Algorithm for k-truncated metric dimension of trees

by

Thomas Molendijk

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 24, 2023 at 10:00 AM.

Student number:	3563357	
Project duration:	April 25, 2023 – August 17, 2023	
Thesis committee:	dr. J. Komjáthy,	TU Delft, supervisor
	dr. A. Bishnoi,	TU Delft, supervisor
	dr. M. Jones,	TU Delft

Abstract

The k-truncated metric dimension of a graph is the minimum number of sensors (a subset of the vertex set) needed to uniquely identify every vertex in the graph based on its distance to the sensors, where the sensors have a measuring range of k. We give an algorithm with the goal that given any tree and any value for the measuring range of the sensors k, the algorithm finds the k-truncated metric dimension of that tree. The algorithm presented in this thesis is a modification of the algorithm given by Gutkovich and Song Yeoh [6]. The algorithm in this thesis improves on their algorithm in both validity and time complexity. We show that given any tree and any value k, the algorithm returns a k-resolving set for that tree. Moreover, we conjecture the difference in the k-truncated metric dimension of any tree and the size of the k-resolving set found by the algorithm for that tree is never greater than one. The time complexity of the algorithm is proven to be $\mathcal{O}(k^3 n)$, where k is the measuring range of the sensors and n is the number of vertices in the tree. This implies that the time complexity is linear in n for fixed k.

1 Introduction

This section provides an introduction to the problem researched in this thesis, motivates research of this problem and gives a mathematical overview of related problems.

1.1 Problem description

The *metric dimension* of a graph is a concept first introduced by Slater in 1975 [23] to solve a source-location problem. The problem is as follows. Given a simple, undirected graph $G = (V, E)$, we want to place *sensors* on a subset $S \subseteq V$ such that we can uniquely identify each vertex $v \in V$ by its distance vector to the sensors $(d(v, s))_{s \in S}$. We call such a set of sensors S *resolving*. The metric dimension of a graph is the cardinality of the smallest resolving set S. Slater later expanded on his idea of the metric dimension and introduced *locating-dominating sets* [24]. The definition is similar; a locating-dominating set is a subset $S \subseteq V$ which measures each vertex $v \in V(T)$ and can uniquely identify each vertex based on the measurements from the sensors, but now the measuring range of the sensors is limited to one, meaning a sensor can only measure vertices adjacent to the sensor. In the same article, Slater presented a linear ($\mathcal{O}(n)$, $n = |V|$) algorithm for finding the smallest locating-dominating set in a tree. Originally, Slater envisioned his ideas would be used for safeguards analysis of facilities such as nuclear power plants, but recently computer scientists have found new applications of these ideas in the detection of malware sources, reviving interest in the subject. These new applications require a generalisation in the allowed values for the measuring range (k) of the sensors, evolving the locating-dominating set into *k-resolving* sets. A k-resolving set (also referred to as a resolving set, when the measuring range of the sensors is clear from context) is similar to a locating-dominating set, but with sensors with a measuring range k, where k can be any natural number. The smallest k-resolving set of a tree is called that tree's k-truncated metric dimension. For practical purposes, an algorithm which efficiently finds the k-truncated metric dimension is needed.

Figure 1: Above are two examples to illustrate the requirements for a set to be a k-resolving set, with $k = 2$. The yellow, square vertices have sensors placed on them. In the left graph, the sensor does not form a k-resolving set, because the sensor can not distinguish between the two adjacent vertices. In the right graph, the sensor does form a k-resolving set, because it measures the two vertices as being at different distances.

1.2 Motivation

The mathematical field of source detection is a large and active field of research. Due to their ambiguous nature, source detection problems have a wide range of applications, such as navigation [13], locating the source of rumors in a social network [22], locating a contaminant in water distribution networks [21] or finding the origin of a virus in social- and digital networks [25].

The truncated metric dimension of a graph is one of the many variants of the metric dimension of graphs [14]. Finding an efficient algorithm for the truncated metric dimension of a family of graphs, such as trees, can give insight into how the metric dimension, or any variants of the metric dimension, behave on that family of graphs, or graphs which are similar to that family.

1.3 Related problems, results and mathematical overview

In a recent article, Gutkovich and Song Yeoh gave an algorithm for computing the truncated metric dimension of a tree for fixed k [6]. Neither the validity nor the time complexity of their algorithm was proved. In this thesis we modify their algorithm in order to improve it in both validity and time complexity.

Finding the metric dimension of general graphs is NP-Hard [13]. Nevertheless, the metric dimension of different families of graphs has been studied extensively [9, 12, 8, 15]. Moreover, many variants of the metric dimension have been studied, such as fault-tolerating metric dimension [7], k-metric dimension [4] and local metric dimension [14].

The concept of *r-locating dominating codes* (also known as k-metric dimension) is similar to the k-truncated metric dimension, in that every vertex must be uniquely identified by the sensors in the graph while placing the minimum number of sensors. The difference between the truncated metric dimension and r-locating dominating codes is that for r-locating dominating codes, the sensors do not reveal their distance to any vertex which they measure. The only signal the sensors can give is whether they measure a vertex or not. The concept was first introduced by Karpovsky et al. in 1998 to find faults in multiprocessor systems [11].

Recently, Bartha, Komjáthy and Raes disproved a conjecture by Tillquist et al. on the lower bound of the k-truncated metric dimension for a tree of size n. The conjecture stated that the maximum size of a tree with m sensors and a measuring range of k is $\Theta(\frac{mk^2}{4})$. In their article, they proved a lower bound which is not only larger than previously conjectured ($\Theta(\frac{mk^2}{3})$), but also sharp, and gave a construction for an optimal graph [2].

Stochastic source-detection in graphs is a large field of research. As opposed to deterministic source detection problems, in stochastic source-detection the phenomenon which spreads through the network is not assumed to spread in a deterministic manner. This makes the stochastic approach highly applicable to studying the source of a virus or misinformation spreading in social- or digital networks [25]. Early research assumed the underlying network sufficed the conditions to be a tree-network and used the standard SI-model to model how the phenomenon spreads [20, 10, 16, 17]. Later, other models for disease spread were used in order to study source-detection problems for other types of phenomenon which spread in a way which the SI-model is not suited for, such as the SIR-model [28, 3] and the SEIR-model [27, 19]. Recent research is focused on developing source-detection methods which are effective on general graphs, i.e. without assuming the underlying network is a tree [5, 18].

2 Preliminaries

In the thesis, only rooted trees will be considered. In some definitions we mention the distance between two points in a graph, by which we mean the graph distance. We give definitions necessary to understand this thesis in the following subsection.

2.1 Definitions

First, we give a formal definition of the concept which is central to this thesis.

Definition 2.1. *Fix $k \in \mathbb{N}$. Define $d_k(\cdot,\cdot) = min(d(\cdot,\cdot), k+1)$. A* k-resolving set *for a graph with vertex set V is a subset $S \subseteq V$ such that $\forall v, w \in V \setminus S, \exists s \in S$ such that $d_k(v,s) \neq d_k(w,s)$.*

Note that taking $k = 1$ in this definition makes it identical to the definition of locating-dominating set as given by Slater [23].

Definition 2.2. *The* k-truncated metric dimension *of a graph is the cardinality of the smallest* k*-resolving set of that graph.*

Definition 2.3. *P_{xy} is the ordered set of vertices on the shortest path from x to y. The set is ordered from smallest distance to x to largest distance to x. In this thesis we only consider (rooted) trees, so this path is unique.*

Definition 2.4. *In a tree with root v, we call a vertex u the* parent *of a vertex w if $u \in P_{wv}$ and $d(v, u) = 1$. We call w the* child *of u. The parent of a vertex v will be denoted as $p(v)$.*

The algorithm iterates over the vertices in the order of the endvertex list of the tree. We give the defintion of an endvertex list.

