PTV: Scalable Version Detection of Web Libraries and its Security Application – Technical Report

XINYUE LIU and LUKASZ ZIAREK, University at Buffalo, USA

This document provides proofs and time complexity analysis for algorithms introduced in the paper *PTV: Scalable Version Detection of Web Libraries and its Security Application.*

1 ALGORITHM DESIGN

In this section we provide the core algorithms and their complexity analysis that underpin our implementation of pTree-based JavaScript library detection.

1.1 Basic Definition

1.1.1 Labeled Tree. We denote a labeled tree as $T = (V, E, \Sigma, L)$, consisting of a *vertex* set V, an *edge* set E, an *alphabet* Σ for vertex labels, and a *labeling function* $L : V \longrightarrow \Sigma$. The *size* of T is the number of vertices in the tree.

A *path* is a sequence of vertices $p = (v_1, v_2, ..., v_n) \in V \times V \times ... \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. When the path's first vertex is root and the last vertex is a leaf, we call it a *full path.* For a tree T, we use $T.P$ to represent the set of all paths in T, and $T.P_f$ to represent the set of all full paths in T.

1.1.2 Induced Subtree. For a tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an *induced subtree* of T, denoted as $T' \preceq T$, if and only if (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) The labeling of V' is preserved in T'. If $T' \preceq T$, we also say that T *contains* T'. Intuitively, an induced subtree T' can be obtained by repeatedly removing leaf vertices in T, or possibly the root vertex if it has only one child.

We say two trees T_1 and T_2 are *isomorphic* to each other, denoted as $T_1 = T_2$, if there is a one-to-one mapping from the vertices of T_1 to the vertices of T_2 that preserves vertex labels and adjacency. Based on the definition, it is easy to see that relation $\preceq$ is antisymmetric and transitive, i.e., $T_1 \preceq T_2$ and $T_2 \preceq T_1$ implies $T_1 = T_2$; $T_1 \preceq T_2$ and $T_2 \preceq T_3$ implies $T_1 \preceq T_3$. We use symbol $T_1 \prec T_2$ when $T_1 \preceq T_2$ but $T_1 \neq T_2$.

1.2 Problem Description

We can generalize the version detection problem in the following description. Assume there is a detection object labeled tree ϕ and a collection of detection samples, represented as a set of labeled trees $\Gamma = \{T_1, T_2, ..., T_n\}$.

In our practical problem, Γ is a collection of generated pTrees from one library under different versions, and ϕ is the detected library pTree generated during web page runtime. Each vertex in the pTree will carry extra information – name, value, and type - represented as labels mapping to vertices.

We say a tree in Γ is the *base tree* of ϕ if ϕ is grown from it through adding root and leaf vertices. For simplicity, we define the predicate $B_\phi(T)$: tree $T \in \Gamma$ is the base tree of ϕ. Based on our definition, we can deduce that the base tree has the following two properties.

(1) **Necessity:** if $B_\phi(T)$, then $T \preceq \phi$;
(2) **Uniqueness:** exact one $T \in \Gamma$ satisfies $B_\phi(T)$.

The first principle introduces the necessary condition of the base tree. If T is a base tree of ϕ, then T has to be an induced subtree of ϕ. The second principle claims that only one sample tree is the base tree. We want to find the exact base tree that the detection object tree ϕ is built from. And this matches our practical situation – a loaded library should only have one version.

Besides, the detection object tree ϕ is restricted by the following property:

Proposition 1.2.1. *Assume T_k is the base tree, we have $\forall p \in \phi.P$, if $p \notin T_k.P$, then $p \notin \bigcup_{T\in\Gamma} T.P_f$.*

This property indicates that during the ϕ growing process, i.e., when more vertices are added to the base tree to build ϕ, the newly created paths will not be the full paths already in sample trees in Γ. Intuitively, this property ensures that ϕ is not a mixture of multiple trees in Γ, otherwise there is no way to uniquely determine the base tree. This property holds in our real-world detection task, because multiple versions of a library will be not loaded in the same place.

