling is to present which nodes in $D(N,T)$ are corresponding to a node in the current bag. The iso-label FUTURE indicates that the node's correspondance to a node in $D_{IN}(N_{IN}, T_{IN})$ is currently unknown, but will be decided in the future. The iso-label PAST indicates that the node previously had a correspondance to a node in $D_{IN}(N_{IN}, T_{IN})$, which has now been forgotten. To match the naming of FUTURE and PAST iso-labels, S is often referred to as the present. For ease of reading, the symbol ι (with varying subscripts) is used for all iso-labellings in the rest of this paper.

Definition 3.8. An $(S, Y)-$containment structure is a tuple $(D(N,T), \delta, \iota)$ consisting of a display graph $D(N,T)$, an embedding δ on that display graph and an iso-labeling ι with label set Y and present set S.

Definition 3.9. A *signature* is an $(S, \{FUTURE, PAST\})$-containment structure.

Again for ease of reading, the symbol σ (with varying subscripts) is used for all signatures in the rest of this paper.

Definition 3.10. Arcs and nodes can become *redundant* in the following ways:

- A non-embedding arc $ab \in N$ is redundant if $\iota(a) = \iota(b)$.
- An arc $ab \in T$ and all arcs in its embedding path $\delta(ab)$ are redundant if $\iota(a) = \iota(b) = \iota(a')\ \forall a' \in \delta(ab)$.
- A non-embedding node $a \in N$ is redundant if all its arcs are redundant.
- A node $a \in T$ and its embedding $\delta(a)$ are redundant if $\iota(a) = \iota(\delta(a))$ and they both only have redundant arcs.

Redundancy is used to limit the size of display graphs in signatures.

Definition 3.11. A signature is *compact* if it has no network node $a \in N$ that has all of the following properties:

1. a is not an embedding node
2. a has indegree 1 and outdegree 1
3. $\iota(a) \in \{\text{PAST}, \text{FUTURE}\}$
4. $\iota(b) \in \{\text{PAST}, \text{FUTURE}\}$ for any node b adjacent to a

A signature that is not compact can be turned into a compact signature by *compacting* all nodes that violate the compactness definition. This is done by removing the node and replacing any arc of it with a new arc from its parent to its child in N and in any embedding path in δ (if applicable).

The opposite procedure is sometimes referred to as *de-compacting* a network arc $ab \in N$. This is done by inserting a node c in between it, which requires replacing the arc ab with the arcs ac and cb in N and in any embedding path in δ.

Definition 3.12. A *compact restriction* is a procedure taken on a signature. A compact $\{X \to Y\}$-restriction changes the iso-labelling of all nodes iso-labelled X into the iso-label Y. Subsequently, all nodes and arcs that have become redundant during this procedure are removed. Lastly, the signature is made compact by compacting all nodes that violate the compactness properties.

Due to the extensive use of restrictions in some proofs of this paper, the alternate shorter notation $\phi_{\{X \to Y\}}(\sigma)$ is often used to denote a $\{X \to Y\}$-restriction on signature σ.

Definition 3.13. Let X be a *Join bag* with children Y_L and Y_R and let S be the set of present nodes in X. Then a $(S, \{\text{LEFT, RIGHT, FUTURE}\})$-containment structure is referred to as a *reconciliation* for σ.

3.2 Theoretical algorithm

Before heading into the tree-decomposition algorithms developed and implemented for this paper, it is important to mention the theoretical algorithm created by van Iersel et al. (2022) [1] on which they are based. For lack of a better name, this will be referred to as THE THEORETICAL ALGORITHM in the rest of this paper. The pseudo code, in which CV denotes the collection of signatures in a bag, is as follows:

Algorithm 1 THE THEORETICAL ALGORITHM

1: Compute $tw(N_{IN})$ and $tw(D_{IN}(N_{IN}, T_{IN}))$
2: **if** $tw(D_{IN}(N_{IN}, T_{IN})) > 2tw(N_{IN}) + 1$ **then return** $False$
3: Compute a nice tree-decomposition $\mathcal{T}$ of $D_{IN}(N_{IN}, T_{IN})$
4: **for** bag $X = (P, S, F) \in \mathcal{T}$ in a bottom-up traversal **do**
5:     **if** X is a *Leaf bag* **then**
6:         $CV_X \leftarrow \{\text{compact signature } \sigma \mid \iota^{-1}(PAST) = \emptyset\}$
7:     **if** X is an *Forget bag* with child bag $Y = X \cup \{z\}$ **then**
8:         $CV_X \leftarrow \{\phi_{\{z \to \text{PAST}\}}(\sigma) \mid \sigma \in CV_Y\}$
9:     **if** X is a *Introduce bag* with child bag $Y = X \setminus \{z\}$ **then**
10:         $CV_X \leftarrow \{\text{compact signature } \sigma \mid \phi_{\{z \to \text{PAST}\}}(\sigma) \in CV_Y\}$
11:     **if** X is a *Join bag* with children Y_L and Y_R **then**
12:         **for** each compact reconciliation μ for X **do**
13:             $\sigma_L \leftarrow \phi_{\{\text{LEFT} \to \text{PAST}, \text{RIGHT} \to \text{FUTURE}\}}(\mu)$
14:             $\sigma_R \leftarrow \phi_{\{\text{RIGHT} \to \text{PAST}, \text{LEFT} \to \text{FUTURE}\}}(\mu)$
15:             **if** $\sigma_L \in CV_{Y_L}$ and $\sigma_R \in CV_{Y_R}$ **then**
16:                 Add $\phi_{\{\text{LEFT}, \text{RIGHT} \to \text{PAST}\}}(\mu)$ to CV_X
17:     **if** X is the *Root bag* **then**
18:         **if** there is a $\sigma \in X$ such that $\iota^{-1}(\text{FUTURE}) = \emptyset$ **then return** $True$
    return $False$

The first two lines of this pseudocode are based on the following theorem by Janssen et al (2018) [8]:

Theorem 1. *Let N be an unrooted binary phylogenetic network and let T be an unrooted binary phylogenetic tree on the same set of leaves as N. If there exists an embedding of T in N then $tw(D_{IN}(N_{IN}, T_{IN})) \leq 2tw(N_{IN}) + 1$*

For the way this theorem is used in the algorithm's check, note that a rooted tree cannot be embedded in a rooted network if the unrooted form of the tree cannot be embedded in the unrooted form of the network.

The rough idea behind the algorithm is to construct an embedding of T_{IN} in N_{IN} by navigating through $D_{IN}(N_{IN}, T_{IN})$ using a tree-decomposition. Each bag contains signatures which maintain constructed parts of the input graphs with the corresponding constructed embeddings. The properties of tree-decompositions ensure that the algorithm can add a node, add all its neighbors and then permanently forget the node again. In this way, the algorithm tries out every

embedding part combination through the node. When more nodes are added, some of the attempted embedding part combinations will prove impossible and can be discontinued, while new combinations are made using the successful embedding parts. This is an advantage over the brute force algorithm shown in Section 9.1, which can only find out that a embedding part does not work when it has tried every embedding of T_{IN} in N_{IN} that includes it. Additionally, The theoretical algorithm uses the FUTURE iso-label to delay decisions until more information is known about the remainder of $D(N_{IN}, T_{IN})$. An example of this is when a node from T_{IN} is added with a corresponding embedding node, the embedding node will have the iso-labelling FUTURE. This prevents the algorithm from immediately having to decide which network node it should be embedded to. Some examples of this algorithm's procedures on forget bags, introduce bags and join bags are shown with images in Section 3.3 A full example of this algorithm from start to finish is shown in the appendix in Section B.

3.3 Examples

To display signatures in images, it is useful to introduce some consistent ways of showing certain attributes. In the rest of this paper, nodes colored blue have the FUTURE iso-label, nodes colored green have a present iso-label that is stated in between round brackets and nodes colored red have a PAST iso-label. Leaves are represented using rectangles, while other nodes are shown as ellipses. Arcs displayed with solid arrowheads show that the arc is either an embedded arc or an embedding arc. Arcs displayed with hollow arrowheads show that the arc is not part of any embedding path. Embeddings are usually given by showing a mapping of the nodes, since the combination of the node mapping and the arrowheads is sufficient to determine all embedding paths (as shown in Section 7.1).

Within this section, Figure 2 shows two examples of forget node steps, Figure 3 shows two examples of introduce node steps and Figure 4 shows an example of a join node step.

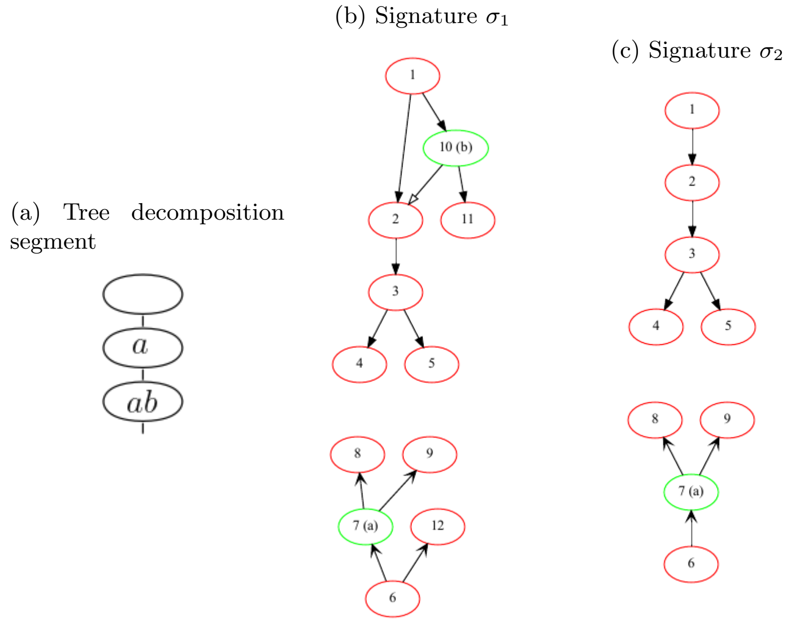


Figure 2: Examples of two signatures that undergo the forget node steps in the tree-decomposition shown in image (a). The empty bag is the root and the other bags are its child and grandchild. The signature σ_1 shown in image (b) belongs to the bag containing the nodes a and b and the signature σ_2 shown in image (c) belongs to the bag containing only node a. The node embedding δ_1 of σ_1 is $\{6 \to 1,\ 7 \to 3,\ 8 \to 4,\ 9 \to 5,\ 12 \to 11\}$ and the node embedding δ_2 of σ_2 is the same without the node 12. Note that $\phi_{\{b \to \mathrm{PAST}\}}(\sigma_1) = \sigma_2$ and $\phi_{\{a \to \mathrm{PAST}\}}(\sigma_2) = \sigma_\emptyset$ where $\sigma_\emptyset$ is the empty signature (with empty functions for its iso-labelling and embedding). In the former compact restriction, the arcs $(1, 10)$, $(10, 11)$ and $(10, 2)$ become redundant when node 10 gets the PAST iso-label. In turn, the nodes 10, 11 and 12 also become redundant. The latter restriction causes all arcs and nodes to become redundant once node 7 is assigned the PAST iso-label.

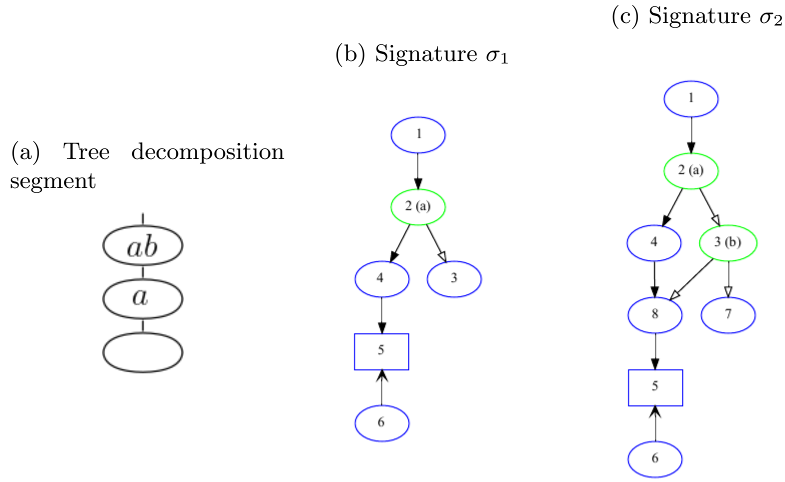
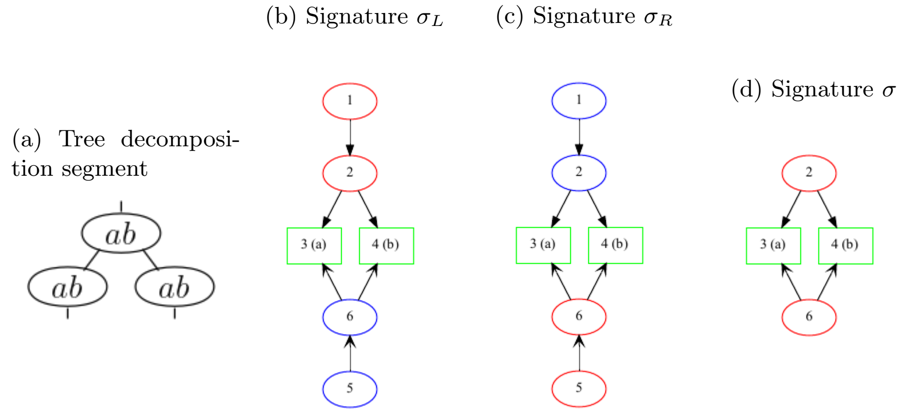


Figure 3: Examples of two signatures created by the introduce node steps on the tree-decomposition shown in image (a). The empty bag is the leaf and the other bags are its parent and grandparent. The signature σ_1 shown in image (b) belongs to the bag containing only node a and the signature σ_2 shown in image (c) belongs to the bag containing the nodes a and b. The node embedding δ_1 of σ_1 is $\{6 \to 1,\ 5 \to 5\}$ and the node embedding δ_2 of σ_2 is the same. Note that $\phi_{\{a \to \text{FUTURE}\}}(\sigma_1) = \sigma_\emptyset$ where $\sigma_\emptyset$ is the empty signature (with empty functions for its iso-labelling and embedding) and $\phi_{\{b \to \text{FUTURE}\}}(\sigma_2) = \sigma_1$. The former is due to all arcs and nodes in σ_1 becoming redundant once the node 2 gets a FUTURE iso-label. For the latter, note arcs $(3, 8)$ and $(3, 7)$ become redundant once 3 gets the FUTURE iso-label. This results in node 7 becoming redundant and node 8 violating the compactness properties.



(a) Tree decomposition segment

(b) Signature σ_L

(c) Signature σ_R

(d) Signature σ

Figure 4: Example of a join node signature creation. Image (a) shows a segment of a tree-decomposition where the top-most bag is a join bag. The signature σ_L from image (b) is contained in the left child bag and σ_R from image (c) is contained in the right child bag. These signatures have an identical embedding $\delta_L = \delta_R = \{5 \to 1,\ 6 \to 2,\ 3 \to 3,\ 4 \to 4\}$. The reconciliation μ in this case is identical to σ_L or σ_R with the following isolabelling: $\iota_\mu = \{1, 2 \to \text{LEFT},\ 5, 6 \to \text{RIGHT},\ 3 \to a,\ 4 \to b\}$. Note that $\phi_{\{\text{RIGHT}\to\text{PAST},\text{LEFT}\to\text{PAST}\}}(\mu) = \sigma$, which is shown in image (d). The nodes 1 and 5 are removed due to redundancy.