Definition 2.5. *Let T be a tree with n vertices. Then $(v_1, v_2, ..., v_n)$ is an* endvertex list *for T if $\{v_1, v_2, ..., v_n\} = V(T)$ and for $1 \leq i \leq p - 1$ each v_i is adjacent to at most one v_j with $j > i$. An endvertex list is a listing of the vertex set V(T) in which each vertex appears before its parent.*

Definition 2.6. *For the given endvertex list of some tree with root v_n, $EL = (v_1, v_2, ..., v_n)$, the associated* adjacency list *$AL = (u_1, u_2, ..., u_n)$ satisfies $u_i = v_i$ if v_i is the root of the tree and u_i is equal to the parent of v_i otherwise.*

2.2 Results

We present the result of this thesis in the following theorem.

Theorem 2.7. *Fix $k \in \mathbb{N}$. Given a tree T of size n, there is an $\mathcal{O}(n)$ algorithm which finds a k-resolving set for the tree T.*

Proof. Combining Proposition 4.1 with proposition 5.7, it becomes clear the statement in the theorem holds. □

Optimality of the algorithm presented in this thesis has not been proved yet, therefore we conjecture that for any tree, the size of the output of the algorithm is always at most one greater than the truncated metric dimension of that tree.

Conjecture 2.8. *Fix $k \in \mathbb{N}$. Given a tree T of size n, there is an $\mathcal{O}(n)$ algorithm which finds a k-resolving set for T of size at most one greater than the truncated metric dimension of T.*

The following section explains the algorithm in detail.

3 Algorithm for k-truncated metric dimension

We give a heuristic explanation of the algorithm before we introduce the specific items the algorithm keeps track of or any of its operations.

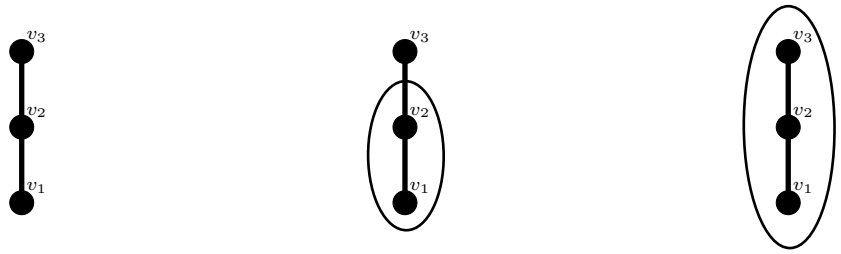
3.1 Heuristic explanation of the algorithm

In principal, the idea behind the algorithm for the k-truncated metric dimension of trees is simple; delay placing a sensor for as long as possible. The complicating factor in the case of general (fixed) k is that there are many options to measure a vertex, namely, all vertices within distance k. This issue is resolved by placing sensors at the absolute latest moment possible, and then doing a case analysis to determine which of the remaining options is optimal.

The algorithm iterates over an *endvertex list* (Definition 2.5) and its associated *adjacency list* (Definition 2.6). This means the algorithm will start processing the tree from the bottom, and it will process all children of a vertex, before processing that vertex. Note that an endvertex list does not require any special ordering

of vertices from the same 'generation', so the algorithm might process a vertex in one part of the tree one step, and process a vertex in an entirely different part of the tree the next. By 'processing' a vertex, we mean the algorithm is deciding to place a sensor on that vertex or not. After a vertex is processed, its status from sensor to non-sensor will never change.

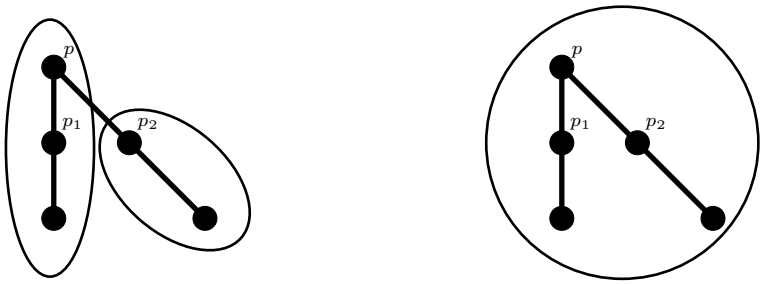
The algorithm processes the tree from the bottom up and it has no knowledge of the structure of the tree before starting to process the tree. Every time the algorithm processes a vertex it expands its 'view' of the structure of the tree by looking at the parent vertex of the vertex it is processing. When processing a vertex, it does not handle that vertex as if it is independent from the rest of the tree. Instead, it sees that the vertex it is processing is attached to a part of the tree which the algorithm has processed earlier (due to the ordering in the endvertex list, Definition 2.5), and it will decide whether to place a sensor on the vertex it is processing or not based on information from that earlier processed part of the tree. Therefore, it is important for the algorithm to know which vertices are in the same part of the tree.



(a) (b) (c)

Figure 2: Pictured in (a), (b) and (c) is a rooted tree with root vertex v_3. Vertices which the algorithm views as being connected, i.e. in the same part of the tree are in the same circle. In (a), the algorithm has not processed any part of the tree yet, therefore it has no knowledge of the structure of the tree and no vertices are viewed as being connected. In (b), the algorithm has processed the leaf, v_1. When processing v_1 the algorithm looked up at the parent of v_1, v_2, and now sees they are connected. In (c), the algorithm has processed v_2. Again, the algorithm looked up at the parent of v_2, v_3, and now sees they are connected.

In any tree which is not a path, there is some vertex with multiple children, call such a vertex p. When the algorithm processes the first child of p, it expands its view of the tree to include p, as shown in Figure 2. Subsequently, when the algorithm processes the second child of p, it expands its view of the tree to include p again, but p is already seen as a part of the tree, therefore the algorithm must *merge* these two parts of the tree which it did not view as being connected into one single, connected part of the tree.



(a) (b)

Figure 3: Vertices which the algorithm views as being connected are drawn in the same circle. In (a), the algorithm has processed p_1, so it now sees p is the parent of p_1. It has also processed the child of p_2, so it views p_2 and its child as being connected. In (b), we draw how the algorithm views the tree after processing p_2. While processing p_2, it looked to the parent of p_2, p, and saw the parent was already known to be connected to some part of the tree. Therefore these two parts of the tree which the algorithm did not view as being connected are in fact connected, and the algorithm merges its knowledge of the two parts of the tree together into one single part of the tree.

At every step of the algorithm, it is maintained that the sensors in any part of the tree which the algorithm has processed either fully resolve that part of the tree or 'almost' resolve that part of the tree. With 'almost' resolve, we mean to say exactly one more sensor is required to resolve that part of the tree. The algorithm keeps track of exactly how far that sensor would need to be placed. This property of the algorithm is what will be called the *fundamental property/requirement.*

3.2 Notation

Before explaining the algorithm in detail, it is necessary to define the notation used in this thesis.

Definition 3.1. *Given a tree T, we will refer to a vertex x as being* measured through *another vertex y if for all sensors s in the tree T which measure x, the shortest path from x to s passes through y, i.e. if for all sensors $s \in T$ such that $|P_{xs}| \leq k$, $y \in P_{xs}$.*

Definition 3.2. *A vertex v is* processed *when the algorithm has iterated over that vertex in the endvertex list. In practice, this means the status of v as sensor or non-sensor will never change.*

Earlier in our explanation of the algorithm, we referred to 'part of the tree'. For ease of writing we introduce a notation to specify what 'part of the tree' we mean exactly.

Definition 3.3. *A subgraph of a tree which is also a tree will be called a* subtree. *Let T be a tree with root r. Let v be a vertex in T and let $m \in \{1, 2, ..., ch(v)\}$. Then $T_{v,m} \subset T$ is the* subtree *with root v, which contains v and all vertices $x \in T$ such that one of the first m children of v is on the path from x to r.*

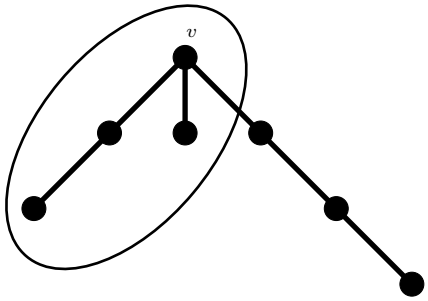


Figure 4: The subtree inside the circle is $T_{v,2}$, because it contains v and all vertices for which one of the first two children of v is on the path from that vertex to v.