With these properties, the question is: given Γ and ϕ, how to find the $T \in \Gamma$, such that $B_\phi(T)$?

1.3 Simple Solution

First, in order to simplify later descriptions, here we make some additional definitions.

For two labeled trees T and T', if $T \preceq T'$, we say T' is a *supertree* of T; if $T \prec T'$, we say T' is a *strict supertree* of T. Given a tree set Γ, we use the symbol $\mathbb{S}_\Gamma(T)$ to represent the set of all supertrees of T contained in Γ, named *supertree set* (Typically the tree set Γ will be fixed and thus we will drop the dependence on Γ in our notation). In other words, $\mathbb{S}(T) = \{T' \in \Gamma \mid T \preceq T'\}$. Similarly, We use the symbol $\mathbb{S}_{st}(T)$ to represent the set of all strict supertrees of T.

We define the *equivalence class* of a tree T with respect to Γ as the set of all trees in Γ that is isomorphic to T, denoted as $[T]$, where $[T] = \{T' \in \Gamma | T' = T\}$. Easy to see that $[T] = \mathbb{S}(T) - \mathbb{S}_{st}(T)$. We provide an example for these definitions in Fig. 1.

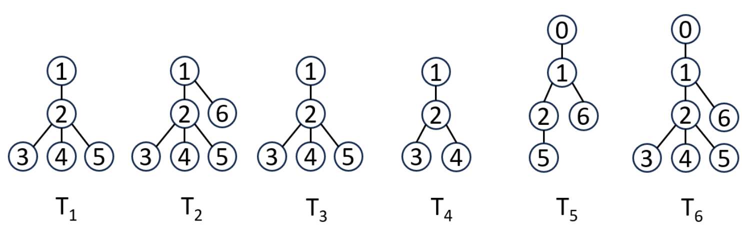

Fig. 1. Assume Γ consists of six trees in the plot, then we have $\mathbb{S}(T_1) = \{T_1, T_2, T_3, T_6\}$, $\mathbb{S}_{st}(T_1) = \{T_2, T_6\}$, $[T_1] = \mathbb{S}(T_1) - \mathbb{S}_{st}(T_1) = \{T_1, T_3\}$.

Furthermore, let's extend the predicate B_ϕ to B^*_ϕ when the predicate variable is a set of trees. For a tree set S, the $B^*_\phi(S)$ is defined as " $\exists\, T \in S$ such that $B_\phi(T)$"; and the $\neg B^*_\phi(S)$ is defined as " $\forall T \in S$, $\neg B_\phi(T)$". Based on this definition, we can reach two corollaries about B^*_ϕ.

Corollary 1.3.1. *For any two sets $S_1, S_2 \subseteq \Gamma$, if $B^*_\phi(S_2)$ and $\neg B^*_\phi(S_1)$, then $B^*_\phi(S_2 - S_1)$.*

Corollary 1.3.2. *For any two sets $S_1, S_2 \subseteq \Gamma$ that satisfy $S_1 \subseteq S_2$, if $\neg B^*_\phi(S_2)$, then $\neg B^*_\phi(S_1)$.*

Coro. 1.3.1 is due to the existence of the base tree – if the base tree does not exist in S_1, then it must be in $S_2 - S_1$. Coro. 1.3.2 describes that if the base tree does not exist in a set, then will not be in its subset as well. Both corollaries can be obtained directly from the definition of B^*_ϕ, so the proof is omitted. With these corollaries in hand, we can reach a vital proposition (Prop. 1.3.1), which enables us to determine which tree in Γ is the base tree through induced subtree judgment.