4 Implemented decomposition algorithms

The first algorithm introduced in this paper is referred to as TWITCH, which is an acronym for "tree-width implementation of TreeContainment by Huijsman". This algorithm is very similar to THE THEORETICAL ALGORITHM in its structure, but has some major differences in the procedures for introduce bags and join bags. The key idea is that TWITCH explicitly constructs a set of signatures for a bag based on the set of signatures of its children bags, whereas THE THEORETICAL ALGORITHM constructs the same set by first generating all possible signatures and then throwing away those that do not follow certain rules.

Let CV_X denote the collection of signatures for bag X and let $\sigma_\emptyset = (\emptyset, \phi_\emptyset, \iota_\emptyset)$ be an "empty" signature where $\phi_\emptyset$ and $\iota_\emptyset$ are empty functions. Then the pseudo code of TWITCH is as follows:

Algorithm 2 TWITCH

```
 1: Compute a nice tree-decomposition 𝒯 of D_IN(N_IN, T_IN)
 2: for bag X = (P, S, F) ∈ 𝒯 in a bottom-up traversal do
 3:     if X is a Leaf bag then
 4:         CV_X ← {σ_∅}
 5:     if X is an Forget bag with child bag Y = X ∪ {z} then
 6:         CV_X ← {φ_{z→PAST}(σ) | σ ∈ CV_Y}
 7:     if X is a Introduce bag with child bag Y = X \ {z} then
 8:         CV_X ← ⋃_{σ∈CV_Y} explicit_introduce_node(σ, z)
 9:     if X is a Join bag with children Y_L and Y_R then
10:         CV_X ← ⋃_{σ_L∈CV_L, σ_R∈CV_R} explicit_join_node(σ_L, σ_R)
11:     if X is the Root bag then
12:         if σ_∅ ∈ CV_X then return True
    return False
```

Note that the seemingly different requirements in leaf bags between TWITCH and THE THEORETICAL ALGORITHM do not make a difference. Signatures in leaf bags cannot have present or PAST iso-labelled nodes since there have not been any nodes introduced yet, and any signature with only FUTURE iso-labelled nodes will have all of its arcs and vertices be redundant. Similarly, signatures in root bags cannot have present iso-labelled nodes and thus if it also has no FUTURE iso-labelled nodes, any PAST iso-labelled nodes would become redundant.

The $\phi_{\{z \to \text{FUTURE}\}}$ can only affect the node z, its neighbors and its embedding/embedded arcs (the inverse for introducing nodes, as shown in the proof of Lemma 3). Hence, $\phi_{\{z \to \text{FUTURE}\}}(\sigma)$ can be generated by taking σ and checking the *redundancy* and *compact* definitions locally instead of checking them in the entire signature. This allows it to run with a constant running time. The in-depth workings of this are explained in Appendix C.

The two functions *explicit_introduce_node* and *explicit_join_node* generate sets of signatures; their workings are more advanced and are explained in Section

5 and Section 6 respectively.

The second algorithm introduced in this paper is referred to as PITCH, which is an acronym for "path-width implementation of TreeContainment by Huijsman". This algorithm differs from TWITCH by using a path-decomposition instead of tree-decomposition. This allows for the usage of "super-compactness" when *introducing* nodes, which is explained in Section 5.2 and Section 6.3. Since path-decompositions have no join bags, PITCH does not have a join node step. The pseudo code is as follows:

Algorithm 3 PITCH

1: Compute a nice path-decomposition $\mathcal{P}$ of $D_{IN}(N_{IN}, T_{IN})$
2: **for** bag $X = (P, S, F) \in \mathcal{P}$ in a bottom-up traversal **do**
3:     **if** X is a *Leaf bag* **then**
4:         $CV_X \leftarrow \{\sigma_\emptyset\}$
5:     **if** X is an *Forget bag* with child bag $Y = X \cup \{z\}$ **then**
6:         $CV_X \leftarrow \{\phi_{\{z \to \text{PAST}\}}(\sigma) \mid \sigma \in CV_Y\}$
7:     **if** X is a *Introduce bag* with child bag $Y = X \setminus \{z\}$ **then**
8:         $CV_X \leftarrow \bigcup_{\sigma \in CV_Y} super_compact_explicit_introduce_node(\sigma, z)$
9:     **if** X is the *Root bag* **then**
10:         **if** $\sigma_\emptyset \in CV_X$ **then return** $True$
return $False$

Note that the first two lines of pseudo code in THE THEORETICAL ALGORITHM are omitted in TWITCH and PITCH. These lines made sure that the algorithm immediately returned *False* if $tw(D_{IN}(N_{IN}, T_{IN})) > 2tw(N_{IN})$. The reason this check is omitted is that it requires an exact calculation of $tw(D_{IN}(N_{IN}, T_{IN}))$, instead of the tree-decomposition that can be generated by a heuristical approach. The most accessible implementation for this was the treewidth function from SageMath [9]. Unfortunately, testing during development showed that this check rarely returned *False* while simultaneously taking more time than the PITCH algorithm itself. Hence, even though it is required for the theoretical time complexity result for THE THEORETICAL ALGORITHM, the check is not used in the implementations introduced in this paper.

This paper does not contain a centralized proof of correctness for TWITCH and PITCH. Instead, it proves that the explicit functions used in TWITCH and PITCH (which replace the implicit procedures in THE THEORETICAL ALGORITHM) are sufficient to generate all signatures that are required for the proof that van Iersel et al. (2022) [1] have created for THE THEORETICAL ALGORITHM.

5 Introduce node

When introducing new nodes, THE THEORETICAL ALGORITHM from van Iersel et al. (2022) [1] adds all signatures to the new bag for which there exists a certain restriction in the child bag. This could theoretically be done by going through all possible signatures of the required type and checking whether there is a compact-$\{z \rightarrow \text{FUTURE}\}$-restriction of it in the child bag. van Iersel et al. (2022) [1] have shown an upper bound on the number of possible valid compact signatures for each bag of $2^{\mathcal{O}(k^2)}$. Generating those by trying to create every possible signature and removing those that are not a compact-$\{z \rightarrow \text{FUTURE}\}$-restriction of a signature in the child bag is rather impractical, since the number of possible signatures far surpasses the upper bound of $2^{\mathcal{O}(k^2)}$. Instead, this paper introduces the sub-algorithm EXPLICIT_INTRODUCE_NODE, which is used in the TWITCH algorithm as mentioned in Section 4. EXPLICIT_INTRODUCE_NODE aims to create only the required signatures by extending existing signatures in the child bag.

5.1 Explicit introduce node

New node construction:
Let a be the node that is new in the introduce bag. To find out which existing or new node $a' \in D(N,T)$ could receive the iso-label a, the explicit introduce node procedure goes the following steps:

1. Initiate a tuple of lists named *present_list*. For each parent b of a in $D_{IN}(N_{IN}, T_{IN})$ check if there exists a node $b' = \iota^{-1}(b) \in D(N,T)$. If that is the case, do the following:
   - Create a list of all children c' of b' in $D(N,T)$ for which $\iota(c') = \text{FUTURE}$ and add it to the *present_list* tuple.
   - If $b \in N_{IN}$ and a is a tree node or a reticulation node, create a list of all outgoing arcs $b'c'$ of b' to children c' that could be de-compacted to create a'. To be able to de-compact them, they should have the following properties:
     (a) $\iota(c') \neq \text{PAST}$
     (b) $b\iota(c') \notin N_{IN}$
     (c) $b'c' \in \delta(A(T))$
2. For each child b of a in $D_{IN}(N_{IN}, T_{IN})$ do the same as in step 1 and add the acquired lists to the *present_list* tuple.
3. Initiate the list *possible_candidates* by taking the intersection $possible_candidates = \bigcap_{A_i \in present_list} A_i$. This is done to check which nodes or arcs found in steps 1 and 2 have all the present parents and children in the correct configuration.

4. Do the following based on the results of step 3:

   (a) If *possible_candidates* $\neq \emptyset$, do the following:

   - For each node $a' \in$ *possible_candidates*, create a copy of the signature in which the node is iso-labeled $\iota(a') = a$.
   - For each arc $b'c' \in$ *possible_candidates*, also create a copy signature in which the arc $b'c'$ is de-compacted to create a'. Then iso-label $\iota(a') = a$.

   (b) If *possible_candidates* $= \emptyset$ while there were present neighbors in the display graph, the function immediately returns $\emptyset$.

   (c) If *possible_candidates* $= \emptyset$ and there were no present neighbors in the display graph, do the following:

      i. Create a copy signature with a newly created node a' that is iso-labelled $\iota(a') = a$. Then do one of the following depending on a:
         - If $a \in N_{IN} \cap T_{IN}$ (a is a leaf), let $\delta(a') = a'$
         - If $a \in N_{IN} \setminus T_{IN}$: create $b' \in T$, iso-label it $\iota(b') =$ FUTURE and let $\delta(b') = a'$. Also create copy signature without b' where $a' \notin \delta(T)$.
         - If $a \in T_{IN} \setminus N_{IN}$: create $b' \in N$, iso-label it $\iota(b') =$ FUTURE and let $\delta(a') = b'$.

      ii. Check whether there are existing nodes with the FUTURE iso-label that have the same node type as a in $D_{IN}(N_{IN}, T_{IN})$. For each, create a copy signature with the chosen node iso-labelled a.

Adjacent arc construction:
Once these steps are done, there is a list of new signatures where the iso-label a has been assigned to a new or existing node a' in $D(N, T)$. The following steps construct any missing adjacent arcs and possible embedding paths of the node a'.

5. Make requirements for additional arcs of a' needed to make sure the numbers of incoming and outgoing arcs of a' in $D(N, T)$ match the numbers of incoming and outgoing arcs of a in $D_{IN}(N_{IN}, T_{IN})$. Two lists *parent_list* and *child_list* respectively contain the number of required parents and children that are currently missing. These lists also contain their types, which is used to make handle leafs and certain edge cases. An example of this can be seen in Figure 5.

(a) $D_{IN}(N_{IN}, T_{IN})$

(b) $D(N, T)$ during the introduction of a

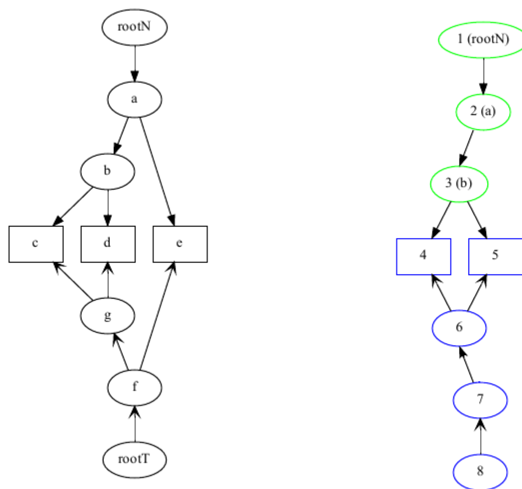


Figure 5: Example of a node introduction. Node a is being introduced in the signature shown in figure (b) by iso-labelling node $\iota(2) = a$. This signature has the embedding $\delta = \{8 \to 1,\ 7 \to 2,\ 6 \to 3,\ 4 \to 4,\ 5 \to 5\}$. In $D_{IN}(N_{IN}, T_{IN})$ in figure (a), a has one tree node child and one leaf child. In figure (b), a already has one child that is a tree node (node 3). Hence, the second child that is currently missing should be a leaf node. The *child_list* (that is used for arc requirements) therefore contains 1 leaf node.

6. Check if there are existing nodes or arcs that can be de-compacted in $D(N, T)$ to satisfy these requirements in *parent_list* and *child_list* from step 5.

7. If new embedding paths will be created from the arcs to neighbors, check if there are non-embedding nodes with degree 1 that can be inserted in between embedding paths. These will be referred to as "extra nodes".

8. Create a copy signature for each combination from steps 7 and 8, in which all a' adjacent arcs are added in the following way.

   (a) If a is a leaf, the arc to a parent in the network and the arc to the parent in the tree are added.

   (b) If $a \in T$, add neighbors (as specified in steps 5 and 6) of a and their embeddings. Additionally, add copies of the new signature with any combination of the "extra nodes" from step 9 added in between new network arcs and add combinations with extra new nodes specifically next to a.

(c) If $a \in \phi(T)$ (as determined in step 5), add neighbors (as specified in steps 5 and 6) of a and their corresponding embedded nodes.

(d) If $a \in N \setminus \phi(T)$, create all three options for embedding paths if a is a reticulation node or a tree node:

i. No path through a.

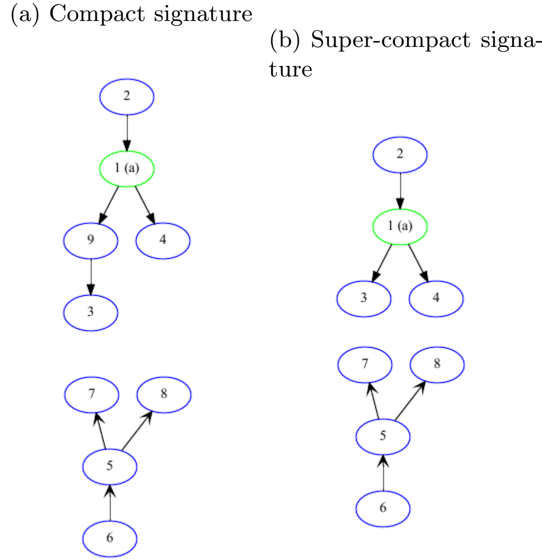
ii. A path through a from the leftmost parent or child of a.

iii. A path through a from the rightmost parent or child of a.

5.2 Super-compactness

When introducing a node into the network and creating its embedding node neighbor, a new non-embedding node can be inserted in between the introduced node and the embedding node neighbor. This does not alter the signature's compactness, since the non-embedding node is adjacent to the introduced PRESENT node. Since this is possible for all neighbors, it can increase the number of possible signatures considerably. Meanwhile, the added signatures also have a larger size due to the additional nodes, which can give more options for neighbors in the next node introductions. This motivates the following definition.

Definition 5.1. A signature is *super-compact* if it is compact and has no indegree-1, outdegree-1 non-embedding FUTURE network nodes with one FUTURE node neighbor.



(a) Compact signature

(b) Super-compact signature

Figure 6: Two signatures that show the difference between compactness and super-compactness. Let $\phi_a = \{6 \to 2, 5 \to 1, 7 \to 3, 8 \to 4\}$ be the embedding in both (a) and (b). Then node 9 in (a) is a non-embedding node that does not fulfill the super-compactness criteria.

When using path-decompositions, it is safe to only consider super compact restrictions since the non-embedding node can always be added later in between if needed. This intuition is proven later in Lemma 7. In the example from Figure 6b, this is done when node 9 is introduced by de-compacting (1, 3). However, only generating super-compact signatures can cause issues for the join node steps using tree-decompositions, which is shown in Section 6.3. This gives rise to an alternate sub-algorithm called SUPER-COMPACT_EXPLICIT_INTRODUCE_NODE, which is used in the PITCH algorithm that operates on path-decompositions instead of tree-decompositions.