Definition 3.4. *Let $T_{v,m}$ be a subtree. Let $x, y \in V(T_{v,m})$. We define $d_k(x,y) = min(d(x,y), k+1)$.*

Definition 3.5. *We define $ch(v)$ as the number of children of a vertex v.*

Definition 3.6. *In the subtree $T_{v,m}$, we define the* pivotal vertex *of x, $w(x,v)$, as the vertex at which the paths P_{xv} and P_{xs} split for all sensors s in $T_{v,m}$ which measure x. A pivotal vertex may not exist, for instance when two sensors s_1, s_2 in $T_{v,m}$ measure x but P_{vx} and P_{xs_1} do not split at the same vertex as P_{vx} and P_{xs_2}.*

$$\forall s_1, s_2 \in S(v,m) \text{ s.t. } |P_{xs_1}|, |P_{xs_2}| \leq k : P_{xv} \cap P_{xs_1} = P_{xv} \cap P_{xs_2} := P_{xw(x,v)}$$

Note that s_1, s_2 do not need to be distinct sensors, so a vertex which is measured by only one sensor can still have a pivotal vertex.

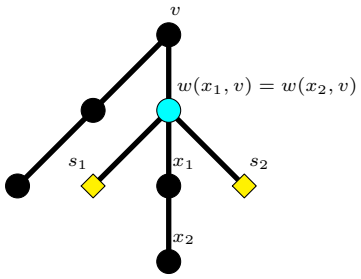


Figure 5: $k = 3$. Both x_1 and x_2 are measured by s_1 and s_2, and for both x_1 and x_2 the paths to s_1 and s_2 have the same point at which they split from the path from x_1 to v and from x_2 to v, respectively, so the vertex in blue, $w(x_1, v) = w(x_2, v)$, is the pivotal vertex corresponding to x_1 and x_2.

3.3 Quantities the algorithm keeps track of

While running the algorithm we keep track of eight items, all are functions of their position in the tree: $Seen(v), S(v,m), DistS(v,m), NotM(v,m), Pr_1(v,m), Pr_2(v,m), Pr_f(v,m), l(v,m)$, where $v \in V(T)$ and $m \in \{0, 1, .., ch(v)\}$. In the list below and in Section 3.3.1, we define each item and motivate why we need them.

1. $Seen(v)$ is the binary variable which equals 1 if the vertex v is part of a subtree for which the algorithm has allocated sensors, 0 otherwise. When processing a vertex v the algorithm can perform two operations;

adding the parent $p(v)$ of the vertex v to the subtree $T_{v,ch(v)}$, and merging the subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ together into $T_{p(v),m}$. Remember that in our definition of the notation, $m \in \{1, 2, ..., ch(p(v))\}$ (Definition 3.3). The algorithm decides which operation to perform based on whether $Seen(p(v))$ is equal to 0 or 1. We will call a vertex v *considered* if the algorithm sees that vertex as being part of a subtree ($Seen(v) = 1$) but v itself has not been processed yet. For example, for any subtree $T_{v,m}$ where v is not a leaf, the root of the subtree is the only vertex in $T_{v,m}$ which has not been processed yet. However, because v is considered part of a subtree (i.e. $Seen(v) = 1$), we say v is considered by the algorithm.

2. We define $S(v, m)$ as the already allocated sensor-vertices in a subtree $T_{v,m}$.

3. $DistS(v, m)$ is the distance from v to the closest sensor to v in the subtree $T_{v,m}$. When the algorithm merges subtrees it is necessary to know which of the vertices in one subtree are measured by the sensors in the other subtree. For this, we need to know how far the closest sensor in a subtree is from its root.

4. $l(v, m)$ is the largest distance such that, for any vertex s^* outside of $T_{v,m}$ with $d(s^*, v) = l(v, m)$, the set $S(v, m) \cup \{s^*\}$ is a k-resolving set for $T_{v,m}$. We set $l(v, m) = null$ if $S(v, m)$ already resolves $T_{v,m}$. The idea that $S(v, m)$ combined with a sensor outside $T_{v,m}$ at distance $l(v, m)$ from v resolves $T_{v,m}$ is very important and will be relied on many times to explain and prove the algorithm. Due to its great importance, this property/requirement that for any subtree $T_{v,m}$, the allocated sensors $S(v, m)$ in the subtree either fully resolve that subtree or placing one more sensor no further than distance $l(v, m)$ from v resolves $T_{v,m}$ is called the *fundamental property/requirement*. We use $l(v, m)$ to keep track of where we need to place a sensor at a later point in the algorithm, while still keeping our options open for where exactly we place that sensor.

5. $NotM(v, m)$ is the set of distances from v to all vertices $y \in T_{v,m}$ which are not measured by the sensors in its subtree $S(v, m)$.

$$NotM(v, m) = \{d(v, y) \mid y \in T_{v,m} : \forall s \in S(v, m), d(y, s) > k\}$$

This set is used to check the fundamental requirement of the algorithm holds while processing a vertex.

3.3.1 Problematic vertices

The set of vertices in a subtree $T_{v,m}$ which can cause problems when adding the parent vertex to the subtree or when merging two subtrees will from now on be referred to as the set of *problematic vertices* in $T_{v,m}$, denoted as $Pr(v, m)$. By 'problems', we mean a contradiction in the fundamental requirement for the algorithm that $S(v, m) \cup \{s^*\}$ (where s^* is a vertex at distance $l(v, m)$ from the subtree's root v) is a k-resolving set for $T_{v,m}$. We check this requirement holds throughout the algorithm. For instance, in the situation that $l(v, ch(v)) = null$ and the algorithm decides to add the parent vertex $p(v)$ to the subtree, the algorithm needs to check whether this parent vertex is still distinguished from all other vertices. If it is distinguished, the sensors in $T_{v,ch(v)}$ form a k-resolving set for $T_{p(v),1}$ and therefore we set $l(p(v), 1) = null$. If it is not distinguished, we need a sensor somewhere which distinguishes $p(v)$. The largest distance from $p(v)$ such that a sensor at that distance can distinguish $p(v)$ from all other vertices in $T_{p(v),1}$ is k, so we set $l(p(v), 1) = k$. We define two types of problematic vertices by how they are measured:

(Pr 1) We call a vertex x *type-1* problematic if for all sensors that measure x, the path from x to that sensor passes through the pivotal vertex of x, $w(x, v)$ (Definition 3.6), and $w(x, v) = v$, so all sensors measuring x measure x through its pivotal vertex, and this pivotal vertex is equal to the root of the subtree v. All paths from $p(v)$ to a sensor must go through v, so a vertex which is only measured through v is indistinguishable from $p(v)$ when it has the same distance to v as $p(v)$, namely one. The reason we keep track of all vertices measured only through v and not just those adjacent to v is because these vertices can still become (type-2) problematic later in the algorithm, when the subtree containging x has expanded. In $Pr_1(v, m)$, the set of of type-1 problematic vertices in $T_{v,m}$, where $m \in \{1, 2, ..., ch(v)\}$, we store a tuple of two values. Let x be type-1 problematic, then

$$(d(v, x), d(v, s)) \in Pr_1(v, m)$$

Remark 3.7. *In the tuple* $(d(v,x), d(v,s))$*,* s *is the closest sensor to* v*.*

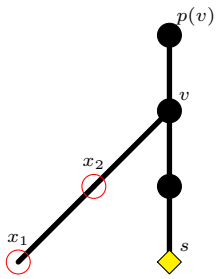


Figure 6: $k = 4$. The yellow, square vertex (s) has a sensor. The red, circle vertices (x_1, x_2) are measured through the root v of the subtree, and are thus type-1 problematic. When adding the parent vertex $p(v)$ to the subtree, x_1, x_2 can become indistinguishable from $p(v)$, depending on their distance from v. If a second sensor is placed on one of the two circle-vertices, there would be no type-1 problematic vertices in this graph. It is important to remember that type-1 problematic vertices are not necessarily indistinguishable from $p(v)$, we keep track of them because they might be indistinguishable now *or* because they might become indistinguishable from the parent of the subtree's root at some point later in the algorithm, when the algorithm has added more vertices to that subtree. In this example, the leftmost type-1 problematic vertex will never be indistinguishable from $p(v)$ because there are no more vertices in the tree left to add to the subtree.