Lemma 1.3.1. *For an induced subtree T of ϕ, $B^*_\phi(\mathbb{S}(T))$.*

Proof. First let's prove that $\neg B^*_\phi(\Gamma - \mathbb{S}(T))$. Suppose $B^*_\phi(\Gamma - \mathbb{S}(T))$, which means that there exists a tree T' in Γ, such that $T \npreceq T'$ and $B_\phi(T')$. From $T \npreceq T'$, we know that there is a full path p of T not in the path set of T'. The lemma gives that T is an induced subtree of ϕ, so $p \in \phi.P$. Hence, p is a *path* satisfies all the conditions in Prop. 1.2.1 but contradicts its conclusion – $p \notin \bigcup_{T \in \Gamma} T.P_f$. As a result, $\neg B^*_\phi(\Gamma - \mathbb{S}(T))$. Then with Coro. 1.3.1, because $B^*_\phi(\Gamma)$, we have $B^*_\phi(\mathbb{S}(T))$. □

Lemma 1.3.2. *For a tree T which is not an induced subtree of ϕ, $\neg B^*_\phi(\mathbb{S}(T))$.*

Proof. Suppose $B^*_\phi(\mathbb{S}(T))$, which means that there exists a tree T' in Γ, such that $T \preceq T'$ and $B_\phi(T')$. According to the necessity, $T' \preceq \phi$, so T is an induced subtree of ϕ (transitivity). Contradiction. □

Proposition 1.3.1. *For an induced subtree T of ϕ, if $\forall T_s \in \mathbb{S}_{st}(T), T_s \npreceq \phi$, then $B^*_\phi([T])$; otherwise, $\neg B^*_\phi([T])$.*

Proof. If $\forall\, T_s \in \mathbb{S}_{st}(T)$, $T_s \npreceq \phi$, then $\neg B^*(T_s)$ (Necessity). So $\neg B^*_\phi(\mathbb{S}_{st}(T))$. We know that $B^*_\phi(\mathbb{S}(T))$ because $T \preceq \phi$ (Lemma. 1.3.2). Then, based on Coro. 1.3.1, we have $B^*_\phi(\mathbb{S}(T) - \mathbb{S}_{st}(T)) \Rightarrow B^*_\phi([T])$.

Otherwise, if there exits a tree T_s in $\mathbb{S}_{st}(T)$, such that $T_s \preceq \phi$. Then based on Lemma. 1.3.1, we know $B^*_\phi(\mathbb{S}(T_s))$. So $\neg B^*_\phi(\Gamma - \mathbb{S}(T_s))$. Consider that $T \prec T_s$, so $[T] \subseteq \Gamma - \mathbb{S}(T_s)$, thus $\neg B^*_\phi([T])$. □

Prop. 1.3.1 shows that we can determine whether the base tree exists in T's equivalence class by checking $T \preceq \phi$ and all its strict supertrees' $T_s \preceq \phi$. In T's equivalence class, each tree is isomorphic to the other, so there is no method to tell who is the base tree. Ensuring the base tree is in a specific equivalence class is a satisfactory result for our problem.

Informally, the algorithm to find the base tree in Γ can be described as follows: iterate all trees in Γ, for each tree $T \in \Gamma$, check whether $T \preceq \phi$ and whether every strict supertree of it satisfies $T_s \preceq \phi$. Combined with Prop. 1.3.1, the result can be $B^*_\phi([T])$ or $\neg B^*_\phi([T])$. If the former is the case, then the algorithm terminates and the output is $[T]$. This algorithm ensures that the equivalence class in which the base tree is located will be found.

1.4 Unique Subtree Mining

Although we have given a deterministic algorithm to find the base tree (informally), in our practical application scenarios, the sample trees (trees in Γ) are usually large and numerous. If the algorithm in the previous section is used for runtime detection, the time and space costs are unaffordable. As a result, in this section, we propose an algorithm to minify sample trees' size by unique subtree mining and ensure that the previous algorithm is still valid.