5.3 Improvements

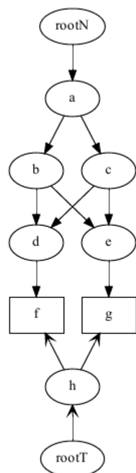
When introducing nodes in specific situations, some of the options can be abandoned before further processing is done. This saves time by requiring fewer actions or in some cases even having fewer signatures. Here is a list of the most notable optimizations made in the explicit introduce node implementation:

- **Forced embeddings:** When an introduced network node has two leaf children in the input graph, the node must be an embedding node. Hence, the introduce constructor abandons the processing of any signature with this condition where the node is not an embedding node. The same is done for the root node in the network.

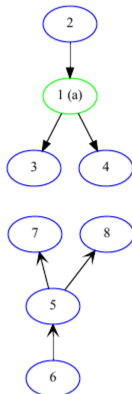
- **Interchangeable nodes and leaves:** When a new non-leaf node z is introduced in the THE THEORETICAL ALGORITHM and $\iota^{-1}(z)$ is missing a child c_z in the signatures of the child bag. Then THE THEORETICAL ALGORITHM will add signatures where c_z is an embedding node and signatures where c_z is a non-embedding node in the embedding path (z, c_z, c_{c_z}), where c_{c_z} is the child of c_z. In the latter signatures, there is a variant of the signatures in which c_{c_z} is a leaf and a variant of the signatures in which c_{c_z} is not a leaf. This is somewhat undesirable since it increases the number of signatures in each bag, however it is necessary in certain situations. When a non-leaf network embedding node z is introduced, it could be that (unbeknownst to the algorithm) the embedding solution(s) of $D_{IN}(N_{IN}, T_{IN})$ would require an embedding path from z to a leaf with two or more non-embedding nodes in between. An example of this situation is the display graph $D_{IN}(N_{IN}, T_{IN})$ shown in Figure 12a when node a is introduced first as an embedding node. The embedding path can only be extended by de-compacting it and adding another non-embedding node in between the path. This means that the end of the embedding path remains c_{c_z}, which is an issue when c_{c_z} is not a leaf.

  EXPLICIT_INTRODUCE_NODE handles this differently. Instead, it only creates signatures where c_{c_z} is not a leaf, and then fixes the issue when it occurs by changing c_{c_z} and its corresponding embedded node into a leaf. This can be done when a leaf node y is introduced and there is a tree node with outdegree 0 embedded to a network node that also has outdegree 0. In this case, EXPLICIT_INTRODUCE_NODE will create an extra signature where the two nodes are transformed into one leaf with the label y. This is functionally doing the same as originally passing both a signature where the node is a leaf and a signature where the nodes are separate. However, by only passing one option and changing it into the other when necessary, the work required for this structure in subsequent bags is reduced. This is even more relevant for SUPER-COMPACT_EXPLICIT_INTRODUCE_NODE, which is more reliant on adding nodes in between paths later than creating initial long paths, due to the super-compactness property. An example of the change of a node to a leaf is shown for SUPER-COMPACT_EXPLICIT_INTRODUCE_NODE in Figure 10.

(a) Input display graph



(b) Initial signature



(c) Subsequent signature

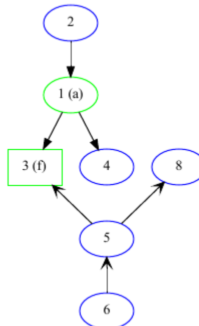


Figure 7: Example of a situation where SUPER-COMPACT_EXPLICIT_INTRODUCE_NODE uses interchangeable nodes and leaves. When the children of node 1 are created in (b), it is unknown whether they are leaves since a has no leaf children in the input display graph in (a). When node f is introduced in (c), nodes 3 and 7 are merged into a leaf node.

- **Root to root embedding:** Note that the introduce node implementation assumes that every embedding node can always have its incoming and outgoing arcs be embedding arcs to new or existing embedding nodes (with possible extra nodes in between). This is trivially true for outgoing arcs, since every path must lead to a leaf and leaves are always embedded to themselves. Hence, every embedding node always has a number of embedding node descendants equal or larger than its outdegree. For incoming arcs, this assumption is not trivial and requires that we always embed the root from the tree to the root from the network. Otherwise, no ancestor of the embedding node in the network corresponding to the root from the tree could be an embedding since there is no ancestor of the root in the tree to embed to them. To show that this assumption can be made without affecting the correctness of the algorithm, note the following lemma:

  Lemma 2. *If there exists an embedding on $D(N,T)$, then there exists an embedding where the root node in T is embedded to the root node in N.*

  Proof. Assume there is an embedding σ where the root node r_T in T is not embedded to the root node r_N in N, but instead to another node r'.

Let p be the path from r_N to r'. Then there can be no embedding nodes in p aside from r' itself. Let c be the child of r_T and let p_c be the embedding path in σ corresponding to the tree arc r_Tc. Then let σ' be an function on $T, A(T)$ identical to σ except for the node r_T being mapped to r_N and the arc r_Tc being mapped to the path $p_c \cup p$. Then σ' is an embedding, which proves the lemma. □

Embedding the root of the tree to the root of the network can only be done reliably when the roots have outdegree 1. A counterexample can be found in Section A.

Mapping the root to the root is very advantageous, since it eliminates one option (that multiplies with the other options) for every embedding node introduction in the network. It also guarantees that reticulation nodes are never embedding nodes, since a reticulation node could otherwise be the embedding node corresponding to the root.

Lemma 3. *Let X be an introduce bag that introduces node z, with child bag Y.* Explicit introduce node *generates every compact signature σ of which there exists a compact-$\{z \rightarrow FUTURE\}$-restriction in bag Y.*

Proof. Let σ_Y be a signature from bag Y and let z be the node that is introduced in bag X. Let σ_X be a signature from X such that σ_Y is the compact-$\{z \rightarrow$FUTURE$\}$-restriction of σ_X.

Assume a is a node in $D_X(N_X, T_X)$ that is not a neighbor of z or part of a embedding path of z. Then a must exist in $D_Y(N_Y, T_Y)$, since it cannot be removed by the compact-$\{z \rightarrow$FUTURE$\}$-restriction by the definitions of redundancy and compactness. Hence, any change from $D_Y(N_Y, T_Y)$ to $D_X(N_X, T_X)$ can only occur in neighbors of z or embedding paths of z.

Now assume that a is a neighbor of z in $D_{IN}(N_{IN}, T_{IN})$, and the arc between a and z is not added in $D_X(N_X, T_X)$. In subsequent bags, this arc can only be added to a parent bag when a compact-$\{a \rightarrow$FUTURE$\}$-restriction would remove it from a signature in the child bag. Since one of the arc's ends (z) will now no longer be iso-labelled FUTURE, the arc can never become redundant. Hence, the arc cannot be added by introduction nodes unless it is introduced immediately. If a itself is not an embedding node, the same holds for any arc ab from a to a new embedding node b.

Now, instead assume a is an existing node that should be part of a new embedding path of z, but is not added to it. In subsequent bags, this node can only be added to the embedding path if the compact-$\{a \rightarrow$FUTURE$\}$-restriction would remove it from a signature in the child bag. Since z is no longer iso-labelled FUTURE, the arc in the tree that is embedded to the embedding path can no longer become redundant. Hence, the arcs connecting a with the rest of the embedding path cannot become redundant and therefore a cannot become redundant. Therefore, a cannot be added to the embedding path by introduction nodes unless it is introduced immediately. Note that this does not apply if a was not already existent in the network. In that case, a can be introduced later by de-compacting an arc in the embedding path.

Hence, introduce node should add z to the signature and all its adjacent nodes and embedding paths (excluding nodes that can be added by de-compacting an arc later). This is exactly what is done in explicit introduce node, as explained in Section 5.1. □

Lemma 4. *Let X be an introduce bag that introduces node z, with child bag Y.* Explicit introduce node *only generates compact signatures σ of which there exists an compact-$\{z \rightarrow FUTURE\}$-restriction in bag Y.*

This is clear from the specific ways in which explicit introduce node creates new nodes and arcs. It only creates new nodes and arcs in the neighbors of the introduced node and its embedding paths. These are exactly the parts that can be changed from signatures in the previous bag, as shown in the first part of the proof of Theorem 3. These previous two lemmas can be combined into the following:

Lemma 5. *Let X be an introduce bag that introduces node z, with child bag Y.* Explicit introduce node *generates exactly all compact signatures σ of which there exists an compact-$\{z \rightarrow FUTURE\}$-restriction in bag Y.*

This is sufficient to prove the correctness of its use in the twitch algorithm due to the similarities with the theoretical algorithm. An identical lemma exists for Super-compact explicit introduce node:

Lemma 6. *Let X be an introduce bag that introduces node z, with child bag Y.* Super-compact explicit introduce node *generates exactly all super-compact signatures σ of which there exists an compact-$\{z \rightarrow FUTURE\}$-restriction in bag Y.*

The proof of this lemma is nearly identical to that of the combined proofs of Lemma 3 and Lemma 4. However, this alone is not sufficient to prove the workings of pitch. It is also necessary to show that every possible embedding structure that can occur within $D_{IN}(N_{IN}, T_{IN})$ can eventually be created by using super-compactness node introductions.

Lemma 7. *When every node is introduced once in a path-decomposition using* Super-compact_explicit_introduce_node*, every required possible embedding structure that can occur within $D_{IN}(N_{IN}, T_{IN})$ can eventually be created.*

Let a be the node that is given the iso-label z when z is introduced into $D_x(N_x, T_x)$. Let z' be a neighbor of z in the input graph, and let b be a node that will be given the iso-label z' at some point. Depending on the order of introductions, a and/or b can be newly created nodes, or a and/or b can already be present in $D_Y(N_Y, T_Y)$ with the FUTURE iso-label. Without loss of generality, assume b is a's child. In the following summation of cases, the cases where a node is introduced from the opposite side of the display graph (tree or network) will be noted with O.

1. $z \in T$
   Nodes a and b will be created if they do not already exist in the $D_y(N_y, T_y)$, after which the arc (a, b) will be constructed. Note the following variation of this case:

   (O1) $\phi(a)$ or $\phi(b)$ is introduced before a
   In this case, a signature will be created where an embedding path is created from $\phi(a)$ to $\phi(b)$. When this is done, (a, b) is constructed in T as the arc that is embedded to the embedding path.

2. $z \in \phi(T)$

   (a) $b \in \phi(T)$
   Since a and b are both embeddings, the arc (a, b) can be added as an embedding path without any special procedures. The embedded arc in the tree will be $(\phi^{-1}(a), \phi^{-1}(b))$. Note the following variation of this case:

   O2 $\phi^{-1}(a)$ or $\phi^{-1}(b)$ is introduced before a
   In this case, a signature will be created where an embedded arc is created from $\phi^{-1}(a)$ to $\phi^{-1}(b)$. When this is done, (a, b) is constructed in T as the embedding path corresponding to the embedded arc.

   (b) $b \notin \phi(T)$
   Since a is an embedding node, any arc starting at a must be part of an embedding path. An embedding path cannot stop at b since b is not an embedding node. Hence, the embedding path $((a, b), (b, c))$ will be created using some existing or new embedding node c. The embedded arc in the tree will thus be $(\phi^{-1}(a), \phi^{-1}(c))$. The creation of the embedding path differs depending on the previous existence of b:

   i. $b \in D_y(N_y, T_y)$
   Here, b will be used as an extra node and inserted in between a and c during the creation of the embedding path. This also applies to when $\phi^{-1}(a)$ is introduced before a (O3).

   ii. $b \notin D_y(N_y, T_y)$ Here, b will later be inserted in between a and c by de-compacting the arc (a, c). This also applies to when $\phi^{-1}(a)$ is introduced before a (O4).

3. $z \in N \setminus \phi(T)$ When a is not an embedding node, it cannot be the start of the path. Either a is created and connected to some existing or new embedding node c, or a is created by de-compacting an arc. In the latter case, a already has an embedding path going through it. In that case, b can only be a non-embedding node and it can be connected through an arc without any difficulties since there would be no path using the arc (a, b). If a is not created by de-compacting an arc, b can either be a non-embedding node or an embedding node:

(a) $b \in \phi(T)$

Let c be a new or existing embedding node. Then a embedding path can be constructed from c to b, going through a. Note the following variation of this case:

(O5) $\phi^{-1}(b)$ is introduced before a or b

In this case, a signature will be created where an embedded arc is created from $\phi^{-1}(a)$ to $\phi^{-1}(b)$. When this is done, (a, b) is constructed in T as the embedding path corresponding to the embedded arc.

(b) $b \notin \phi(T)$

In this case, (a, b) can be constructed as an arc without any embedding path using it. Alternatively, an embedding path can be created that goes through a, but uses b as an extra node in between a and an embedding descendant of a.

Hence, explicit introduce node can introduce any node with any arc to any other adjacent node, in any setting (tree nodes, embedding nodes, non-embedding nodes, combinations of these etc.), regardless of the order of introductions.

6 Join node

Similar to introduce node, THE THEORETICAL ALGORITHM generates every possible structure (compact reconciliations in this case), and then removes those that do not satisfy certain requirements. To avoid doing this, this paper introduces the sub-algorithm EXPLICIT_JOIN_NODE. This sub-algorithm instead creates all signatures that satisfy the requirements directly. This is done in the following way:

Let CV_X be a join bag and let σ_L and σ_R be signatures in the left child bag and the right child bag respectively

1. Calculate $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)$ and $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$. The implementation of this is very similar to that of the restrictions taken in forget node steps.

2. For each restricted signature calculated in step 1, check if there are nodes that can be changed into leaves (as described in Section 5.3). Change nodes that should certainly be leaves into leaves and try every combination of nodes or leaves for those that remain uncertain.

3. Check if the resulting signatures from step 2 are isomorphic (as described in Section 7.1). If they are isomorphic, compute the mapping and let σ'_L and σ'_R be the versions of the signatures σ_L and σ_R with the changed leaves.

4. Create the resulting signature $\sigma = \sigma'_L \underset{\{\text{L,R}\to\text{PAST}\}}{\cup} \big(\sigma'_R \backslash \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma'_R)\big)$, using the isomorphism mapping from step 3. The union $\underset{\{\text{L,R}\to\text{PAST}\}}{\cup}$ here denotes a construction where the display graph and embeddings are created using the union and its iso-labelling is defined using the subscript text instructions. The setminus $\backslash$ denotes a construction where all nodes and arcs from the signature right of the symbol are removed from the signature left of the symbol. The iso-labellings (aside from removing the aforementioned nodes) is unchanged. This construction is defined more formally in Section 6.2.

5. Compact any nodes that violate the compactness properties and remove any nodes or arcs that are redundant in σ.

The first two steps are done in a single loop that goes through all signatures of both the left and right bags once. The last two steps are done in a double loop that goes through every combination of one signature from the left bag and one signature from the right bag. The second and fourth steps are explained in detail in Section 6.1 and Section 6.2 respectively.

6.1 Uncertain leaf nodes

As mentioned in Section 5.3, nodes are sometimes changed into leaves when necessary. Situations can occur where a signature from one child bag has PAST

iso-labeled leaf nodes, while the signature from the other child bag has not yet changed these (now FUTURE iso-labeled) nodes into leaves. To compare these signatures, a series check is done on all tree nodes (since network nodes might be non-embedding nodes which cannot be leaves). These checks are shown in the flowchart in Figure 8.