(Pr 2) We call a vertex $x \in T_{v,m}$ *type-2* problematic if for all sensors that measure x, the path from x to that sensor passes through the pivotal vertex of x, $w(x,v)$, where $w(x,v) \neq v$ and two further requirements hold: $d(x, w(x,v)) = d(w(x,v), v) + 1$ and there is no sensor s in $S(v,m)$ which measures $p(v)$ such that $w(x,v)$ is not in $P_{sp(v)}$. These two requirements state that the pivotal vertex is equally far away from x as from $p(v)$ and there is no sensor in $S(v,m)$ which distinguishes x and $p(v)$. In $Pr_2(v,m)$, we store identical values to $Pr_1(v,m)$. Let x be type-2 problematic, then

$$(d(v,x), d(v,s)) \in Pr_2(v,m)$$

Remark 3.8. *In the tuple* $(d(v,x), d(v,s))$*,* s *is the closest sensor to* v*.*

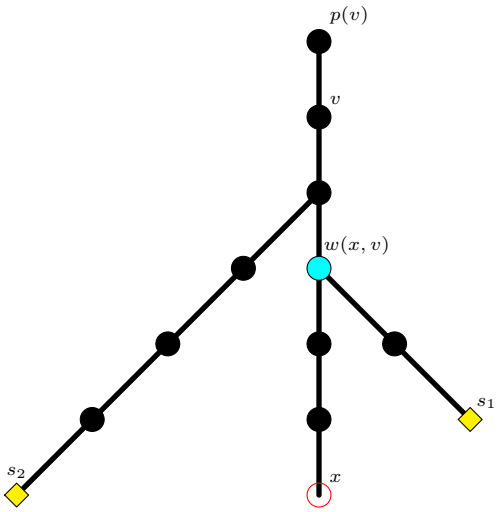


Figure 7: $k = 5$. Sensors (yellow, square), pivotal vertex $w(x, v)$ (blue, dot), type-2 problematic vertex x (red, circle). The sensors resolve $T_{v,ch(v)}$, but they do not resolve $T_{p(v),1}$, because x is only measured through its pivotal vertex $w(x, v)$, $w(x, v)$ is equally far away from x as from $p(v)$ and for all sensors that measure $p(v)$, $w(x, v)$ is on the path from that sensor to $p(v)$. This means that when the algorithm processes vertex v and adds its parent, $p(v)$, to the subtree, that subtree is now no longer resolved by the sensors in the subtree.

(Pr f) A vertex x is in $Pr_f(v, m)$ if for all sensors that measure x, the path from x to that sensor passes through the pivotal vertex of x, $w(x, v)$, and two further requirements hold: $d(x, w(x, v)) \neq d(w(x, v)) + 1$ and there is no sensor s in $S(v, m)$ which distinguishes a vertex outside $T_{v,m}$ at distance $d(x, w(x, v))$ from $w(x, v)$ and the vertex x.

For algorithmic reasons we will explain in Section 3.7.5 and for the sake of computational complexity, we require an efficient way to update Pr_1 and Pr_2. To this end, note there is a flow of vertices from Pr_1 to Pr_2. A vertex in $Pr_1(v, m)$ is only measured through the root of the subtree v, and when the algorithm has expanded the subtree enough, v becomes the vertex exactly in between x and the parent of the new root, so the sensors measuring x can not distinguish between x and the parent of the new root, therefore it is type-2 problematic. In fact, all vertices in Pr_2 must, at an earlier point, have been vertices in Pr_1. To efficiently update Pr_2, we keep track of another item while running the algorithm, Pr_f (f stands for future). The set Pr_f consists of all vertices that used to be in Pr_1 but are now no longer type-1 problematic, but can still become type-2 problematic in the future.

$$Pr_1 \longrightarrow Pr_f \longrightarrow Pr_2$$

In $Pr_f(v, m)$, we store two additional values to the values stored in $Pr_1(v, m)$. The additional values are $d(w(x, v), v)$ and $d(w(x, v), x)$, where $w(x, v)$ is the pivotal vertex through which x is measured (3.6), and v is the current root of the subtree. When $d(w(v, x), x) = d(w(x, v), v) + 1$, x becomes type-2 problematic. We need these additional values to check when a vertex in Pr_f needs to go to Pr_2 and in order to perform *Check-resolving* (See 3.7.5 for an explanation of *Check-resolving*). To reiterate, when x is in $Pr_f(v, m)$, we store

$$(d(v, x), d(v, s), d(w(x, v), v).d(w(x, v), x)) \in Pr_f(v, m)$$

Remark 3.9. *In this tuple, s is the closest sensor to v.*

The set of problematic vertices is the set of all type-1 and type-2 problematic vertices combined:

$$Pr(v, m) = Pr_1(v, m) \cup Pr_2(v, m)$$

We now give a formal definition of the types of problematic vertices. Let x be a vertex in a subtree $T_{v,m}$ with root v and let s be any sensor measuring x.

(Case 1) If $P_{xv} \cap P_{xs} = \{x\}$, then all sensors which measure x are part of the subtree with root x, i.e. $s \in T_{x,ch(x)}$. These vertices are always distinguished from any vertex outside $T_{v,m}$, so $x \notin Pr(v,m) \cup Pr_f(v,m)$.

(Case 2) If $\exists s_1, s_2 \in S(v,m), |P_{xs_1}|, |P_{xs_2}| \leq k$ such that $P_{xv} \cap P_{xs_1} \neq P_{xv} \cap P_{xs_2}$, then we know that for any vertex outside $T_{v,m}$, either s_1 or s_2 distinguishes x and that vertex, so $x \notin Pr(v,m) \cup Pr_f(v,m)$.

(Case 3) If $\forall s_1, s_2 \in S(v,m)$ such that $|P_{xs_1}|, |P_{xs_2}| \leq k : P_{xv} \cap P_{xs_1} = P_{xv} \cap P_{xs_2}$, then by how we defined the pivotal vertex of x, this means for all sensors which measure x, the pivotal vertex $w(x,v)$ is on the path from x to that sensor (Definition 3.6). For such x, we separate three subcases:

(Case 3.1) If $w(x,v) = v$, then for all sensors which measure x, the root of the subtree v is on the path from that sensor to x, so $x \in Pr_1(v,m)$.

(Case 3.2) If $w(x,v) \neq v$ and $d(w(x,v),x) = d(w(x,v),v) + 1$ and $\forall s \in S(v,m) : d_k(s,p(v)) = d_k(s,x)$, then the pivotal vertex has equal distance from x as $p(v)$ and there is no sensor in $S(v,m)$ which distinguishes x and $p(v)$, so $x \in Pr_2(v,m)$.

(Case 3.3) If $w(x,v) \neq v$ and $d(w(x,v),x) \neq d(w(x,v),v) + 1$ and $\forall s \in S(v,m) : min(k+1, |P_{p(v)s}| + d(x,w(x,v)) - d(w(x,v),p(v))) = min(k+1,|P_{xs}|)$, then the pivotal vertex $w(x,v)$ is not in the middle of x and $p(v)$ yet, and there is no sensor in $S(v,m)$ which distinguishes x and the vertex outside $T_{v,m}$ at distance $d(w(x,v),x)$ away from $w(x,v)$ which would clash with x, so the vertex x can become type-2 problematic in the future. Therefore, $x \in Pr_f(v,m)$.

(Case 3.4) Otherwise, $x \notin Pr(v,m) \cup Pr_f(v,m)$.

It is possible for a vertex to move from Pr_1 directly to Pr_2, as illustrated by the figure below.