Algo. 1 shows the overall algorithm to reduce the size of sample trees. The input to the algorithm is the sample tree set Γ. For each $T \in \Gamma$, the output is its supertree set $\mathbb{S}(T)$ and a unique subtree T_m. We envision that T_m is a tree with a smaller size than T but the supertree set does not change. Namely, $\mathbb{S}(T_m) = \mathbb{S}(T)$. Note that the T in Prop. 1.3.1 is not required to be a member of the set Γ; this proposition applies whenever $T_m \preceq \phi$, so we can use T_m to replace the original T during the runtime detection. If $\mathbb{S}(T_m) = \mathbb{S}(T)$, then T_m must be an induced subtree of T, otherwise $T \in \mathbb{S}(T)$ while $T \notin \mathbb{S}(T_m)$. Therefore, our algorithm generates the specific T_m for each $T \in \Gamma$ by using a subset of the tree paths to reconstruct an induced subtree that satisfies $\mathbb{S}(T_m) = \mathbb{S}(T)$.

For each sample tree, Algo. 1 first calculates the path recording, whose details are presented in Algo. 2. In Algo. 2 line 1, we initialize the recording collection Ω as an empty set. For each full path in tree T, we use a recording set ω to record the path's occurrence in other trees in Γ (Algo. 2 line 3). If the full path occurs in the path set of another tree T_i, the tree's index i will be recorded in ω.

Algorithm 1 Unique Subtree Mining

Input: the sample tree set: Γ
Output: $\mathbb{S}(T)$ and the unique subtree T_m for each $T \in \Gamma$

```
1: for each T ∈ Γ do
2:     Ω ← PathRecording (T, Γ)
3:     S(T) ← ⋂_{ω∈Ω} ω
4:     Let Ω̄ := {Γ − ω | ω ∈ Ω}
5:     I ← MinCoverSet (Ω̄, Γ − S(T))
6:     T_m ← BuildTreeFromPath (T, I)
7: end for
```

Here we choose the full path instead of the normal path because we want to make sure the size of the generated T_m is large enough to prevent false positives during detection. In line 5, we combine all the recording set ω of each full path together as a recording collection Ω.

Algorithm 2 PathRecording

Input: the target set: Γ, a tree: $T \in \Gamma$
Output: a coloring collection Ω

```
Initialization: Ω ← ∅
for each full path f in T.P_f do
    f's coloring set ω := {T_i ∈ Γ | f ∈ T_i.P}
    Ω ← Ω ∪ {ω}
end for
```

After getting the recording collection Ω, in Algo. 1 line 3, the value of $\mathbb{S}(T)$ is obtained by intersecting all elements in the Ω.

Proposition 1.4.1. $\bigcap \Omega = \mathbb{S}(T)$.

Proof. If a tree $T' \in \Gamma$ is in $\bigcap \Omega$, then all the full paths of T is contained in the path set of T', so $T \preceq T'$; otherwise, at least one full path of T is not in the path set of T', so $T \npreceq T'$. As a result, $\bigcap \Omega$ equals the set of all supertrees of T. □

The unique subtree T_m is generated from a subset of full paths of T. To ensure $\mathbb{S}(T_m) = \mathbb{S}(T)$, we need to find the smallest subset of Ω, such that the intersection of all its elements still equals $\mathbb{S}(T)$. If we complement both sides of the equation in Prop. 1.4.1, we can get $\bigcup_{\omega\in\Omega}(\Gamma - \omega) = \Gamma - \mathbb{S}(T)$ by De Morgan's laws. In this form, our question is equivalent to a well-known NP-complete problem – the *set cover problem* – which is described as follows.

> Given a set of elements $\{1, 2, \ldots, n\}$ (called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe.

In our algorithm, the collection S in the description of the set cover problem is the inversed recording collection $\overline{\Omega}$ defined in Algo. 1 line 4; and the universe U is $\Gamma - \mathbb{S}(T)$. In line 5, we invoke Algo. 3 to calculate the minimum cover subset of $\overline{\Omega}$. This algorithm will return an index set I, which contains the index of all elements that constitute the minimum cover subset. Using these indexes, in line 6, we construct the unique subtree T_m by invoking Algo. 4.