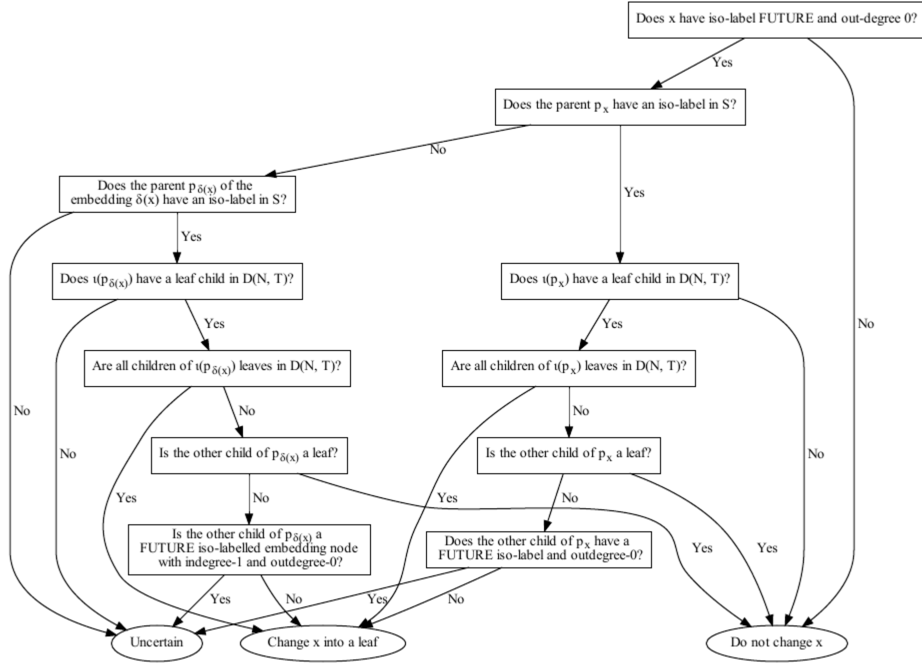


Figure 8: A flowchart that shows actions taken in EXPLICIT_JOIN_NODE, based on the properties of a tree node x.

Note that only the out-degree of the tree node is checked and not the in-degree or the degrees of the embedding node. This is done since tree nodes cannot have indegree-2 and since the embedding node always has identical in-degree and outdegree as the embedded tree node. Also note that the iso-labelling of the embedding node is not checked. If the embedding node would have an iso-labelling in $S \cup \{PAST\}$, then it would already either have been a leaf or a node with outdegree larger than 1 (and thus the embedded node also has outdegree larger than 1). Hence, it can be assumed that the embedding node has a FUTURE iso-label.

6.2 Signature construction

This subsections shows how to construct the resulting signature that was described in step 4 of EXPLICIT_JOIN_NODE in Section 6. Let $f : \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L) \to \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$ be the mapping for the isomorphism between the restrictions. Then the algorithm makes a copy σ of σ_L, and loops through every node $a \in D_R(N_R, T_R)$. If $f(a)$ is not defined, create a new node element a' (with a unique number). The set of nodes $a \in D_R(N_R, T_R)$ for which $f(a)$ is not defined is referred to in Section 6 as $\sigma'_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma'_R)$. When this is done for $a_1, \ldots, a_n$, extend

$$f'(x) = \begin{cases} f(x) \text{ if } x \in \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L) \\ a'_i \text{ if } x = a_i, \text{ for } i \in 1, \ldots, n \end{cases} .$$

When this is done, the algorithm loops through all arcs $ab \in D_R(N_R, T_R)$ and does adds the arc $f'(a)f'(b)$ to σ. In the python library networkx used for phylogenetic networks in this implementation, adding these arcs $f'(a)f'(b)$ also automatically adds any nodes $f'(a)$ or $f'(b)$ if they were not yet present in σ. The embeddings are added in the same way by using the extended mapping f'. The iso-labelling for $x \in D(N, T)$ is then defined by

$$\iota(x) = \begin{cases} \iota_L(x) & \text{if } f'(x) \text{ is undefined (i.e. } x \notin D_R(N_R, T_R)) \\ \iota_R(x) & \text{if } f'(x) \text{ is defined but } f(x) \text{ is undefined (i.e. } x \notin D_L(N_L, T_L) \\ \text{PAST} & \text{if } \underbrace{f(x) \text{ is defined}}_{\text{(i.e. } x \in \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L))} \text{ and } \iota_L(x) = \text{PAST or } \iota_R(f(x)) = \text{PAST} \\ \text{FUTURE} & \text{if } f(x) \text{ is defined and } \iota_L(x) \neq \text{PAST and } \iota_R(f(x)) \neq \text{PAST} \\ \iota_L(x) = \iota_R(x) & \text{if } f(x) \text{ is defined and } \iota_L(x) \in S \end{cases}$$

After these have been added, the resulting signature is the signature that was earlier referred to in Section 6 as $\sigma = \sigma'_L \underset{\{\text{L,R}\to\text{PAST}\}}{\cup} \left(\sigma'_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma'_R)\right)$. The following is an important lemma that shows the correctness of this join node approach.

Lemma 8. *Let $(L \cup R, S, F)$ be a join bag and let σ_L and σ_R be signatures in the left child bag and the right child bag respectively. The $\{PAST \to FUTURE\}$-restrictions of σ_L and σ_R are isomorphic if and only if there is a valid compact reconciliation μ and a signature σ such that:*

1. $\sigma = \phi_{\{LEFT,RIGHT\to PAST\}}(\mu)$
2. $\sigma_L = \phi_{\{LEFT\to PAST,RIGHT\to FUTURE\}}(\mu)$
3. $\sigma_R = \phi_{\{LEFT\to FUTURE,RIGHT\to PAST\}}(\mu)$

Proof. To prove "$\Leftarrow$", let there be valid compact reconciliation μ and a signature σ with properties $1, 2$ and 3 (as shown in Lemma 8). By the definition of a

reconciliation, there are no PAST nodes in μ. Hence,

$$\begin{aligned}\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L) &= \\ \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu)) &= \\ \phi_{\{\text{LEFT}\to\text{FUTURE},\text{RIGHT}\to\text{FUTURE}\}}(\mu) &= \\ \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\phi_{\{\text{LEFT}\to\text{FUTURE},\text{RIGHT}\to\text{PAST}\}}(\mu)) &= \\ \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R).\end{aligned}$$

Note that these equations hold true due to the transitivity of restrictions, which is proven by van Iersel et al. (2022) [1, Lemma 4].

To prove "$\Rightarrow$", let $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)$ and $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$ be isomorphic. For σ and μ, use the following constructions:

$$\begin{aligned}\mu =& \sigma_L \bigcup_{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}} \left(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\right) \\ \sigma =& \phi_{\{\text{LEFT, RIGHT}\to\text{PAST}\}}(\mu)\end{aligned}$$

Then property 1 follows immediately from the construction of σ and μ. For property 2, start with the direction

$$\begin{aligned}\sigma_L \subseteq& \phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu) \\ =& \phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}\left(\sigma_L \bigcup_{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}} \left(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\right)\right)\end{aligned}$$

Let ab be an arc in σ_L and assume $ab \notin \sigma_L \subseteq \phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu)$ Neither endpoint a or b can be given an iso-label from the present, otherwise the arc is always in $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu)$. There are three cases:

1. ab is a non-embedding arc. Then ab can only be redundant in σ if $\iota(a) = \iota(b)$. However, this is impossible since $\iota_L(a) = \iota_L(b)$ and $\iota_R(a) = \iota_R(b)$.

2. ab is a tree arc with embedding path $\delta(ab)$.

   (a) $\iota_L(a) = \iota_L(b) = \text{FUTURE}$. Assume the union changes the iso-labelling by having $\iota_L(b) = \iota_R(b) = \text{PAST}$. Then the $\{\text{LEFT} \to \text{PAST}, \text{RIGHT} \to \text{FUTURE}\}$-restriction still ensures that $\iota(a) = \iota(b) = \text{FUTURE}$. Assume that the nodes $c' \in \delta_L(ab)$ have iso-labelling $\iota_L(c') = \text{PAST}$. Then $\iota(c') = \text{PAST}$. Hence, ab does not become redundant in this construction and has not changed its iso-labelling either.

   (b) $\iota_L(a) = \iota_L(b) = \text{PAST}$. This is almost identical to (a).

3. ab is an embedding arc. This is almost identical to (2).

The proof of property 3 is identical when using the following construction:

$$\begin{aligned}\mu =&\sigma_L \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \big(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\big)\\ =&\sigma_L \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \sigma_R\\ =&\sigma_R \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \big(\sigma_L \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)\big).\end{aligned}$$

To prove the other direction ($\sigma_L \supseteq \phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu)$), let ab be an arc in

$$\begin{aligned}\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}(\mu) =\\ \phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}\Bigg(\sigma_L \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \big(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\big)\Bigg) =\\ \sigma_L \underset{\{\text{L}\to\text{PAST},\text{R}\to\text{FUTURE}\}}{\cup} \big(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\big)\Bigg) =\end{aligned}$$

and assume that $ab \notin \sigma_L$. Since the restriction cannot add arcs (it can only take them away) and $ab \notin \sigma_L$, it follows that $ab \in \big(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\big)$. Hence, $ab \in \sigma_R$ but ab is removed by the restriction $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ on σ_R. Since $ab \notin \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$, it also follows that $ab \notin \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)$ since they are isomorphic. However, ab is not removed by the $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}$ restriction of the union. In both restrictions, the nodes in R are iso-labelled FUTURE. Hence, the union with σ_L must add or changes something that prevents ab from being removed by $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}$.

For ab to become redundant in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$, it must be that $\iota_R(a), \iota_R(b) \notin S$. Since $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)$ and $\phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)$ are isomorphic, it follows that $\iota_L(a), \iota_L(b) \notin S$ since present nodes are unchanged by $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$. There are two options for ab:

1. ab is a non-embedding arc. Then ab would be redundant if $\iota(a) = \iota(b)$. Hence, having ab exist in σ_R would require $\iota_R(a) = \text{PAST}$ and $\iota_R(b) = \text{FUTURE}$ or the other way around, both of which go against the iso-labelling rules in σ_R.

2. ab is a tree arc with embedding path $\delta_R(ab)$. Since both become redundant with the same criteria, ab could be the the tree arc or some arc in the embedding path without loss of generalization. Assume $\delta_R(ab)$ contains no present nodes, otherwise it cannot become redundant in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$. Since PAST and FUTURE iso-labelled nodes may not be adjacent, it must be that $\iota_R(a) = \iota_R(b)$ and $\iota_R(a) \neq \iota_R(c')$ for $c' \in \delta_R(ab)$. There is again two cases:

   (a) $\iota_R(a) = \iota_R(b) = \text{PAST}$, $\iota_R(c') = \text{FUTURE}$ for $c' \in \delta_R(ab)$. If $\iota_L(a) = \text{FUTURE}$, then $\iota_(a) = \iota_R(a) = \text{PAST}$ due to the union

restriction. If $\iota_L(c') = \text{PAST}$ for $c' \in \delta_R(ab)$ then ab and the arcs in $\delta(ab)$ will become redundant immediately.

(b) $\iota_R(a) = \iota_R(b) = \text{FUTURE}$, $\iota_R(c') = \text{PAST}$ for $c' \in \delta_R(ab)$. This is very similar to the (a).

Hence, if ab would be made redundant from σ_R by $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$, there is nothing that can be added in the specified union with σ_L that can prevent ab from becoming redundant in $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}$.

Now assume that ab is removed in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ because an adjacent node is compacted. Note that the arc must be an embedding arc since otherwise with the other criteria of a compaction, it would be removed by redundancy. Compactions require nodes to be non-embedding nodes, have indegree-1 and outdegree-1, and have the same iso-label as its parent and child. Hence, there are three cases to be considered:

1. An endpoint of ab was compacted in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ due to a change in it being an embedding node, while the union has caused it to no longer be an embedding node. A restriction cannot remove a tree node from a signature without also removing its embedding node, so an embedding node in a signature can never become a non-embedding node.

2. One endpoint of ab was compacted in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ due to a change in its indegree and outdegree, but the union restriction has changed that. As already established, arcs within σ_R that are made redundant by $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ are also removed by the $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}$ even after taking the union with σ_L. Hence, the indegree and outdegree cannot be made smaller by taking the union. The indegree and outdegree can also not be made bigger, since this would require the node to have indegree-1 and outdegree-1 in σ_R. In this case, σ_R would not be a compact signature.

3. An endpoint of ab was compacted in $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ due to a change in it having the same iso-label as its child and parent in σ_R, but the union has changed that. It has already been established that the union restriction cannot cause an arc to have endpoints with differing iso-labels.

Since there cannot be a node without adjacent arcs in signatures, this also concludes that there is no node in σ_R that is removed by the $\phi_{\{\text{PAST}\to\text{FUTURE}\}}$ but not by $\phi_{\{\text{LEFT}\to\text{PAST},\text{RIGHT}\to\text{FUTURE}\}}$ after taking the union with σ_L.

Again, the proof of property 3 is identical when using the following construction:

$$\begin{aligned}\mu =&\sigma_L \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \big(\sigma_R \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_R)\big)\\ =&\sigma_L \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \sigma_R\\ =&\sigma_R \underset{\{\text{L}\to\text{LEFT},\text{R}\to\text{RIGHT}\}}{\cup} \big(\sigma_L \setminus \phi_{\{\text{PAST}\to\text{FUTURE}\}}(\sigma_L)\big).\end{aligned}$$

□

6.3 Super-compact incompatibility

From testing, the join node procedure implementation does not fully work with the super-compact node introductions. Signatures that represent similar graph structures and embeddings from the left and right child bags of a join node often do not match up. This is caused by a difference between the PAST iso-labelled graph structure parts in one child bag with the corresponding FUTURE iso-labelled structure parts in the other. Notably, the PAST iso-labelled graph structure parts seem to be larger with more nodes, even when using the same super-compact principle in the forget node procedure. This results in expected signatures missing from the join node bags, which further results in some of the test cases giving an incorrect *False* result. Due to the great time and effort required to identify the issues behind missing expected signatures in test cases (that often include thousands of signatures), gaining a greater understanding of this issue is rather tricky and trying to solve this problem has been left out of the scope of this paper.

7 Signature filters

Certain features of signatures can be used to disqualify them without affecting the end result of the algorithm. Often, this can be done before the creation of such a feature during the *explicit introduce node* sub algorithm. However, sometimes this can be difficult or impractical to implement in such a way. Instead, certain features are checked in loops that go through all signatures in a bag before the algorithm can proceed to the next bag. Since these checks occur after the features have already been created, the checks are sometimes referred to in this paper as filters. This section describes all filters in detail.

7.1 Isomorphism check

Definition 7.1. Two signatures $\sigma_1 = (D_1(N_1, T_1), \delta_1, \iota_1)$ and $\sigma_2 = (D_2(N_2, T_2), \delta_2, \iota_2)$ are *isomorphic* if there exists a bijective mapping $f : V(D_1(N_1, T_1)) \to V(D_2(N_2, T_2))$ such that:

1. $ab \in A(D_1(N_1, T_1)) \Leftrightarrow f(a)f(b) \in A(D_2(N_2, T_2))$.
2. $\iota_1(a) = \iota_2(f(a))$ for $a \in V(D_1(N_1, T_1))$
3. $f(\delta_1(a)) = \delta_2(f(a))$ for $a \in V(T_1)$
4. $f(\delta_1(ab)) = \delta_2(f(ab))$ for $ab \in A(T_1)$

Isomorphic signatures can occur in bags due to symmetries in the introduction of new nodes or due to the removal of non-symmetric nodes or arcs due to redundancy or compacting of PAST nodes. For two isomorphic signatures, the object class implementation will not return *True* when checked on equality. However, both signatures will behave exactly the same under all procedures of *TWITCH* or *PITCH*. Hence, they are detrimental to the running time of the algorithm, as the same work will need to be done multiple times. Worse still, the consequences of an isomorphism created in one bag may persist throughout the rest of the tree-decomposition if not removed. When this effect occurs in multiple bags, the running time increase can even be exponential in the number of bags in the tree decomposition.