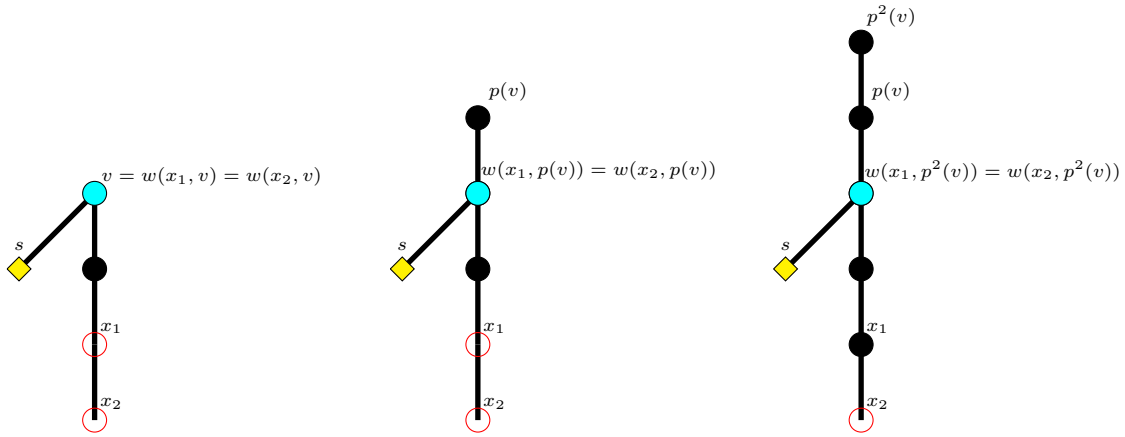


Figure 8: $k = 4$. In (a), x_1 and x_2 are measured only though their pivotal vertex which is the subtrees root v, so $x_1, x_2 \in Pr_1(v, ch(v))$. In the consecutive step, seen in (b), $w(x_1, p(v))$ is now the unique vertex with $d(w(x_1,p(v)),p(v)) + 1 = d(w(x_1,p(v)),x_1)$, so $x_1 \in Pr_2(p(v), ch(p(v)))$. Vertex x_2 is now in $Pr_f(p(v), ch(p(v)))$, because it is no longer type-1 problematic, but it can still become type-2 problematic in the future. In the final step (c), x_1 is no longer type-2 problematic and can never become problematic in the future. Vertex x_2 is still measured only through its pivotal vertex $w(x_2, p^2(v))$ which is now placed exactly in between x_2 and the parent of the current root of the subtree, $p^3(v)$, so $x_2 \in Pr_2(p^2(v), ch(p^2(v)))$.

In the next subsection, we give a method to update all items.

3.4 Updating quantities the algorithm keeps track of

The manner in which we update the quantities we keep track of is important for those interested in implementing the algorithm and for analyzing the time complexity. How we update the quantities depends on whether the algorithm decides to add the parent vertex of the vertex it is processing or whether it decides to merge subtrees.

3.4.1 Updating quantities the algorithm keeps track of when adding parent

Suppose the algorithm is processing a vertex v and adds the parent of v, $p(v)$, to the subtree $T_{v,ch(v)}$, thereby expanding the tree to $T_{p(v),1}$. We list the processes of updating quantites for when a sensor is placed on $p(v)$ and when no sensor is placed on $p(v)$ separately:

Sensor on $p(v)$, i.e. $l(v, ch(v)) = 1$

(a) $Seen(p(v)) = 1$. Vertex $p(v)$ is now part of a subtree which the algorithm has considered, so $Seen(p(v))$ is set to 1.

(b) $DistS(p(v), 1) = 0$. If a sensor is placed on $p(v)$, naturally the closest sensor in $S(p(v), 1)$ has distance 0 from $p(v)$, so $DistS(p(v), 1)$ is set to 0.

(c) $NotM(p(v), 1) = \emptyset$. After placing a sensor on $p(v)$, $S(p(v), 1)$ forms a resolving set for $T_{p(v),1}$ by the fundamental requirement of the algorithm. For $S(p(v), 1)$ to be a resolving set, every vertex in $T_{p(v),1}$ must be measured, so $NotM(p(v), 1) = \emptyset$.

(d) $Pr_1(p(v), 1) = NotM(v, ch(v))$. Every vertex which was not measured by $S(v, ch(v))$ is now measured by the sensor on $p(v)$, so moves to $Pr_1(p(v), 1)$. Every vertex which was in $Pr_1(v, ch(v))$ was measured only through v. After adding $p(v)$ to the subtree, v is no longer the root of the subtree, therefore everything that was in $Pr_1(v, ch(v))$ is no longer type-1 problematic.

(e) $Pr_f(p(v), 1) = \emptyset$. Every vertex x which was in $Pr_f(v, ch(v))$ is not moved to $Pr_f(p(v), 1)$, because the sensor on $p(v)$ distinguishes x from any vertex at distance $d(x, w(w, v))$ from v, so we set $Pr_f(p(v), 1) = \emptyset$.

(f) $Pr_2(p(v), 1) = \emptyset$. Vertices in $Pr_2(v, ch(v))$ are not distinguished from $p(v)$ by the sensors $S(v, ch(v))$. The sensor on $p(v)$ distinguishes all vertices in $Pr_2(v, ch(v))$ from $p(v)$, so everything that was in $Pr_2(v, ch(v))$ is no longer type-2 problematic.

No sensor on $p(v)$, i.e. $l(v, ch(v)) \neq 1$

(a) $Seen(p(v)) = 1$. The parent of v, $p(v)$ is now part of a subtree which the algorithm has considered, so $Seen(p(v))$ is set to 1.

(b) $DistS(p(v), 1) = DistS(v, ch(v)) + 1$. The sensor in $T_{v,ch(v)}$ which was closest to v is also the sensor in $T_{p(v),1}$ closest to $p(v)$, but the distance from $p(v)$ to the sensor is naturally one greater than the distance from v to the sensor, so we increase $DistS(v, ch(v))$ by 1.

(c) Since no sensor was placed on $p(v)$, the vertices which were unmeasured by $S(v, ch(v))$ are still unmeasured by $S(p(v), 1)$, and now their distance from the root (the root of the subtree changes from v to $p(v)$) is 1 greater than it was before, so all elements in $NotM(v, ch(v))$ are increased by 1 and moved to $NotM(p(v), 1)$. If $p(v)$ is not measured by $S(p(v), 1)$ (so if $DistS(p(v), 1) > k$), 0 is added to $NotM(p(v), 1)$. Otherwise, $p(v)$ is measured so no new element is added to $NotM(p(v), 1)$.

(d) $Pr_1(p(v), 1) = \emptyset$. There is no sensor on the new root of the subtree $p(v)$, so there is no sensor in $S(p(v), 1)$ which can measure any vertex in $T_{p(v),1}$ through $p(v)$. Therefore $Pr_1(p(v), 1) = \emptyset$.

(e) All vertices x in $Pr_f(v, ch(v))$ for which

$$Pr_f^{(1)} \coloneqq \{x \in Pr_f(v, ch(v)) \mid d(x, w(x, p(v)) \neq d(w(x, p(v)), p(v)) + 1\}$$

are moved to $Pr_f(p(v), 1)$. All vertices $x \in Pr_1(v, ch(v))$ for which

$$Pr_f^{(2)} \coloneqq \{x \in Pr_1(v, ch(v)) \mid d(x, w(x, p(v))) \neq d(x, p(v)) + 1\}$$

are moved to $Pr_f(p(v), 1)$. Combining these sets, we find

$$Pr_f(p(v), 1) = Pr_f^{(1)} \cup Pr_f^{(2)}$$

(f) All vertices in $Pr_2(v, ch(v))$ are not moved to $Pr_2(p(v), 1)$, as they can no longer clash with the parent of the new root of the subtree, $p(v)$, by definition of type-2 problematic vertices (Definition 3.3.1). All vertices x in $Pr_1(v, ch(v))$ or $Pr_f(v, ch(v))$ for which

$$Pr_2(p(v), 1) = \{x \in Pr_1(v, ch(v)) \cup Pr_f(v, ch(v)) \mid d(x, w(x, p(v))) = d(w(x, p(v)), p(v)) + 1\}$$

are moved to $Pr_2(p(v), 1)$, as these vertices are not distinguished from the parent of the new root of the subtree (the new root of the subtree is $p(v)$) by the sensors in $S(p(v), 1)$.