Algo. 3 is a famous greedy algorithm to solve the set cover problem within approximate polynomial time. At each stage, it chooses the set with the largest number of uncovered elements. This

Algorithm 3 MinCoverSet

Input: a set collection: $S = \{\omega_1, \omega_2, ..., \omega_n\}$, the universe U
Output: a set $I \subseteq \{1, 2, ..., n\}$, such that $\bigcup_{i \in I} \omega_i = U$

1: Initialization: $I \longleftarrow \varnothing$, $C \longleftarrow \varnothing$
2: **while** $C \neq U$ **do**
3: Find the $i \in \{1, 2, ..., n\} - I$, such that $|C \cup \omega_i|$ is largest
4: $I \longleftarrow I \cup \{i\}$
5: $C \longleftarrow C \cup \omega_i$
6: **end while**

algorithm achieves an approximation ratio of $H(s)$, where s is the size of the set to be covered. In other words, it finds a set covering that may be $H(n)$ times as large as the minimum one, where $H(n)$ is the n-th harmonic number:

$$H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \tag{1}$$

Algorithm 4 BuildTreeFromPath

Input: a tree T with a full path set $T.P_f = \{p_1, p_2, ..., p_k\}$, an index set $I \subseteq \{1, 2, ..., k\}$
Output: the unique subtree T_m

1: Initialization: $T_m \longleftarrow \varnothing$
2: **for** each $i \in I$ **do**
3: Add path p_i to the tree T_m
4: **end for**

Algo. 4 shows the detail of T_m constructing. The input to the algorithm is a tree T and an index set I. We select the path whose index appears in the index set I to construct the tree.

Lastly, let's use the trees in Fig. 1 to illustrate the whole unique subtree mining process. Here $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$. For T_1, firstly, we calculate the recording of each full path of it in Algo. 2. There are three full paths and the corresponding recording sets ω of T_1 shown in Table 1. Then we can get the value of $\mathbb{S}(T_1)$ by union all the ω. And the inversed recording collection $\overline{\Omega} = \{\{T_5\}, \{T_5\}, \{T_4\}\}$. The Algo. 3 helps us to find the minimum sets from collection $\overline{\Omega}$ whose union equals $\Gamma - \mathbb{S}(T_1) = \{T_4, T_5\}$. And we can see that the combination of path $(1, 2, 3)$ and $(1, 2, 5)$ can meet the requirement. As a result, the unique subtree $(T_1)_m$ consists of these two paths. Similarly, we can get other unique subtrees. All unique subtrees are shown in Fig. 2.

Table 1. Unique subtree mining algorithm calculation result on T_1 in Fig. 1.

full paths	ω	$\mathbb{S}(T_1)$	$\overline{\Omega}$
$(1, 2, 3)$	$\{T_1, T_2, T_3, T_4, T_6\}$	$\{T_1, T_2, T_3, T_6\}$	$\{\{T_5\}, \{T_5\}, \{T_4\}\}$
$(1, 2, 4)$	$\{T_1, T_2, T_3, T_4, T_6\}$		
$(1, 2, 5)$	$\{T_1, T_2, T_3, T_5, T_6\}$		

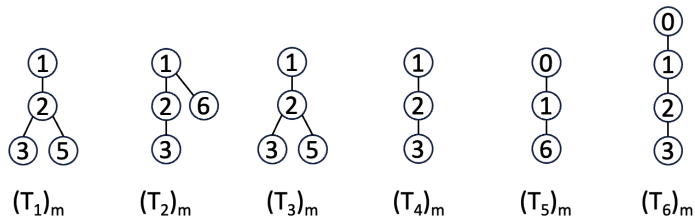

Fig. 2. The unique subtrees of trees in Fig. 1.

1.5 Strict Supertree Set Minify

In Prop. 1.3.1 we need to verify all strict supertrees of T to determine whether the base tree is located in $[T]$; however, it is not necessary to iterate through the whole $\mathbb{S}_{st}(T)$. In Sec. 1.6, we will prove that the trees in the minified strict supertree set $\mathbb{S}_m(T)$, which is a subset of $\mathbb{S}_{st}(T)$, are all we need to check. Algo. 5 shows the process of generating $[T]$ and $\mathbb{S}_m(T)$ for each $T \in \Gamma$.