To combat this problem, a check is done after each bag that removes isomorphisms. This uses the VF2 algorithm made by Cordella et all [10], which can detect isomorphisms between two directed graphs with and returns a mapping. This algorithm has a worst-case time complexity of $\mathcal{O}(|N|! \cdot |N|)$ and best-case time complexity of $\mathcal{O}(|N|^2)$. Before passing the display graph to this algorithm, several node and arc attributes are added to simulate the iso-labelling and the embedding in the display graph. This is done in the following way:

- For every node, add the node's iso-labelling to the node attribute.
- For every network arc, add a binary value that displays whether the arc is part of some embedding path.

- For every tree node, add a new outgoing arc to the corresponding embedding node in the network. Give the added arc a special attribute such that it doesn't interfere with normal arcs.

Lemma 9. *The three additional attributes mentioned above ensure that the VF2 algorithm only finds isomorphism mappings that match all 4 properties of the signature isomorphism (Definition 7.1).*

Proof. Assume the VF2 algorithm returns an isomorphism mapping. Property 1 is included in the definition of (non-attribute) graph isomorphisms, which the VF2 algorithm creates. Property 2 is ensured by adding the iso-labelling attributes to each node. For property 3, let there be nodes $a \in T_1$ and $a' \in N_1$ such that $\delta_1(a) = a'$. By construction there is an arc aa' with special attribute to indicate that it represents an embedding. By graph isomorphisms, it holds that $f(a)f(a') \in A(D_2(N_2, T_2))$. Then since an embedding is injective, there can be no other arcs with the special embedding attribute adjacent to $f(a)$ or $f(a')$. Hence $f(a)$ must be embedded to $f(a')$, or in other words $\delta_2(f(a)) = f(a') = f(\delta_1(a))$. For property 4, note that there can only be one embedding path from an embedding node to another embedding node. Since the arcs of such a path are given a special arc attribute, the mapping is forced to have embedding paths match between the two signatures. □

7.2 Descendant check

In the network N, the set of nodes that are adjacent to a certain node can change when arcs are de-compacted. However, when a node is a descendant of another node, it will always remain a descendant of that node. The descendant check goes through N by starting at the leaves and iterating through their predecessors until it reaches the root. During this iteration, it keeps track of every present node's set of present descendants. These sets of present descendants are then compared to pre-calculated sets of descendants for nodes in N_{IN}. If a present node has a present node descendant that is not a descendant in N_{IN}, the signature is eliminated.

When a node is changed or an arc is added somewhere in the display graph, all ancestors of that node may have their list of descendants altered. Hence, a local descendant check (that would only check nodes near the change) is often not sufficient for this purpose. Also note that when all nodes are checked for their descendants, there is no need to check for their ancestors due to symmetry (a is an ancestor of b if b is a descendant of a).

7.3 Distance check

The distance check removes signatures where for any combination of present nodes a, b such that there is a path from a to b, with present iso-labellings $\iota(a), \iota(b) \in S$, the following equation does not hold:

$$\underset{\max}{distance}(a, b) \leq \underset{\max}{distance}(\iota(a), \iota(b)) \tag{1}$$

Note that it can be assumed that there is a path from $\iota(a)$ to $\iota(b)$, otherwise the descendant check would have already removed the signature prior. The distance check can be done safely since a path between a and b in the signature cannot become shorter unless nodes are compacted or made redundant. Both require nodes to be iso-labelled PAST, which in turn requires the node to have had a present iso-label at some point. When the maximum distance between a and b is larger than the maximum distance in $D_{IN}(N_{IN}, T_{IN})$, there are not enough present iso-labels between a and b to create this path.

Additionally, for each present node a the distance to the furthest descendant is checked. The difference here is that the furthest descendant can also be a non-present node. If this distance is larger than the distance from $\iota(a)$ to its furthest descendant in $D_{IN}(N_{IN}, T_{IN})$ then the signature is removed.

Conveniently, the distances of the nodes in the signatures can be calculated in the same loop as the descendant check. This minimizes the amount of extra work required.

7.4 Grandparent check

The grandparent check removes signatures which have present nodes a, b with present iso-labellings $\iota(a), \iota(b) \in S$ such that $\iota(a)$ is the grandparent of $\iota(b)$ but a is not the parent or grandparent of b. When a node is introduced, all its neighbors are added immediately as previously explained in Section 5. Hence, if b is introduced after a, b can always be attached to a child of a or b can be introduced by de-compacting the arc between a and one of a's children. When a is introduced after b, the same can be done for a parent of b. Hence, any signature where a is not a parent or grandparent of b after both a and b have been introduced can be removed safely.

7.5 Sibling check

Sibling check removes signatures where two present nodes $a, b \in D(N, T)$ do not share a common parent while $\iota(a)$ and $\iota(b)$ do share a common parent in $D_{IN}(N_{IN}, T_{IN})$. When two present nodes $a, b \in D(N, T)$ do not share a common parent, for similar reasons as for the grandparent check, they cannot share a common parent in any subsequent signatures. Hence, those signatures where a and b do not share a common parent can be removed by the sibling check without affecting the outcome of the algorithm. Sibling check does the same for nodes who share a common child in $D_{IN}(N_{IN}, T_{IN})$, for which the same reasoning applies.

7.6 Neighbor check

Since arcs cannot be de-compacted in T, checks for T specifically can be much more thorough. For every node $a \in V(T)$, neighbor check collects the set of present neighbors $\{b \in V(T) \mid b \text{ is adjacent to } a \text{ and } \iota(b) \in S\}$, and checks if that set is contained within any of the pre-calculated sets of neighbors of nodes

in T_{IN}. If there is a node for which this does not hold true, the signature is removed. This can be done for any node in T, regardless of its iso-labelling.

7.7 Quantity checks

The quantity check is the simplest check performed on signatures. It checks a number of inequalities:

1. $|V(N)| \leq |V(N_{IN})|$
2. $|V(T)| \leq |V(T_{IN})|$
3. $|\delta^{-1}(V(N))| \leq |V(T_{IN})|$
4. $|\{a \in V(N)$ such that a is a reticulation node$\}| \leq \frac{|V(N_{IN})|-|V(T_{IN})|}{2}$
5. $|\{a \in V(N)$ such that a is a non-embedding tree node$\}| \leq \frac{|V(N_{IN})|-|V(T_{IN})|}{2}$

If any of these inequalities do not hold for a given signature, that signature is removed. Most of these checks do not remove signatures in large test-cases with large amounts of reticulations. They are instead most helpful to the algorithm when dealing with with very odd test-cases. An advantage of these is that most of them run in $O(1)$ speed.

8 Nice-decomposition manipulation

When an introduced node $a \in D_{IN}(N_{IN}, T_{IN})$ is adjacent $b \in D_{IN}(N_{IN}, T_{IN})$, the iso-labelling definition requires $\iota^{-1}(a)$ to be to be adjacent to $\iota^{-1}(b)$. This means that the introduction of a has fewer options, and thus results in fewer signatures, if b has already been introduced prior (and vice versa).

This property of having introduced nodes be adjacent to other present nodes can be prioritised during the transition from a tree/path-decomposition to a "nice" tree/path-decomposition. The following is a piece of pseudo code that shows how this is accomplished in the leaves of the decompositions:

Algorithm 4 Nice-decomposition manipulation

1: **procedure** IMPROVED_NICE_DECOMPOSITION_LEAVES($X(D(N,T))$)
2: $\quad$ $subgraphs = \emptyset$
3: $\quad$ **for** leaf bag $X_k \in X(D(N,T))$ **do**
4: $\quad\quad$ Let $X_k = \{x_1, \ldots x_n\}$
5: $\quad\quad$ **for** $x \in (X_k \setminus done_nodes)$ **do**
6: $\quad\quad\quad$ $todo_list \leftarrow todo_list \setminus \{x\}$
7: $\quad\quad\quad$ $done_nodes \leftarrow done_nodes \cup \{x\}$
8: $\quad\quad\quad$ $connected_nodes \leftarrow \{x\}$
9: $\quad\quad\quad$ **for** $y \in (X_k \setminus done_nodes)$ **do**
10: $\quad\quad\quad\quad$ **if** y is adjacent to $connected_nodes$ **then**
11: $\quad\quad\quad\quad\quad$ $connected_nodes \leftarrow connected_nodes \cup \{y\}$
12: $\quad\quad\quad\quad\quad$ $done_nodes \leftarrow done_nodes \cup \{y\}$
13: $\quad\quad\quad$ $subgraphs \leftarrow subgraphs \cup \{connected_nodes\}$
14: $\quad\quad$ Let $(C_n)_{n \leq |subgraphs|}$ be the sequence of sets $C_n \in subgraphs$, re-ordered from largest to smallest.
15: $\quad\quad$ Let $(a_n)_{n \leq |X_k|}$ be the sequence of elements in $C_1, \ldots, C_c$, where the first $|C_1|$ elements are in C_1 etc.
16: $\quad\quad$ Replace X_k with $(a_n)_{n \leq |X_k|}$ in $X(D(N,T))$
17: $\quad\quad$ **for** $0 < i \leq |X_k|$ **do**
18: $\quad\quad\quad$ Add the arc $((a_n)_{n \leq i}, (a_n)_{n \leq i-1})\}$ to $X(D(N,T))$
$\quad$ **return** $False$

Note that the subsequences used in line 18 refer to bags with the elements of the subsequence as its elements. Also note that the second loop for y does not iterate over $\{x\}$, as *done_nodes* has been updated between the two loops. Lastly, note that this code defines $(a_n)_{n \leq 0} = \emptyset$, which forms the leaf nodes in the nice-decomposition.

By the construction of the *connected_nodes* sets, the subsequent elements within the connected subgraphs are always adjacent to eachother. By the re-ordering of the sets based on their size, the algorithm also prioritises nodes in larger connected sub-graphs of present nodes over nodes in smaller connected sub-graphs of present nodes.

The nice-decomposition segments in between bags (as opposed to leaves) are

done in a somewhat similar way. The biggest difference is that the code now searches for nodes from the parent bag that are adjacent to those in the child bag, instead of searching for connected subgraphs. This makes the in between segments a little easier, and hence the pseudo code is omitted.

9 Implemented branching algorithms

This section introduces new implementations of two branching algorithms that do not use tree-decompositions. These algorithms have the added role within this paper to provide a benchmark for testing the tree-decomposition algorithms discussed in Section 4.

9.1 Brute force

The following is a brute force algorithm that, given $D_{IN}(N_{IN}, T_{IN})$, attempts to find an embedding of T_{IN} in N_{IN} by going through all possible embeddings in a depth-first manner.

Algorithm 5 Brute force

```
1:  procedure BRUTE_FORCE((D_IN(N_IN, T_IN)))
2:      todo_list ← {N_IN}
3:      while todo_list ≠ ∅ do
4:          N ← last element of todo_list
5:          todo_list ← todo_list \ {N}
6:          if there are reticulation nodes in N then
7:              x ← reticulation node
8:              N_L ← N
9:              Remove the left incoming arc of x in N_L
10:             "Clean up" the left parent of x in N_L
11:             N_R ← N
12:             Remove the right incoming arc of x in N_R
13:             "Clean up" the right parent of x in N_R
14:             todo_list ← todo_list ∪ {N_L, N_R}
15:         else
16:             if N' is isomorphic with T then
17:                 return True
        return False
```

Detecting whether N' is isomorphic with T is done using the tree isomorphism algorithm made by Aho et al. [11]. This algorithm's time complexity is linear in the number of nodes in the trees $\mathcal{O}(|T|)$.

When the incoming arcs are removed in lines 9 and 12, this can leave former tree node parents with indegree 1 and outdegree 1 or former reticulation node parents with indegree 2 and outdegree 0 [1]. Cleaning up a node in this algorithm hence refers to compacting these nodes with indegree 1 and outdegree 1 and removing these nodes with indegree 2 and outdegree 0. The latter can lead to a chain reaction where multiple cleanups need to be performed in a row.

The idea behind this brute force algorithm is relatively well-known and therefore a proof of its correctness is omitted in this paper. Its worst-case running

[1] Note that x cannot have a parent that is the root, since roots have outdegree 1 in this paper and thus are unable to have a reticulation node child.

time is $\mathcal{O}(2^r \cdot |T_{IN}|)$, where r is the number of reticulation nodes. For positive test-cases, the more possible embeddings exist, the more likely it is that the depth-first implementation of brute force finds one quickly. Hence, its average running time for positive test-cases will be faster by a factor of $s + 1$ where s is the number of possible embeddings of T_{IN} in N_{IN}.

9.2 BOTCH

This section introduces an algorithm that will be referred to in this paper as BOTCH. This is an acronym for "branching onwards to tree-child by Huijsman". This algorithm uses the cherry-picking algorithm from Janssen and Murakami (2021) [3] and specifically its implementation from Huijsman (2019) [12]. This algorithm can solve TreeContainment in linear time for the following class of networks:

Definition 9.1. A phylogenetic network is *tree-child* if it does not contain either of the following:

1. A reticulation node with a reticulation node parent.
2. A tree node whose children are both reticulation nodes.

BOTCH uses brute-force techniques to remove network nodes that do not follow this tree-child criteria, in order to turn the network into a tree-child network. This makes the implementation similar to that of the brute-force algorithm shown in Section 9.1, aside from the following differences:

1. The algorithm now only selects reticulation nodes that violate the tree-child criteria instead of all reticulation nodes for the branching process.
2. The isomorphism check is changed to the tree-child algorithm implementation.

The pseudo-code is as follows:

Algorithm 6 BOTCH

1: **procedure** BOTCH($(D_{IN}(N_{IN}, T_{IN}))$)
2: *todo_list* $\leftarrow \{N_{IN}\}$
3: **while** *todo_list* $\neq \emptyset$ **do**
4: $N \leftarrow$ last element of *todo_list*
5: *todo_list* $\leftarrow$ *todo_list* $\setminus \{N\}$
6: $x \leftarrow$ *node_selector(N)*
7: **if** $x \neq$ *None* **then**
8: $N_L \leftarrow N$
9: Remove the left incoming arc of x in N_L
10: "Clean up" the left parent of x in N_L
11: $N_R \leftarrow N$
12: Remove the right incoming arc of x in N_R
13: "Clean up" the right parent of x in N_R
14: *todo_list* $\leftarrow$ *todo_list* $\cup \{N_L, N_R\}$
15: **else**
16: **if** *cherry_picking_algorithm*$(N, T_{IN}) =$ *True* **then**
17: **return** *True*
return *False*

The simple function *node_selector* checks the parents and siblings of reticulation nodes to find a node that violates the tree-child criteria. When no node is found, a *None* statement is passed. The function *cherry_picking_algorithm* refers to the aforementioned implementation from Huijsman (2019) [12].