3.4.2 Updating quantities the algorithm keeps track of when merging subtrees.

Suppose the algorithm is processing a vertex v which is the m-th child of $p(v)$ and the algorithm merges the subtrees $T_{p(v),m-1}$ and $T_{v,ch(v)}$ into the single subtree $T_{p(v),m}$. We list the processes of updating quantities for when the algorithm merges subtrees:

(a) No value of the function *Seen* needs to be updated, because the algorithm does not expand its 'view' of any subtree, it only merges two.

(b) $DistS(p(v), m) = min(DistS(v, ch(v)) + 1, DistS(p(v), m - 1))$. Both subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ have a sensor which is closest to the root of their subtree, v and $p(v)$ respectively. The algorithm takes the distance from $p(v)$ to the nearest sensor in $S(p(v), m)$ as the new value for $DistS(p(v), m)$.

(c) All vertices in $NotM(v, ch(v))$ which are not measured by the sensor corresponding to $DistS(p(v), m-1)$ are moved to $NotM(p(v), m)$.

$$NotM(p(v), m)^{(1)} \coloneqq \{x \in NotM(v, ch(v)) \mid d(x, v) + 1 + DistS(p(v), m - 1) > k\}$$

These vertices were unmeasured by the sensors in their own subtree $S(p(v), m - 1)$ and are still unmeasured by the sensors in the other subtree $S(v, ch(v))$, so they remain unmeasured by the sensors in the merged subtree. All vertices in $NotM(p(v), m - 1)$ which are not measured by the sensor corresponding to $DistS(v, ch(v))$ are moved to $NotM(p(v), m)$.

$$NotM(p(v), m)^{(2)} \coloneqq \{x \in NotM(p(v), m - 1) \mid d(x, p(v)) + 1 + DistS(v, ch(v)) > k\}$$

These vertices were unmeasured by the sensors in their own subtree $S(v, ch(v))$ and are still unmeasured by the sensors in the other subtree $S(p(v), m - 1)$, so they remain unmeasured by the sensors in the merged subtree. We find

$$NotM(p(v), m) = NotM(p(v), m)^{(1)} \cup NotM(p(v), m)^{(2)}$$

(d) All vertices in $NotM(v, ch(v))$ which are measured by the sensor corresponding to $DistS(p(v), m - 1)$ are moved to $Pr_1(p(v), m)$.

$$Pr_1(p(v), m)^{(1)} \coloneqq \{x \in NotM(v, ch(v)) \mid d(x, v) + 1 + DistS(p(v), m - 1) \leq k\}$$

These vertices in $NotM(v, ch(v))$ are unmeasured by the sensors in their own subtree $S(v, ch(v))$, but they are measured by sensor(s) in the other subtree $S(p(v), m - 1)$, which can only measure vertices in $T_{v,ch(v)}$ through the root of the merged subtree, $p(v)$, therefore these vertices are now type-1

problematic for the merged subtree. All vertices in $NotM(p(v), m-1)$ which are measured by the sensor corresponding to $DistS(v, ch(v))$ are moved to $Pr_1(p(v), m)$.

$$Pr_1(p(v), m)^{(2)} \coloneqq \{x \in NotM(p(v), m-1) \mid d(x, p(v)) + 1 + DistS(v, ch(v)) \leq k\}$$

These vertices in $NotM(p(v), m-1)$ are unmeasured by the sensors in their own subtree $S(p(v), m-1)$, but they are measured by sensor(s) in the other subtree $S(v, ch(v))$, which can only measure vertices in $T_{p(v),m-1}$ through the root of the merged subtree, $p(v)$, therefore they are now type-1 problematic for the merged subtree. Everything in $Pr_1(p(v), m-1)$ moves to $Pr_1(p(v), m)$, because no sensor was placed in $T_{p(v),m-1}$, so vertices in $Pr_1(p(v), m-1)$ must still be measured only through the root of the merged subtree, $p(v)$. We find

$$Pr_1(p(v), m) = Pr_1(p(v), m)^{(1)} \cup Pr_1(p(v), m)^{(2)} \cup Pr_1(p(v), m-1)$$

(e) Updating $Pr_f(p(v), m)$ goes as follows. We observe that if a vertex x is type-1 problematic in subtree $T_{v.ch(v)}$, the pivotal vertex of x, $w(x, v)$, is equal to v. All $x \in Pr_1(v, ch(v))$ for which

$$\begin{aligned} Pr_f(p(v), m)^{(1)} \coloneqq \{x \in Pr_1(v, ch(v)) \mid d(x, w(x, v)) \neq d(w(x, v), p(v)) + 1 \text{ and} \\ DistS(p(v), m-1) + d(x, w(x, v)) - 1 > k\} \end{aligned} \tag{3.1}$$

are moved to $Pr_f((p(v), m))$. The first of the two checks functions as a way to see if the vertex x might move to $Pr_f(p(v), m)$ or $Pr_2(p(v), m)$. If it is true, x might move to $Pr_f(p(v), m)$, and definitely not to $Pr_2(p(v), m)$. The second check makes this move definitive, it checks whether a vertex at distance $d(x, w(x, v))$ from v is distinguished by the sensor corresponding to $DistS(p(v), m-1)$ or not. If not, the vertex x moves to $Pr_f(p(v), m)$. For understanding the syntax in the second check it is important we show the following: Let y denote a vertex at distance $d(x, w(x, v))$ from $w(x, v)$, so y is a vertex which is indistinguishable from x iff x belongs in $Pr_f(p(v), m)$ or $Pr_2(p(v), m)$. We will show $d(p(v), y)$ can be computed from the values stored earlier in $Pr_1(v, ch(v))$:

$$\begin{aligned} d(x, y) &= 2 * d(x, w(x, v)) \\ d(p(v), y) &= 2 * d(x, w(x, v)) - d(x, p(v)) \\ d(p(v), y) &= d(x, w(x, v)) - d(w(x, v), v) - 1 \end{aligned}$$

Because x is type-1 problematic in $T_{v,ch(v)}$, $d(x, w(x, v))$ and $d(w(x, v), v)$ are known (Definition 3.3.1), so $d(p(v), y)$ can be computed from values stored in $Pr_1(v, ch(v))$. All $x \in Pr_f(p(v), m-1)$ for which

$$\begin{aligned} Pr_f(p(v), m)^{(2)} \coloneqq \{x \in Pr_f(p(v), m-1) \mid \\ 1 + DistS(v, ch(v)) + d(x, w(x, p(v))) - d(w(x, p(v)), p(v)) > k\} \end{aligned}$$

are moved to $Pr_f(p(v), m)$. This checks is equivalent to the second check in (3.1), so whether a vertex outside $T_{p(v),m}$ at distance $d(x, w(x, p(v))$ from $w(x, p(v))$ is measured by a sensor in $S(v, ch(v))$.
All $x \in Pr_f(v, ch(v))$ for which

$$\begin{aligned} Pr_f(p(v), m)^{(3)} \coloneqq \{x \in Pr_f(v, ch(v)) \mid d(x, w(x, v)) \neq d(w(x, v), p(v)) + 1 \text{ and} \\ DistS(p(v), m-1) + d(x, w(x, v)) - d(w(x, v), v) - 1 > k\} \end{aligned}$$

are moved to $Pr_f(p(v), m)$. In the first check, we check if the vertex $x \in Pr_f(v, ch(v))$ should move to $Pr_f(p(v), m)$ or $Pr_2(p(v), m)$. The second check is equivalent to the second check of (3.1). We find

$$Pr_f(p(v), m) = Pr_f(p(v), m)^{(1)} \cup Pr_f(p(v), m)^{(2)} \cup Pr_f(p(v), m)^{(3)}$$