Algorithm 5 Strict Supertree Set Minify

Input: a tree T with its supertree set $\mathbb{S}(T)$
Output: $[T]$ and the minified strict supertree set $\mathbb{S}_m(T)$

1: Initialization: $[T] \leftarrow \varnothing$, $\mathbb{S}_m(T) \leftarrow \varnothing$
2: **for** each supertree $T' \in \mathbb{S}(T)$ **do**
3:    **if** $T \in \mathbb{S}(T')$ **then**
4:       $[T] \leftarrow [T] \cup \{T'\}$
5:    **end if**
6: **end for**
7: $\mathbb{S}_{st}(T) := \mathbb{S}(T) - [T]$
8: **while** $\mathbb{S}_{st}(T) \neq \varnothing$ **do**
9:    Find $K \in \mathbb{S}_{st}(T)$, such that $|\mathbb{S}(K)|$ is the largest
10:    $\mathbb{S}_{st}(T) \leftarrow \mathbb{S}_{st}(T) - \mathbb{S}(K)$
11:    $\mathbb{S}_m(T) \leftarrow \mathbb{S}_m(T) \cup \{K\}$
12: **end while**

In Algo. 5 line 1 - 6, we calculate $[T]$. The idea is simple: for a supertree of T, if $T \in \mathbb{S}(T')$ and $T' \in \mathbb{S}(T)$, then $T = T'$. Next, we get the value of $\mathbb{S}_{st}(T)$ in line 7 by removing isomorphic supertrees from $\mathbb{S}(T)$. From line 8, we start to generate the minified strict supertree set $\mathbb{S}_m(T)$. The algorithm always selects the element of $\mathbb{S}_{st}(T)$ that has the largest number of supertrees. This greedy algorithm ensures that $\mathbb{S}_m(T)$ holds an important proposition – Prop. 1.5.1, which can help $\mathbb{S}_m(T)$ to replace $\mathbb{S}(T)$ in runtime detection as we will discuss in Sec. 1.6.

Lemma 1.5.1. *If $T \preceq T'$, then $\mathbb{S}(T') \subseteq \mathbb{S}(T)$.*

Proof. $\forall t \in \mathbb{S}(T')$, based on the definition of the supertree set, we know $T' \preceq t$. According to transitiveness, $T \preceq T' \preceq t$, so $t \in \mathbb{S}(T)$. Therefore, $\mathbb{S}(T') \subseteq \mathbb{S}(T)$. □

Proposition 1.5.1. *$\mathbb{S}_m(T)$ is the smallest subset of $\mathbb{S}_{st}(T)$ that satisfies $\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K) = \mathbb{S}_{st}(T)$.*

Proof. We prove this using the greedy algorithm proof scheme.

(1) (Greedy Choice Property) Our greedy choice is K whose supertree set size is the largest in $\mathbb{S}_{st}(T)$. Suppose there is an optimal solution O that does not contain K. Because $K \in \mathbb{S}_{st}(T)$, there exists an K' in solution O such that $K \in \mathbb{S}(K')$; otherwise it can't satisfy $\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K) = \mathbb{S}_{st}(T)$.

So, $K' \preceq K$. Based on Lemma. 1.5.1, we know $\mathbb{S}(K) \subseteq \mathbb{S}(K')$. In our greedy choice, $|\mathbb{S}(K)|$ is the largest, so $\mathbb{S}(K) = \mathbb{S}(K')$. Hence, we can replace K' by K in O and still get an optimal solution.