Its worst-case running time complexity is $\mathcal{O}(2^k \cdot |N_{IN}|)$, where k is the number of reticulation nodes that violate the tree-child criteria.

10 Testing method

10.1 Test graph generation

Positive and negative test cases are generated differently. Both involve first generating a tree and then generating N_{IN} based on the tree. For positive test-cases, the generated tree is used as T_{IN}. For negative test-cases, a second tree is generated independently of the first and then used as T_{IN}. This second randomly generated tree is not embedded in the network most of the time, as opposed to the first tree which is always embedded. When the second tree happens to be embedded by chance, it is removed from the test-cases manually.

Creating the tree is done by initiating all leaf nodes and then carefully yet randomly attaching tree nodes until there is just one connected subgraph remaining. The network is then created by taking a copy of the tree, changing the node names from all non-leaf nodes. Reticulation nodes are added by decompacting 2 arcs and connecting the nodes created from these de-compactions. This is done carefully to avoid creating loops in the graph. The pseudo code is on the next page.

Lines 14 to 18 warrant a bit of explanation here. The goal is to "replicate" tree arcs ab by constructing an arc cd. When $b > n - 1$, the node b is a tree node and the algorithm needs to let d be a new node corresponding to b, but exclusive to the network. This is done by adding the number of nodes in the tree $n - 1$ to it. When $b \leq n - 1$, the node b is a leaf node and the algorithm can use $d = b$ since leaves are identical in the tree and the network. The same is done for c, which is always a new node since leaves do not have outgoing arcs.

Line 23 mentions a function named *descendant_search*. This simple function takes a node and returns a list of all its descendants.

The implementation in the code uses uses two functions named *number_to_letters* and *letters_to_number*. These functions convert integers $0, 1, \ldots$ to strings $a, b, \ldots, y, z, aa, ab \ldots$ and back. These optional functions are used to create clarity between the letter nodes in T_{IN} and N_{IN} and the numbered nodes in signatures. The use of these functions is omitted in the pseudo-code.

The order of the resulting lists of arcs are then randomized to make sure the algorithms are not effected by it. For example, it influences the arc selection of the brute force algorithm, which causes it to find embeddings much faster for non-randomized lists of arcs than it would normally.

10.2 Decompositions

The tree-decompositions are made using the *treewidth_min_degree* function of the networkx python library, which uses the well-known MINIMUM DEGREE HEURISTIC shown by Bodlaender (1993) [7]. The path decompositions are the certificates from the *pathwidth* function in Sage. The running times of the algorithms that create these decompositions are not included in the running times for comparisons with brute force. Note that the tree-decompositions are made using a heuristic, while the path-decompositions are made using an exact algorithm.

Algorithm 7 Graph generation

1: **procedure** MAKE_TREE_ARCS(n)
2:     **for** $i \in \{0, \ldots, n-1\}$ **do**
3:         $leaf \leftarrow i$
4:         $leaf_list \leftarrow leaf_list \cup \{leaf\}$
5:     $arc_list \leftarrow \emptyset$
6:     **while** $leaf_list \neq \emptyset$ **do**
7:         $x \leftarrow n$
8:         $n \leftarrow n + 1$
9:         $y, z \leftarrow$random consecutive elements of $leaf_list$.
10:         $arc_list \leftarrow arc_list \cup \{(x, y), (x, z)\}$
11:         $leaf_list \leftarrow (leaf_list \setminus \{y, z\}) \cup \{x\}$
        return arc_list
12: **procedure** MAKE_NETWORK_ARCS($n, m, tree_arc_list$)
13:     **for** arc $ab \in tree_arc_list$ **do**
14:         $c \leftarrow a + n - 1$
15:         **if** $b > n - 1$ **then**
16:             $d \leftarrow b + n - 1$
17:         **else**
18:             $d \leftarrow b$
19:         $arc_list \leftarrow arc_list \cup \{(c, d)\}$
20:     $k = 3 * n - 2$
21:     **for** $i \in \{0, 1, \ldots, m\}$ **do**
22:         $(a, b), (c, d) \leftarrow$ random arcs in arc_list
23:         **while** $ab = cd$ or $a \in descendant_search(c)$ **do**
24:             $(a, b), (c, d) \leftarrow$ random arcs in arc_list
25:         $x \leftarrow k$
26:         $y \leftarrow k + 1$
27:         $k \leftarrow k + 2$
28:         $arc_list \leftarrow arc_list \setminus \{(a, b), (c, d)\}$
29:         $arc_list \leftarrow arc_list \cup \{(a, x), (x, b), (c, y), (y, d), (x, y)\}$
        return arc_list

This choice was made purely based on convenience of available existing implementations.

11 Results

This section contains the results of the testing that was done with the four algorithms discussed in this paper. To reiterate: TWITCH is the new tree-decomposition algorithm, PITCH is the new path-decomposition algorithm with super-compactness that allows for smaller structures, BRUTE FORCE is a well-known brute-force branching algorithm used primarily as a benchmark and BOTCH is a new branching algorithm that uses a cherry-picking algorithm.

Each of the following scenarios had up to 100 test cases. The algorithms were run on increasingly large test-cases until a test-case required more than an hour. Test-cases that took an hour are represented in the plots using cross markers. Due to the variance in running times for these algorithms, it is possible that some test-cases after the last performed one take less than an hour. These are not included, aside from one exception in Figure 10b due to some oddities. The test cases for PITCH were sometimes limited by the path-decomposition algorithm, which is why those running times often stop quite a bit before the running times get close to an hour. This can be seen when there is no cross marker at the end. Also note that the running times for generating the tree-decompositions and path-decompositions are not included in these results. The test cases where N_{IN} contains T_{IN} are referred to as *positive* test cases, and those where N_{IN} does not contain T_{IN} are referred to as *negative* test cases.

(a) positive test cases (b) negative test cases

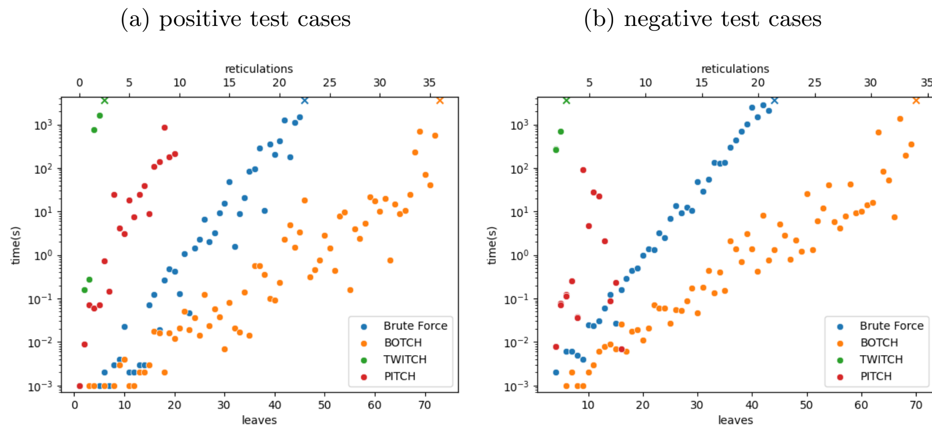
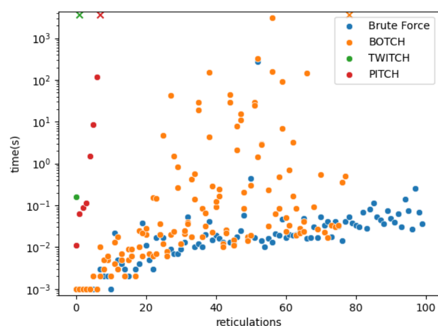


Figure 9: Test cases with a ratio of 2 leaves : 1 reticulation

(a) 5 leaves, varying reticulations, positive test cases



(b) 5 reticulations, varying leaves positive test cases

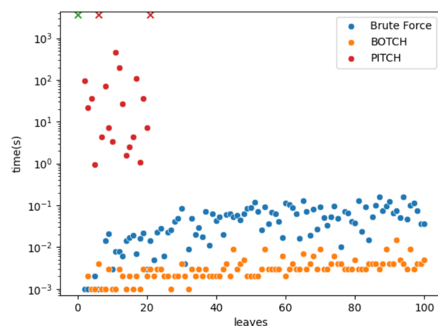
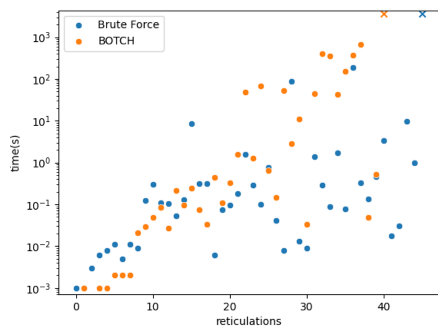


Figure 10: Test cases where one parameter is constant 5.

Figures 11 and 12 show the running time performance of BRUTE FORCE and BOTCH on test cases with larger input graphs. The TWITCH and PITCH algorithms were left out of these tests due to their inability to handle large input graphs as shown in earlier tests. The wrong answer is ignored in

(a) 10 leaves, varying reticulations, positive test cases



(b) 10 reticulations, varying leaves, positive test cases

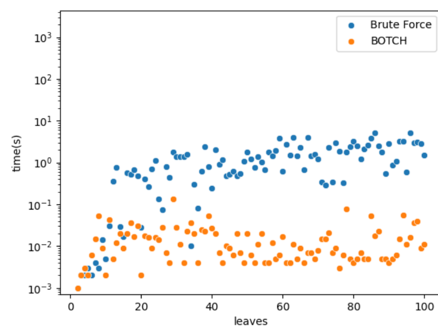


Figure 11: Test cases where one parameter is constant 10.

(a) 20 leaves, varying reticulations, positive test cases

(b) 20 reticulations, varying leaves, positive test cases

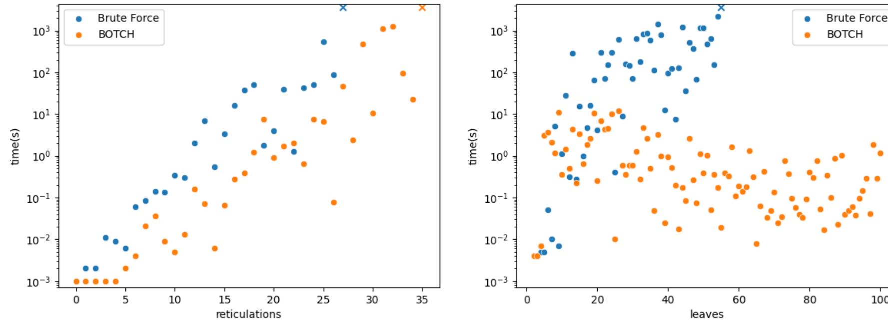


Figure 12: Test cases where one parameter is constant 20.

Note that all tests of TWITCH up to this point have been performed with an older version of the code, which did not contain two minor changes. Due to the nature of these changes, the new speed should be a bit slower.

To further analyse the time complexity behaviour of TWITCH, a series of tests have been performed on test cases that had a constant treewidth. These tests have been performed both with and without the quantity check described in Section 7.7, to see if that changes the behaviour. Test cases with up to 8 reticulations and were set up, but only those with 1 or 2 reticulations could be completed within an hour. Due to the low number of completed test cases, each test was done three times on different graphs with identical sizes. Table 1 contains the results of these tests.

	Test number	Quantity check: ON	Quantity check: OFF
1 reticulation	Test 1	0.38602s	3.30119s
	Test 2	0.31601s	2.89516s
	Test 3	0.85505s	60.2694s
	Average	**0.51903s**	**22.1553s**
2 reticulations	Test 1	26.8035s	90.2631s
	Test 2	196.707s	826.123s
	Test 3	423.327s	2340.86s
	Average	**215.613s**	**1085.75s**

Table 1: Results of TWITCH for testcases with constant treewidth 3 and constant number of leaves 3.

Another small test was done with PITCH to show the effectiveness of the signature filters and super-compactness. The results are shown in Table 2.

Bag	Signatures in PITCH	Signatures without signature filters	Signatures without super-compactness
Bag 1	3	4	11
Bag 2	11	64	56
Bag 3	40	550	207
Bag 4	111	1656	564
Bag 5	16	126530	DNF
Bag 6	120	DNF	DNF

Table 2: Numbers of signatures within bags for PITCH, PITCH without the signature filters and PITCH without super-compactness. These bags were taken from a path-decomposition of a positive test case with 8 leaves and 4 reticulations (reused from one of the tests shown in Figure 9a). In this path-decomposition, bag 0 was a leaf bag and bags 1-6 were all consecutive introduce bags. DNF is used to indicate that a bag did not finish within an hour.

During these tests, all of the four algorithms returned the correct results for both the positive and negative test cases that they managed to complete. Both TWITCH and PITCH reported a small amount of minor error debug messages during testing, but these have not affected the outcomes.

12 Conclusion

The attempts to create a version of the algorithm from van Iersel et al (2022) [1] that could be implemented in practice have been successful. Notably, TWITCH manages to produce the same outcomes as THE THEORETICAL ALGORITHM for the introduce node and join node steps in very different but implementation-wise feasible ways. Meanwhile, PITCH uses a path-decomposition instead of a tree-decomposition which allows for a further improvement of the introduce node step through super-compactness. The number of possible signatures in these algorithms has been further reduced by the filters shown in Section 7 and the nice-decomposition manipulation shown in Section 8.

Despite these efforts to increase the speed of TWITCH and PITCH, they have still failed to outperform the benchmark implementation of BRUTE FORCE as shown in Section 11. Additionally, although the number and size of test cases that could be completed was somewhat limited, it does seem that the tree-width and path-width-based running time scaling of TWITCH and PITCH in their current implementations is worse than the reticulation-based running time scaling of BRUTE FORCE.

From the tests in Section 11, it seems that the other new algorithm BOTCH is the fastest among the tested algorithms. Only in some of the more odd test cases from Figure 10a and Figure 11a with high number of reticulations and very low numbers of leaves, did it occur that BRUTE FORCE was faster than BOTCH. Having very few leaves increases the proportion of reticulation nodes in N_{IN}, which increases the percentage of reticulation nodes that violate the tree-child properties. When most reticulation nodes violate the tree-child property, BOTCH loses its main advantage of having to branch much less than BRUTE FORCE. Interestingly, debug information shows that BRUTE FORCE requires far fewer embedding "guesses" than BOTCH to find a correct embedding. When the number of reticulations is much higher than the number of leaves, there are usually many correct possible embeddings. It could be possible that the BOTCH algorithm's consecutive embedding "guesses" are more similar to each other than the consecutive embedding "guesses" of the BRUTE FORCE algorithm. Another oddity in the comparison between BOTCH and BRUTE FORCE can be seen in Figure 12b. When leaves are added, the proportion of reticulation nodes within N_{IN} decreases due to the increase in tree nodes. Hence, the number of reticulation nodes that violate the tree-child properties decreases, which improves the running time of BOTCH. For BRUTE FORCE, an opposite effect occurs. While the number of reticulation nodes remains constant, the average number of reticulations removed per branch due to consecutive cleanups decreases due to the increase in tree nodes. This means that more branches need to be checked, which slows down the running time of BRUTE FORCE.