(f) Updating $Pr_2(p(v), m)$ goes as follows. All $x \in Pr_2(p(v), m-1)$ for which

$$Pr_2(p(v), m)^{(1)} \coloneqq \{x \in Pr_2(p(v), m-1) \mid DistS(v, ch(v)) + 2 > k\}$$

are moved to $Pr_2(p(v), m)$. This checks if the parent of the root of the merged subtree is measured by a sensor in $S(v, ch(v))$. If it is, this sensor distinguishes x and the parent of the root of the merged subtree. If it is not, the vertex x is moved to $Pr_2(p(v), m)$. This check does not actually depend on x itself, so in the implementation of the algorithm we do not need to iterate over $Pr_2(p(v), m-1)$. We could simply perform the check shown above if $Pr_2(p(v), m-1)$ is not empty and not perform the check above otherwise.
All $x \in Pr_1(v, ch(v))$ for which

$$Pr_2(p(v), m)^{(2)} \coloneqq \{x \in Pr_1(v, ch(v)) \mid d(x, v) = 2 \text{ and } DistS(p(v), m-1) > k-1\}$$

are moved to $Pr_2(p(v), m)$. If a vertex x in $Pr_1(v, ch(v))$ has distance 2 from v and no sensor in $S(p(v), m-1)$ measures the parent of the root of the merged subtree, it is type-2 problematic because it is not distinguished from the parent of the root of the merged subtree by any sensor in $S(p(v), m)$.
All $x \in Pr_f(v, ch(v))$ for which

$$Pr_2(p(v), m)^{(3)} \coloneqq \{x \in Pr_f(v, ch(v)) \mid d(x, w(x, v)) = d(w(x, v), p(v)) + 1 \text{ and } DistS(p(v), m-1) > k-1\}$$

are moved to $Pr_2(p(v), m)$. We find

$$Pr_2(p(v), m) = Pr_2(p(v), m)^{(1)} \cup Pr_2(p(v), m)^{(2)} \cup Pr_2(p(v), m)^{(3)}$$

The following section provides a visual overview of the algorithm.

3.5 Visualisation of the algorithm

In the figure below, we present a flowchart of the steps the algorithm takes when processing a vertex.

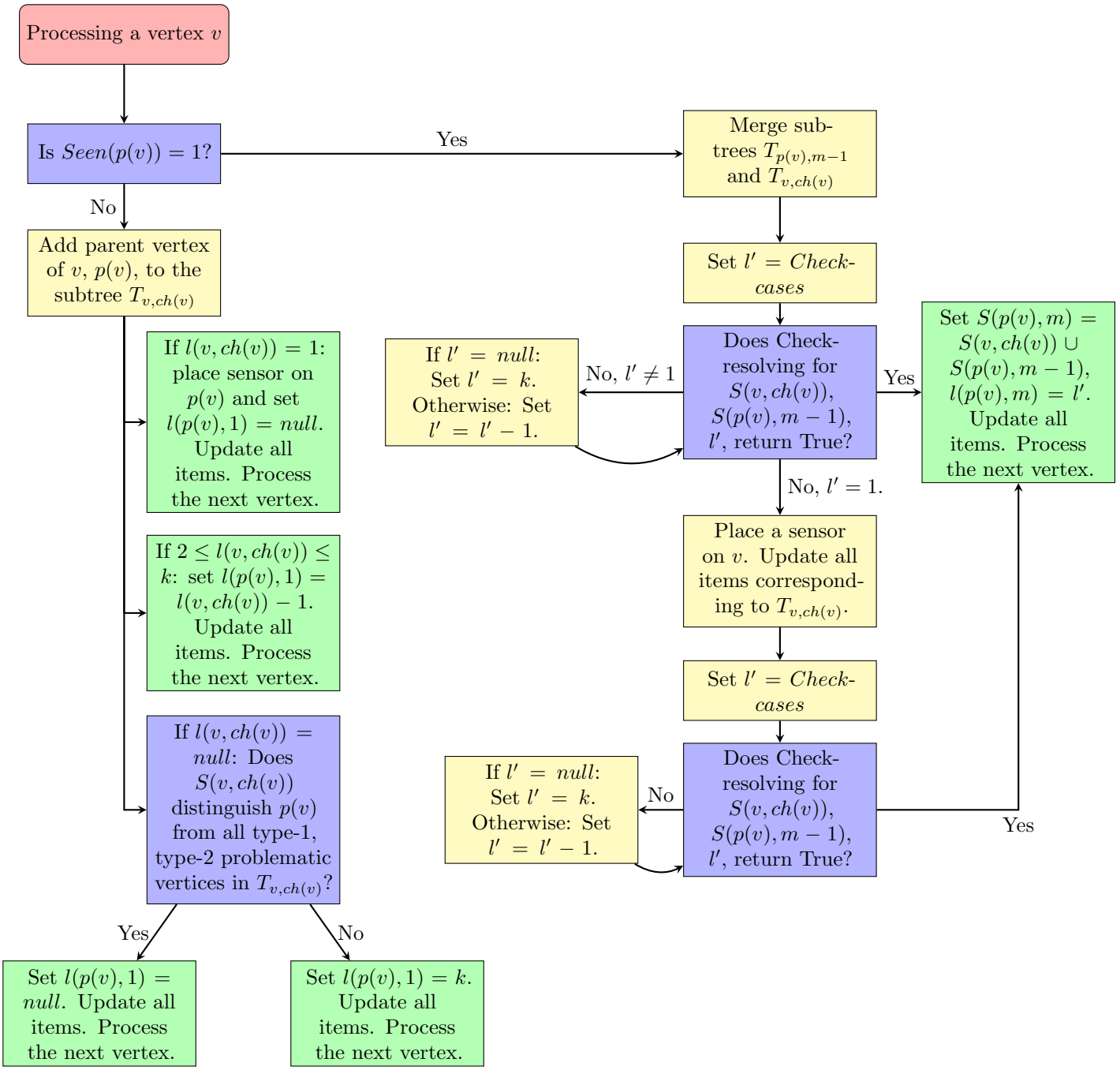


Figure 9: A flowchart of how the algorithm processes a vertex. On each green bubble, the algorithm finishes processing v, and will continue by processing the next vertex in the endvertex list. On each blue bubble, the algorithm performs a Yes/No check.

With this visual aid in mind, we proceed to explain the steps of the algorithm in greater detail.

3.6 Adding the parent vertex

The following subsection explains how the algorithm adds a parent vertex to a subtree. First, we give a heuristic explanation of the ideas in Section 3.6.1. Second, the pseudocode for adding a parent vertex to a subtree is shown in Section 3.6.2.

3.6.1 Heuristic explanation of adding the parent

As mentioned in the previous paragraph, the algorithm decides to add the parent ($p(v)$) of the vertex it is processing (v) when $Seen(p(v)) = 0$.

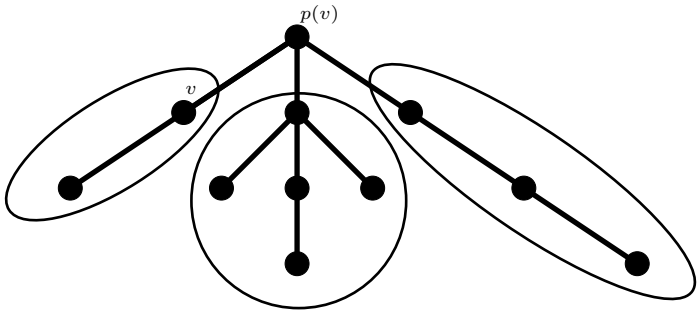


Figure 10: Vertices which the algorithm sees as being part of the same subtree are in the same circle. The vertex v is being processed. The algorithm sees that its parent, $p(v)$, is not seen as part of any subtree, so it decides to add $p(v)$ to the subtree with root v, $T_{v,ch(v)}$. This expands the subtree to $T_{p(v),1}$.