(2) (Optimal Substructure Property) Let O be an optimal solution containing K. Consider the subproblem $\mathbb{S}'_{st}(T) = \mathbb{S}_{st}(T) - \mathbb{S}(K)$. We need to prove O contains the optimal solution for $\mathbb{S}'_{st}(T)$. Suppose $O - \{K\}$ is not an optimal solution for $\mathbb{S}'_{st}(T)$. We denote the optimal solution for $\mathbb{S}'_{st}(T)$ by O'. Then $|O'| < |O - \{K\}| = |O| - 1$. Given that $(\bigcup_{K \in O'} \mathbb{S}(K)) \cup \mathbb{S}(K) = \mathbb{S}_{st}(T)$, $O' \cup \{K\}$ is a solution with a smaller size than O. Hence, O is not an optimal solution. Contradiction. □

Take the trees in Fig. 1 as an example. The value of $\mathbb{S}_m(T_1)$ should be $\{T_2\}$, because $\mathbb{S}(T_2) = \{T_2, T_6\} = \mathbb{S}_{st}(T_1)$. Similarly, we have $\mathbb{S}_m(T_2) = \{T_6\}$, $\mathbb{S}_m(T_3) = \{T_2\}$, $\mathbb{S}_m(T_4) = \{T_1\}$, $\mathbb{S}_m(T_5) = \{T_6\}$, and $\mathbb{S}_m(T_6) = \varnothing$.

1.6 Runtime Detection

So far, for each $T \in \Gamma$, we get its unique subtree T_m in Sec. 1.4 and its minified strict supertree set $\mathbb{S}_m(T)$ in Sec. 1.5. Now, we can rewrite Prop. 1.3.1 in the following new version.

Proposition 1.6.1. *If $T_m \preceq \phi$ and $\forall\, K \in \mathbb{S}_m(T)$, $K_m \npreceq \phi$, then $B^*_\phi([T])$; otherwise, $\neg B^*_\phi([T])$.*

Proof. We divide the condition into three cases.

(1) $T_m \npreceq \phi$.

From Lemma. 1.3.2, we have $\neg B^*_\phi(\mathbb{S}(T_m))$. Due to $[T] \subseteq \mathbb{S}(T) = \mathbb{S}(T_m)$, we have $\neg B^*_\phi([T])$.

(2) $T_m \preceq \phi$, and $\exists\, K \in \mathbb{S}_m(T)$, such that $K_m \preceq \phi$.

From Lemma. 1.3.1, $B^*_\phi(\mathbb{S}(K_m))$. Because $\mathbb{S}(K_m) = \mathbb{S}(K)$, we have $B^*_\phi(\mathbb{S}(K))$, so $\neg B^*_\phi(\Gamma - \mathbb{S}(K))$. From the definition of $\mathbb{S}_m(T)$, we know $T \prec K$, so $[T] \subseteq \Gamma - \mathbb{S}(K)$. Therefore, $\neg B^*_\phi([T])$.

(3) $T_m \preceq \phi$, and $\forall K \in \mathbb{S}_m(T)$, $K_m \npreceq \phi$.

Based on Lemma. 1.3.2, we can get $\forall K \in \mathbb{S}_m(T)$, $\neg B^*_\phi(\mathbb{S}(K_m))$, then $\neg B^*_\phi(\mathbb{S}(K))$. So we have $\neg B^*_\phi(\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K))$, and this can be converted to $\neg B^*_\phi(\mathbb{S}_{st}(T))$ by Prop. 1.5.1. Furthermore, because $T_m \preceq \phi$, by Lemma. 1.3.1, we have $B^*_\phi(\mathbb{S}(T_m))$, thus $B^*_\phi(\mathbb{S}(T))$. Consequently, here comes $B^*_\phi(\mathbb{S}(T) - \mathbb{S}_{st}(T)) \Rightarrow B^*_\phi([T])$. □

Based on Prop. 1.6.1, given Γ and ϕ, we propose an algorithm to detect the base tree of ϕ in Γ, shown in Algo. 6. Let's say the original sample tree set is $\Gamma = \{T_1, T_2, ..., T_n\}$. Then the first input to the algorithm is a unique subtree set $\Gamma_m = \{(T_1)_m, (T_2)_m, ..., (T_n)_m\}$. We represent the indexes of the trees in Γ_m as $I = \{1, 2, ..., n\}$. Then, we define two mappings f_s and $f_e : I \longrightarrow \mathcal{P}(I)$, where for an index $k \in I$, $f_s(k)$ maps to the set of all index of trees in $\mathbb{S}_m(T_k)$, and $f_e(k)$ maps to the set of all index of trees in $[T_k]$. Namely, $f_s(k) = \{i \mid T_i \in \mathbb{S}_m(T_k)\}$, and $f_e(k) = \{i \mid T_i \in [T_k]\}$. The last input to the algorithm is the detect object tree ϕ.