Direct comparisons between TWITCH and PITCH are made somewhat difficult since the TWITCH implementation uses a tree-decomposition heuristic while the PITCH implementation uses an exact path-decomposition, which is not accounted for in the running times. Based on the running time results and the signature quantity tests, I would expect the difference between compact signa-

tures and super-compact signatures to have a greater impact on the running time than the difference between using a tree-decomposition and using a path-decomposition. Furthermore, given that the combined running times of both PITCH and the exact path-decomposition (which never ran for more than an hour due to the website limitations) combined were shorter than TWITCH, it seems that PITCH is superior to TWITCH in their current implementations.

Both BRUTE FORCE and BOTCH are depth-first algorithms, which means that there is much more variance in their running time for positive test cases. Meanwhile for negative test cases, these algorithms must always try every combination. PITCH and TWITCH instead run breadth-first, which means that their positive test cases should be very consistent. However, by the nature of those algorithms, they can stop immediately when finding a structure that has no possible embedding. Hence, they are expected to have more variance in running times for negative test cases. The most noticeable example of this is the incredible speed of the last solved PITCH test case in Figure 9b.

12.1 Future recommendations

With BOTCH being the fastest among the tested algorithms in the running time tests, it might be the most fruitful to seek improvements for BOTCH. Here are two theoretical ways in which I expect the BOTCH algorithm could be improved:

- Instead of first creating all branches and then processing each of them using the cherry-picking algorithm, I conjecture that it might be possible to instead first run the cherry-picking algorithm and branch when needed. This means that for negative cases, the cherry-picking process until the first branch would only be done once instead of 2^k times where k is the number of reticulation nodes that violate tree-child properties. With the rough approximation that all cherry-picking processes in between the violating reticulation nodes take an equal amount of time, this would result in an average negative test-case running time of $\mathcal{O}(\sum_{i=0}^{k} 2^i \frac{|D(N,T)|}{k}) = \mathcal{O}((2^{k-1}-1)\frac{|D(N,T)|}{k}) = \mathcal{O}(\frac{2^k}{k}|D(N,T)|)$ as opposed to the current average $\mathcal{O}(2^k|D(N,T)|)$. This approach does hinge on the correctness of the cherry-picking algorithm when used interchangeably with the branching that removes reticulation nodes, which is unproven as of now. Note that if this works, it also becomes possible to use the cherry-picking algorithm to give other algorithms a "head start" by running it until the first necessary branch and then switching to another algorithm. This could be advantageous when the other algorithm is slower by default but faster than the 2^k branching on reticulation nodes that violate the tree-child property.
- Currently, BOTCH selects its reticulation nodes that violates the tree-child properties in an arbitrary (deterministic by the order of the arcs in the implementation of the input graphs). However, it might be advantageous to select these reticulation nodes more carefully when dealing with networks with very large numbers of reticulations. When there is a chain of n descendants that are all consecutive reticulation nodes, removing the lowest

arc connecting them will cause the cleanup to remove all of them. This results in $n+1$ branching options when always processing the lowest reticulation node first, instead of the 2^n options when always processing the highest reticulation node first. Somewhat similarly, when there is three or more sibling nodes whose parents are reticulation nodes, processing the middle ones is more effective than processing the ones on the edge.

Due to the sheer size and complex nature of the TWITCH and PITCH algorithms, there are also many ways to improve them. While I am not sure I would recommend it given the poor performance of these algorithms in the tests, some of them are listed here:

- Change the approach of PITCH to depth-first instead of breadth-first. Note that this is impossible for TWITCH due to join nodes. A depth-first implementation would be able to find solutions before trying all other options, which means the theoretical average running time can be divided by $1 + s$ where s is the number of possible embeddings. In practice, this might be even better since the isomorphism checks (which iterates through every pair of signatures) take less time for bags in earlier depth-first iterations since there will be fewer other signatures in the bags.
- Find a way to make the join node steps of TWITCH compatible with super-compactness.
- The current decomposition manipulation only prioritises nodes that are adjacent to each other in $D_{IN}(N_{IN}, T_{IN})$. Given the existence of the sibling and grandparent filters that deal with nodes that have distance 2 between them in $D_{IN}(N_{IN}, T_{IN})$, it might be beneficial to have the nice-decomposition manipulation also gives some priority to nodes with distance 2. Even more impactful would be finding or creating a heuristical decomposition algorithm that can provide tree- or path-decompositions where the nodes in bags are more often adjacent to each other by default.

Aside from improving these algorithms further, there are also other places in which further advancements could still be possible. BOTCH uses branching until the input network is tree-child, after which it can use the cherry-picking algorithm. It might be possible to use a similar approach with other algorithms that can solve TREECONTAINMENT on specific subsets of networks. For example, it might be interesting to try branching until the network is nearly stable or reticulation visible, after which the linear running time algorithm from Weller (2017) [5] could be used. Alternatively, one could look at other problems within phylogenetics. Now that is the tree-width approach is shown to be implementable for TREECONTAINMENT, it might be interesting for future research to take a look at tree-width approaches for TREECONTAINMENT or even HYBRIDIZATIONNUMBER.

References

[1] Leo van Iersel, Mark Jones, and Mathias Weller. Embedding phylogenetic trees in networks of low treewidth, 2022. 30th Annual European Symposium on Algorithms, ESA 2022.

[2] Iyad A. Kanj, Luay Nakhleh, Cuong Than, and Ge Xia. Seeing the trees and their branches in the network is hard. *Theoretical Computer Science*, 401(1):153–164, 2008.

[3] Remie Janssen and Yukihiro Murakami. On cherry-picking and network containment. *Theoretical Computer Science*, 856:121–150, 2021.

[4] Philippe Gambette, Andreas D. M. Gunawan, Anthony Labarre, Stéphane Vialette, and Louxin Zhang. Solving the tree containment problem for genetically stable networks in quadratic time. In Zsuzsanna Lipták and William F. Smyth, editors, *Combinatorial Algorithms*, pages 197–208, Cham, 2016. Springer International Publishing.

[5] Mathias Weller. Linear-time tree containment in phylogenetic networks. In *RECOMB International conference on Comparative Genomics*, pages 309-323. Spinger 2018.

[6] Steven Kelk, Georgios Stamoulis, and Taoyang Wu. Treewidth distance on phylogenetic trees. *Theoretical Computer Science*, 731:99–117, 2018.

[7] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11:1–21, 1993.

[8] Remie Janssen, Mark Jones, Steven Kelk, Georgios Stamoulis, and Taoyang Wu. Treewidth of display graphs: bounds, brambles and applications. *J. Graph Algorithms Appl.*, 23:715–743, 2019.

[9] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.7)*, 2021. `https://www.sagemath.org`.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.

[11] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The design and analysis of computer algorithms, 1974.

[12] Robbert Huijsman. Tree-child network containment. Bachelor's thesis, Delft University of Technology, 2019.

A Root to root embedding counterexample

The following counterexample shows that for roots with outdegree 2, not every display graph that has an embedding also has an embedding where the roots are mapped to eachother.

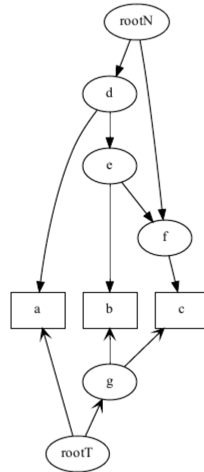
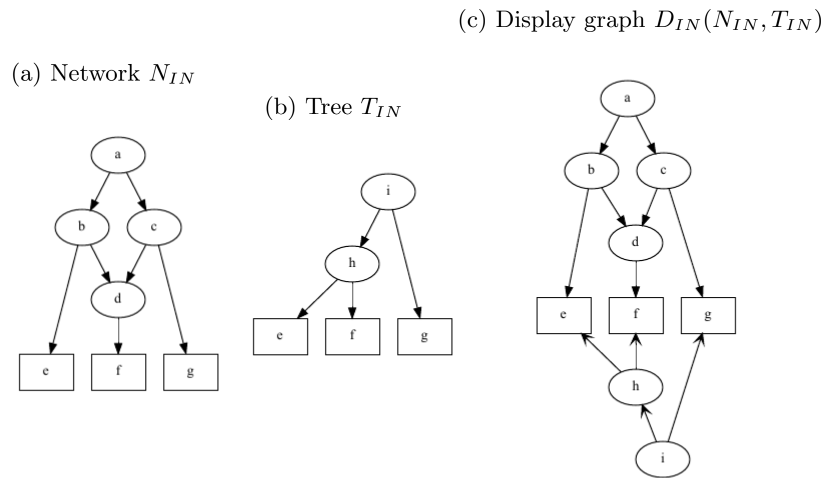


Figure 13: An example of a display graph where the root of the tree cannot be embedded to the root of the network. However, the root of the tree can be embedded to node d.

B Full example

In this example, THE THEORETICAL ALGORITHM is used to see whether there exists an embedding of the tree shown in 14b in the network from figure. 14a.



(a) Network N_{IN}

(b) Tree T_{IN}

(c) Display graph $D_{IN}(N_{IN}, T_{IN})$

Figure 14: The construction of a display graph from a tree and a network. Note that for this example, the tree and network have roots with outdegree 2 instead of 1.

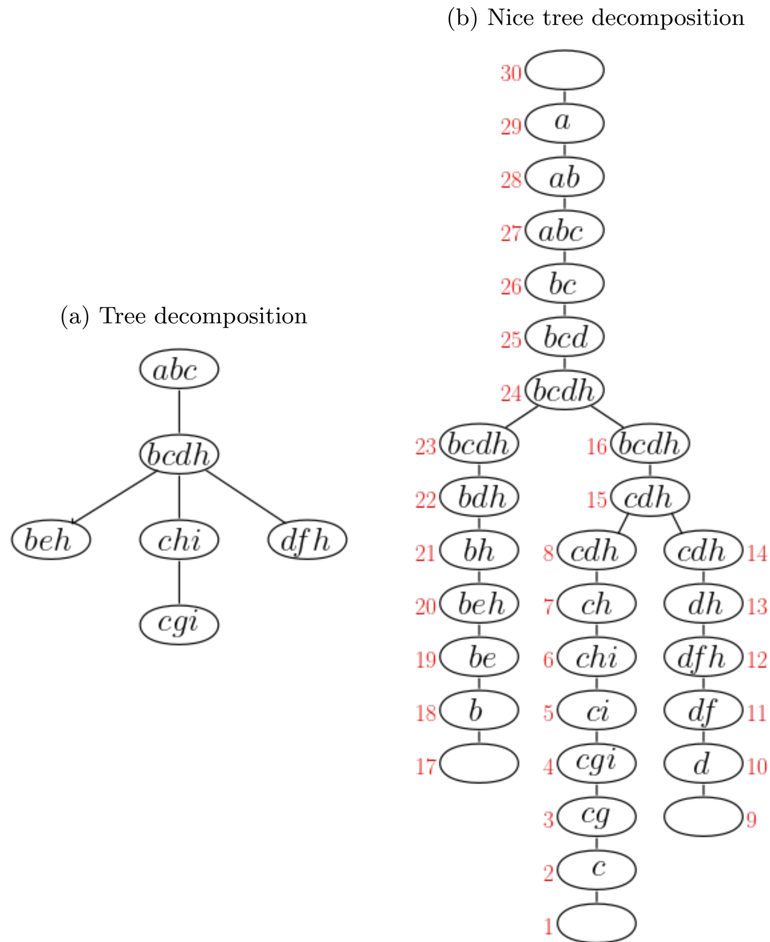


Figure 15

Even in small examples, the total number of possible signatures for every bag within the nice tree decomposition is rather large. Hence, the following example only shows one signature for each bag. Specifically, the signatures chosen are the signatures that are required to reach the last step of the algorithm. Let ϕ be the embedding of $D_{IN}(N_{IN}, T_{IN})$, and for bags $i \in \{1, \ldots, 30\}$ let ϕ_i be this embedding restricted to $D_i(N_i, T_i)$. To save space in this example, the display graphs are denoted by stating their nodes. The set also includes all arcs from included nodes to other included nodes in $D_{IN}(N_{IN}, T_{IN})$, except when explicitly stated otherwise. Note that this notation could be somewhat confusing, since (in the algorithm) the only mapping from a signature display graph to the input display graph $D_{IN}(N_{IN}, T_{IN})$ is given by the iso-labelling. The signatures are as follows:

1. Bag 1 is a leaf bag. For every valid signature in the leafs, there may be

no nodes assigned with the Past label. Since $S = \emptyset$ in the roots, the only available label that a vertex can be given is $Future$. Hence, if the display graph $D_1(N_1, T_1)$ of this bag contained any nodes, they would immediately become redundant. Thus σ_1 uses the rather empty signature

$\sigma_1 = (\emptyset, \delta_1, \iota_1)$, where $\iota_1(v)$ is the empty function.

2. Bag 2 is an introduce bag, which means that it contains all signatures for which the previous bag contained the compact-$\{z \to FUTURE\}$-restriction. Observe the following signature which is used for bag 2:

   $\sigma_2 = (\{a, c, d, g, i\}, \phi, \iota_2)$ where
   $$\iota_2(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, g, i\} \\ c & \text{if } v = c \end{cases}$$

   To see that this signature is actually contained within bag 2, note that its $\{c \to \text{FUTURE}\}$-restriction is σ_1 since giving c the future label would make all nodes and arcs redundant.

3. Similar to the previous bag, bag 3 is also an introduce bag and contains the signatures for which the previous bag contains the $\{g \to \text{FUTURE}\}$-restrictions. Using σ_2, this gives

   $\sigma_3 = (\{a, c, d, g, i\}, \phi, \iota_3)$ Where
   $$\iota_3(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, i\} \\ v & \text{if } v \in \{c, g\} \end{cases}$$

4. Bag 4 is another introduce bag, which introduces vertex i. Like bag 2, the following signature follows from the previous somewhat non-trivially:

   $\sigma_4 = (\{a, b, c, d, g, h, i\}, \phi, \iota_4)$ where
   $$\iota_4(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, d, h\} \\ v & \text{if } v \in \{c, g, i\} \end{cases}$$
   To see that the $\{i \to \text{FUTURE}\}$-restriction of σ_4 is indeed σ_3, note that the restriction makes the arcs ab and ih redundant. When these arcs become redundant, the vertices b and h also become redundant.

5. Bag 5 is a forget bag which forgets vertex g. Therefore, it contains the $\{g \to \text{PAST}\}$-restrictions of signatures from the previous bag. This results in the following signature:

   $\sigma_5 = (\{a, b, c, g, h, i\}, \phi, \iota_5)$ where
   $$\iota_5(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, d, h\} \\ \text{PAST} & \text{if } v \in \{g\} \\ v & \text{if } v \in \{c, i\} \end{cases}$$

6. Bag 6 is an introduce bag which introduces vertex h. This has the following signature: $\sigma_6 = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_6)$ where

$$\iota_6(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, d, e, f\} \\ \text{PAST} & \text{if } v \in \{g\} \\ v & \text{if } v \in \{c, h, i\} \end{cases}$$

To see that the $\{h \to \text{FUTURE}\}$-restriction of σ_6 is indeed σ_5, note that the restriction makes the arcs bd, be, df, he and hf redundant. When these arcs become redundant, the vertices e and f also become redundant.