Algo. 6 guarantees to return the equivalence class of the base tree in Γ – it will traverse all equivalence classes to find the one that meets the condition in Prop. 1.6.1. Note that this algorithm does not require the original sample trees Γ as input, resulting in faster speed and less space occupied during the detection runtime.

1.7 Algorithm Complexity

It is obvious that most part of the algorithm is in trivial linear time complexity. In this section, we only discuss two non-trivial parts – path recording (Algo. 2) and minimum cover set (Algo. 3). Suppose there are n trees in Γ, and N vertices in Γ.

1.7.1 Path recording. To get path recording, Algo. 2 iterates through all full paths in the tree and check whether these full paths appear in the path set of other trees in Γ. In our application,

Algorithm 6 Runtime Detection

Input: the unique subtrees $\Gamma_m = \{(T_1)_m, (T_2)_m, ..., (T_n)_m\}$, two mappings f_s, f_e, and the detect object tree ϕ

Output: the indexes of possible base trees

```
1:  for each i ∈ [n] do
2:      if (T_i)_m ⪯ φ then
3:          for each j ∈ f_s(i) do
4:              if (T_j)_m ⪯ φ then
5:                  go to 9
6:              end if
7:          end for
8:          return f_e(i)
9:      end if
10: end for
```

all the tree in Γ share a same root "window", and the "window" vertex will not appear at other places except root. Hence, given a full path f and a tree T, we only need at most $|f|$ times vertex comparisons to find out whether $f \in T.P$, where $|f|$ represents the number of vertices on the path f. Given a tree $T_1 \in \Gamma$, the time to calculate the path recording of T_1 is:

$$n \cdot \sum_{f \in T_1.P_f} |f| \tag{2}$$

Observe that the number of full paths in a tree is no more than its vertex number, and the vertex number of any full path is no more than tree's vertex number either. We have:

$$n \cdot \sum_{f \in T_1.P_f} |f| \leq n \cdot |T_1.P_f| \cdot |T_1.V| \leq n \cdot |T_1.V|^2 \tag{3}$$

Hence, the time to calculate the path recording for all trees in Γ is:

$$\begin{aligned} T(\text{path recording}) &= n \cdot \sum_{f \in T_1.F} |f| + n \cdot \sum_{f \in T_2.F} |f| + \cdots + n \cdot \sum_{f \in T_n.F} |f| \\ &\leq n \cdot \sum_{T \in \Gamma} |T.V|^2 \leq n \cdot (\sum_{T \in \Gamma} |T.V|)^2 = n \cdot N^2 \end{aligned} \tag{4}$$

So the time complexity of path recording algorithm is $O(n \cdot N^2)$.

1.7.2 Minimum Cover Set. In Algo. 2, the size of set collection S equals the number of full paths. In each iteration, algorithm traverse all elements in S, and there are at most $|S|$ iterations. So, for a tree T, it requires at most $|S|^2$ operations to find the minimum cover set. The time to calculate the minimum cover set for all trees in Γ is:

$$T(\text{Minimum Cover Set}) = \sum_{T \in \Gamma} |T.P_f|^2 \leq \sum_{T \in \Gamma} |T.V|^2 \leq (\sum_{T \in \Gamma} |T.V|)^2 = N^2 \tag{5}$$

So the time complexity of minimum cover set algorithm is $O(N^2)$.

In conclusion, the overall tree processing algorithm has $O(n \cdot N^2)$ worst-case time complexity, where n is the number of tree, and N is the total number of vertex of all trees.