7. Bag 7 is a forget bag which forgets vertex i. This has the following signature: $\sigma_7 = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_7)$ where

$$\iota_7(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, d, e, f\} \\ \text{PAST} & \text{if } v \in \{g, i\} \\ v & \text{if } v \in \{c, h\} \end{cases}$$

8. Bag 8 is an introduce bag which introduces vertex d. This has the following signature: $\sigma_8 = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_8)$ where

$$\iota_8(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, e, f\} \\ \text{PAST} & \text{if } v \in \{g, i\} \\ v & \text{if } v \in \{c, d, h\} \end{cases}$$

9. Bag 9 is a leaf bag which (like bag 1) has the rather empty signature:

   $\sigma_9 = (\emptyset, \phi, \iota_9)$, where $\iota_9(v)$ is the empty function.

10. Bag 10 is an introduce bag which introduces vertex d. This has the following signature: $\sigma_{10} = (\{b, c, d, f, h\}, \phi, \iota_{10})$ where

$$\iota_{10}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{b, c, f, h\} \\ v & \text{if } v \in \{d\} \end{cases}$$

To see that the $\{d \to \text{FUTURE}\}$-restriction of σ_{10} is indeed σ_9, note that the restriction makes the arcs bd, cd, df and hf redundant. When these arcs become redundant, all vertices in the display graph also become redundant.

11. Bag 11 is an introduce bag which introduces vertex f. This has the following signature: $\sigma_{11} = (\{b, c, d, f, h\}, \phi, \iota_{11})$ where

$$\iota_{11}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{b, c, h\} \\ v & \text{if } v \in \{d, f\} \end{cases}$$

12. Bag 12 is an introduce bag which introduces vertex h. This has the following signature: $\sigma_{12} = (\{a, b, c, d, e, f, h, i\}, \phi, \iota_{12})$ where

$$\iota_{12}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, c, e, i\} \\ v & \text{if } v \in \{d, f, h\} \end{cases}$$

To see that the $\{h \to \text{FUTURE}\}$-restriction of σ_{12} is indeed $\sigma_1 1$, note that the restriction makes the arcs ab, be, he and ih redundant. When these arcs become redundant, the vertices a, e and i also become redundant.

13. Bag 13 is a forget bag which forgets vertex f. This has the following signature: $\sigma_{13} = (\{a, b, c, d, e, f, h, i\}, \phi, \iota_{13})$ where
$$\iota_{13}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, c, e, i\} \\ \text{PAST} & \text{if } v \in \{f\} \\ v & \text{if } v \in \{d, h\} \end{cases}$$

14. Bag 14 is an introduce bag which introduces vertex c. This has the following signature: $\sigma_{14} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{14})$ where
$$\iota_{14}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, e, g, i\} \\ \text{PAST} & \text{if } v \in \{f\} \\ v & \text{if } v \in \{c, d, h\} \end{cases}$$
To see that the $\{d \to \text{FUTURE}\}$-restriction of σ_{14} is indeed $\sigma_1 3$, note that the restriction makes the arcs ac, cg and ig redundant. When these arcs become redundant, the vertex g also become redundant.

15. Bag 15 is a join bag with children bags 8 and 14. Consider the following well-behaved F-partial solution: $\sigma_{\text{F-partial}} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{F-partial})$ where
$$\iota_{F-partial}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, e, f, g, i\} \\ v & \text{if } v \in \{c, d, h\} \end{cases}.$$
Then consider the reconciliation that is the $(L \to \text{LEFT}, R \to \text{RIGHT})$-restriction of the previous F-partial solution: $\mu = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_\mu)$ Where
$$\iota_\mu(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, e\} \\ \text{RIGHT} & \text{if } v \in \{f\} \\ \text{LEFT} & \text{if } v \in \{g, i\} \\ v & \text{if } v \in \{c, d, h\} \end{cases}$$
Then $\sigma_L = \sigma_8$ and $\sigma_R = \sigma_{14}$. Hence, the algorithm adds the compact-$\{\{\text{LEFT}, \text{RIGHT}\} \to \text{PAST}\}$-restriction: $\sigma_{15} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{15})$ where
$$\iota_{15}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, b, e\} \\ \text{PAST} & \text{if } v \in \{f, g, i\} \\ v & \text{if } v \in \{c, d, h\} \end{cases}$$

16. Bag 16 is an introduce bag which introduces vertex b. This has the following signature: $\sigma_{16} = (\{a, b, c, d, e, f, h, i\}, \phi, \iota_{16})$ where
$$\iota_{16}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, e\} \\ \text{PAST} & \text{if } v \in \{f, g, i\} \\ v & \text{if } v \in \{b, c, d, h\} \end{cases}$$

17. Bag 17 is a leaf bag which with signature:
$\sigma_{17} = (\emptyset, \phi, \iota_{17})$, where $\iota_{17}(v)$ is the empty function.

18. Bag 18 is an introduce bag which introduces vertex b. This has the following signature: $\sigma_{18} = (\{a, b, d, e, f, h, i\}, \phi, \iota_{18})$ where
$$\iota_{18}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, e, f, h, i\} \\ v & \text{if } v \in \{b\} \end{cases}$$
To see that the $\{d \to \text{FUTURE}\}$-restriction of σ_{18} is indeed σ_{17}, note that the restriction makes the arcs ab, bd, be, eh, df, hf and ih redundant. When these arcs become redundant, all vertices in the display graph become redundant.

19. Bag 19 is an introduce bag which introduces vertex e. This has the following signature: $\sigma_{19} = (\{a, b, e, d, f, h, i\}, \phi, \iota_{19})$ where
$$\iota_{19}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, f, h, i\} \\ v & \text{if } v \in \{b, e\} \end{cases}$$

20. Bag 20 is an introduce bag which introduces vertex h. This has the following signature: $\sigma_{20} = (\{a, b, d, e, f, h, i\}, \phi, \iota_{20})$ where
$$\iota_{20}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, f, i\} \\ v & \text{if } v \in \{b, e, h\} \end{cases}$$

21. Bag 21 is a forget bag which forgets vertex e. This has the following signature: $\sigma_{21} = (\{a, b, d, e, f, h, i\}, \phi, \iota_{21})$ where
$$\iota_{21}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, d, f, i\} \\ \text{PAST} & \text{if } v \in \{e\} \\ v & \text{if } v \in \{b, h\} \end{cases}$$

22. Bag 22 is an introduce bag which introduces vertex d. This has the following signature: $\sigma_{22} = (\{a, b, c, d, e, f, h, i\}, \phi, \iota_{22})$ where
$$\iota_{22}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, c, f, i\} \\ \text{PAST} & \text{if } v \in \{e\} \\ v & \text{if } v \in \{b, d, h\} \end{cases}$$
Note that the arc bd would be missing from this signature, as it would otherwise be redundant.

23. Bag 23 is an introduce bag which introduces vertex c. This has the following signature: $\sigma_{23} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{23})$ Where
$$\iota_{23}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, f, g, i\} \\ \text{PAST} & \text{if } v \in \{e\} \\ v & \text{if } v \in \{b, c, d, h\} \end{cases}$$
To see that the $\{d \to \text{FUTURE}\}$-restriction of σ_{23} is indeed σ_{22}, note that the restriction makes the arcs ac, cg and ig redundant. When these arcs become redundant, the vertex g becomes redundant.

24. Bag 24 is a join bag with children bags 23 and 16. Consider the following well-behaved F-partial solution: $\sigma_{\text{F-partial}} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{F-partial})$ where

$$\iota_{F-partial}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a, e, f, g, i\} \\ v & \text{if } v \in \{b, c, d, h\} \end{cases}$$

Then consider the reconciliation that is the $(L \rightarrow \text{LEFT}, R \rightarrow \text{RIGHT})$-restriction of the previous F-partial solution: $\mu = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_\mu)$ where

$$\iota_\mu(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a\} \\ \text{RIGHT} & \text{if } v \in \{f, g, i\} \\ \text{LEFT} & \text{if } v \in \{e\} \\ v & \text{if } v \in \{b, c, d, h\} \end{cases}$$

Then $\sigma_L = \sigma_{23}$ and $\sigma_R \sigma_{16}$. Hence, the algorithm adds the compact-$\{\{\text{LEFT}, \text{RIGHT}\} \rightarrow \text{PAST}\}$-restriction: $\sigma_{24} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{24})$ where

$$\iota_{24}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a\} \\ \text{PAST} & \text{if } v \in \{e, f, g, i\} \\ v & \text{if } v \in \{b, c, d, h\} \end{cases}$$

25. Bag 25 is a forget bag which forgets vertex h. This has the following signature: $\sigma_{25} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{25})$ where

$$\iota_{25}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a\} \\ \text{PAST} & \text{if } v \in \{e, f, g, h, i\} \\ v & \text{if } v \in \{b, c, d\} \end{cases}$$

26. Bag 26 is a forget bag which forgets vertex d. This has the following signature: $\sigma_{26} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{26})$ where

$$\iota_{26}(v) = \begin{cases} \text{FUTURE} & \text{if } v \in \{a\} \\ \text{PAST} & \text{if } v \in \{d, e, f, g, h, i\} \\ v & \text{if } v \in \{b, c\} \end{cases}$$

27. Bag 27 is an introduce bag which introduces vertex a. This has the following signature: $\sigma_{27} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{27})$ where

$$\iota_{27}(v) = \begin{cases} \text{PAST} & \text{if } v \in \{d, e, f, g, h, i\} \\ v & \text{if } v \in \{a, b, c\} \end{cases}$$

28. Bag 28 is a forget bag which forgets vertex c. This has the following signature: $\sigma_{28} = (\{a, b, c, d, e, f, g, h, i\}, \phi, \iota_{28})$ where

$$\iota_{28}(v) = \begin{cases} \text{PAST} & \text{if } v \in \{c, d, e, f, g, h, i\} \\ v & \text{if } v \in \{a, b\} \end{cases}$$

29. Bag 29 is a forget bag which forgets vertex b. This has the following signature: $\sigma_{29} = (\{a, b, c, g, h, i\}, \phi, \iota_{29})$ where

$$\iota_{29}(v) = \begin{cases} \text{PAST} & \text{if } v \in \{b, c, g, h, i\} \\ v & \text{if } v \in \{a\} \end{cases}$$

The nodes d, e and f disappeared from the display graph due to the

$\{b \to \text{PAST}\}$-restriction which makes the arcs bd, be, df and eh redundant. When these arcs become redundant, the vertices d, e and f become redundant.

30. Bag 30 is a forget bag which forgets vertex a. This has the following signature: $\sigma_{30} = (\emptyset, \phi, \iota_{30})$, where $\iota_{30}(v)$ is the empty function. The lack of nodes is caused by the $\{a \to \text{PAST}\}$-restriction, which makes all remaining arcs and nodes redundant.

    This signature of the tree's root bag has an iso-labelling that has no vertices with the FUTURE label, so the algorithm terminates and returns *True*.

C Forget node

The forget node procedure first checks the redundancy of arcs adjacent to the removed node or part of any embedding paths of the removed node. Then, it checks the redundancy of endpoints of the redundant arcs. By only checking the local arcs and nodes, the procedure can be performed with a running time that is only dependant on the length of the embedding paths.

In the following pseudo code, the set of all redundant arcs in T is written as T_{ra} and the set of all redundant vertices in T is written as T_{rv}. Similarly, N_{ra} and N_{rv} denote the sets of redundant arcs and vertices in N respectively.

Algorithm 8 Forget Node

1: **procedure** FORGET_NODE($(D(N,T),\phi,\iota)$)
2: $signature_node \leftarrow \iota^{-1}(z)$
3: $isolabel(signature_node$, PAST$)$
4: $T_{rv}, N_{rv} \leftarrow local_redundancy_check((D(N,T),\phi,\iota), T_{ra}, N_{ra})$
5: **for** $a \in T_{rv}$ **do**
6: $signature.remove(a)$
7: **for** $u \in N_{rv}$ **do**
8: $signature.remove(u)$
return $(D(N,T)$
9: **procedure** LOCAL_REDUNDANCY_CHECK($(D(N,T),\phi,\iota)$)
10: $part_of_embedding \leftarrow$ True
11: **if** $z \notin tree_{IN}$ **then**
12: **if** $\exists\ y \in tree_{IN}$ such that $z = \phi(y)$ **then**
13: $z \leftarrow y$
14: **else**
15: $part_of_embedding =$ False
16: **if** $part_of_embedding =$ True **then**
17: **for** arc za adjacent to z **do**
18: **if** $\iota(a) = \iota(z)$ **then**
19: **if** $\iota(u) = \iota(z)\ \forall$ node $u \in \phi(za)$ **then**
20: Add za to tra_list
21: **for** arc $vw \in \phi(za)$ **do**
22: Add vw to nra_list
23: **else**
24: **for** arc $a'b'$ adjacent to z **do**
25: **if** $a'b'$ is part of an embedding **then**
26: $ab \leftarrow \phi^{-1}(a'b')$
27: **if** $\iota(a) = \iota(b)$ **then**
28: **if** $\iota(x) = \iota(a)\ \forall$ node $x \in \phi(ab)$ **then**
29: Add ab to tra_list
30: **for** arc $xy \in \phi(ab)$ **do**
31: Add xy to nra_list
32: **else**
33: **if** $\iota(u) = \iota(v)$ **then**
34: Add uv to nra_list
35: Add all elements of tra_list to T_{ra}
36: Add all elements of nra_list to N_{ra}
37: **for** node $a \in tra_list$ **do**
38: **if** $\iota(a) = \iota(\phi(a))$ **then**
39: **if** $bc \in T_{ra}\ \forall$ arc bc adjacent to a **then**
40: **if** $uv \in N_{ra}\ \forall$ arc uv adjacent to $\phi(a)$ **then**
41: Add a to T_{rv}
42: **if** a is not a leaf **then**
43: Add $\phi(a)$ to N_{rv}
44: **for** node $u \in nra_list$ **do**
45: **if** u is not part of an embedding **then**
46: **if** $vw \in N_{ra}\ \forall\ vw$ adjacent to u **then**
47: Add u to N_{rv}
return T_{rv}, N_{rv}

In this pseudo code, lines 9 - 36 find all redundant arcs, lines 37 - 41 find all redundant tree nodes and lines 42 - 47 find all redundant network nodes. When the forgotten node is in the network, finding the new redundant network arcs cannot be done by simply checking adjacent arcs. When the forgotten node is the final node of a embedding path to receive a PAST label, all arcs (including non-adjacent arcs) in the path become redundant. This is also shown in Figure 16.

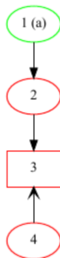


Figure 16: Part of a display graph in a signature. This signature has the embedding: $\delta = \{3 \rightarrow 3,\ 4 \rightarrow 1\}$. When a is forgotten and gets the PAST label, all arcs become redundant.

Hence, when local_redundancy_check is given a network node that is part of an embedding, it instead looks at the tree node that is embedded to it (lines 11-13). Afterwards, the procedure checks every adjacent tree arc according to the redundant tree arc definition. When a tree arc becomes redundant, all network arcs in its embedded path also become redundant.

When the forgotten node is from the network but does not have a tree node embedded to it, the procedure instead loops over all adjacent network arcs. These are then separated based on whether they are part of a path that has a tree arc embedded to it, and handled according to the redundant arc definitions.

When searching for redundant nodes, the procedure iterates through *tra_list* and *nra_list* instead of T_{ra} and N_{ra}. This is done since these lists only contain the new redundant arcs, and non-redundant nodes cannot become redundant nodes without having one of their adjacent arcs changed. Afterwards, tree nodes are checked based on the redundant tree node definition (lines 38-41). Network nodes that have a tree node embedded to them are found when the corresponding tree node is found (lines 42-43). Other network nodes are checked separately (lines 44-47).