When adding the parent vertex, we separate three cases, depending on the value of $l(v, ch(v))$. When v is a leaf, $l(v, 0)$ is assigned the value k (line 4-7 of the pseudocode). We list the separate cases for when v is not a leaf:

- If $l(v, ch(v)) = 1$ (line 10-13 in the pseudocode), then we must place a sensor on $p(v)$ as $p(v)$ is the only vertex outside of $T_{v,ch(v)}$ with distance one from v. Therefore $S(p(v), 1) = S(v, ch(v)) \cup \{p(v)\}, Seen(p(v)) = 1$, and update $DistS(p(v), 1)$, $NotM(p(v), 1)$, $Pr(p(v), 1)$.
- If $2 \leq l(v, ch(v)) \leq k$ (line 14-17 in the pseudocode), we need a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ away from v, so naturally we need a vertex at distance $l(p(v), 1) = l(v, ch(v)) - 1$ away from $p(v)$. Update $DistS(p(v), 1)$, $NotM(p(v), 1)$, $Pr(p(v), 1)$.
- If $l(v, ch(v)) = null$ (line 18-30 in the pseudocode), the set $S(v, ch(v))$ is a k-resolving set for $T_{v,ch(v)}$. When we add the parent $p(v)$, we therefore need to check if $S(v, ch(v))$ is also a k-resolving set for $T_{p(v),1}$, which is equivalent to checking whether $p(v)$ is measured and distinguished from all vertices in $T_{v,ch(v)}$ by $S(v, ch(v))$. If so, naturally we set $l(p(v), 1) = null$. If $p(v)$ is not measured and distinguished from all vertices in $T_{v,ch(v)}$ we need a sensor which distinguishes $p(v)$, preferably as far from $p(v)$ as possible, so we set $l(p(v), 1) = k$. We state a claim which narrows down the vertices the algorithm needs to check:

  Claim 3.10. *Suppose the algorithm is processing a vertex v, which is the root of the subtree $T_{v,ch(v)}$, and allocated $l(v, ch(v)) = null$. All vertices $x \in V(T_{v,ch(v)})$ which are not in $Pr(v, ch(v))$ are distinguished from $p(v)$ by the sensors $S(v, ch(v))$.*

  We prove this claim in Section 4. The process of determining whether $S(v, ch(v))$ distinguishes $p(v)$ and $x \in Pr(v, ch(v))$ is as follows: First, the algorithm checks if $p(v)$ is measured by checking if the

closest sensor to v is within distance $k-1$ from v. If this is true, the algorithm iterates over all vertices $x \in Pr_1(v,m)$, and checks if they are all distinguished from $p(v)$ by checking if x is not adjacent to v. The algorithm proceeds to check if there are any type-2 problematic vertices (Definition 3.3.1). If there are, they are by definition not distinguished from $p(v)$. If none of the checks fail, $S(v, ch(v))$ measures and distinguishes $p(v)$ and all vertices in $T_{v,ch(v)}$ so we set $l(p(v),1) = null$. Otherwise, set $l(p(v),1) = k$.

With this heuristic explanation complete, we continue by giving the pseudocode for adding the parent vertex.

3.6.2 Pseudocode for adding the parent vertex

Algorithm 1 Pseudo-code for adding the parent vertex

V is the vertex which the algorithm is processing.
P is the parent vertex of V.
After updating all items corresponding to $T_{P,1}$, the next vertex is processed.
if $ch(V) = 0$ **then**
  $S(V,0), S(P,1) \leftarrow \emptyset$
  Update all items corresponding to $T_{P,1}$.
  $Seen(P) = 1$
  $l(V,0) \leftarrow k$, $l(P,1) \leftarrow k-1$
  $Seen(V), Seen(P) \leftarrow 1$
else if $ch(V) \neq 0$ **then**
  Note that in this case, $S(V,ch(V)), DistS(V,ch(V)), NotM(V,ch(V)), Pr(V,ch(V))$ are known, and that $Seen(V) = 1$.
  if $l(V,ch(V)) = 1$ **then**
    $S(P,1) \leftarrow S(V,ch(V)) \cup \{P\}$
    $l(P,1) \leftarrow null$
    Update all items corresponding to $T_{P,1}$.
    $Seen(P) \leftarrow 1$
  else if $2 \leq l \leq k$ **then**
    $S(P,1) \leftarrow S(V,ch(V))$
    $l(P,1) \leftarrow l(V,ch(V)) - 1$
    Update all items corresponding to $T_{P,1}$.
    $Seen(P) = 1$
  else if $l(V,ch(V)) = null$ **then**
    We will check if $S(v,ch(v))$ is indeed a k-resolving set for $T_{P,1}$. If this is the case, we proceed with $l(P,1) = null$. If we encounter a contradiction, we proceed with $l(P,1) = k$.
    for $(d(V,x), d(V,s)) \in Pr_1(V,ch(V))$ **do**
      if $d(V,x) \neq 1$ **then**
        x and P are distinguished.
        if $Pr_2(V,ch(V)) = \emptyset$ **then**
          $S(V,ch(V))$ is a k-resolving set for $T_{P,1}$.
          $l(P,1) \leftarrow null$.
          Update all items corresponding to $T_{P,1}$.
          $Seen(P) = 1$
        else
          $x \in Pr_2(V,ch(V))$ and P are not distinguished.
          $l(P,1) \leftarrow k$
          $S(P,1) \leftarrow S(V,ch(V))$
          Update all items corresponding to $T_{P,1}$.
          $Seen(P) = 1$
        end if
      else

29: x and P are not distinguished.
$l(P,1) \leftarrow k$
$S(P,1) \leftarrow S(V, ch(V))$
Update all items corresponding to $T_{P,1}$.
$Seen(P) = 1$
30: **end if**
31: **end for**
32: **end if**
33: **end if**

We return to the analysis and proof of this part of the algorithm in Section 4.1.1.

3.7 Merging subtrees

Below we give a heuristic explanation of how the algorithm merges subtrees in Section 3.7.1 and the pseudo-code for merging subtrees in Section 3.7.2. In the pseudo-code for the final algorithm we separate the cases when the vertex being processed v is a leaf and when v is not a leaf. We do this because adding the parent vertex is a slightly different operation when v is a leaf as opposed to not a leaf. In the pseudo-code for merging subtrees, there is no need to distinguish between cases when v is a leaf and v is not a leaf, so this is left out of the pseudo-code in Section 3.7.2.

3.7.1 Heuristic explanation of merging subtrees

When the algorithm is processing a vertex v which is the m-th child of $p(v)$ and decides to merge subtrees, it will merge the two subtrees $T_{v,ch(v))}$ and $T_{p(v),m-1}$ into the single subtree $T_{p(v),m}$, so the subtree with root v will merge with the subtree of its parent $p(v)$.

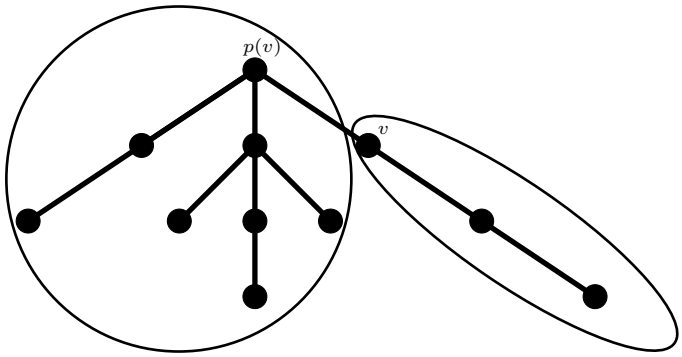


Figure 11: Vertices which the algorithm sees as being part of the same subtree are in the same circle. The vertex v is being processed. The algorithm sees that its parent, $p(v)$, is already part of a subtree which the algorithm has seen, so it decides to merge the subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ into one subtree, $T_{p(v),m}$. In this case, $m = 3$, because v is the third child of $p(v)$.

The idea behind merging subtrees is that we see if we can postpone placing a sensor, and only when we encounter a contradiction in our requirement that $S(p(v), m)$ combined with a sensor outside $T_{p(v),m}$ at distance $l(p(v), m)$ from $p(v)$ is a k-resolving set for $T_{p(v),m}$ do we place a new sensor. We do not allocate $l(p(v), m)$ immediately. Instead, we allocate a 'temporary' value for $l(p(v), m)$, denoted by l'. We use this temporary value l' to check if using this value for $l(p(v), m)$ would contradict the aforementioned requirement.

If the algorithm finds a contradiction, this value l' would not work as a value for $l(p(v), m)$, so a new value for l' must be chosen or, if there are no more values for l' left to try, a sensor must be placed on v.

For the merged subtree $T_{p(v),m}$ to be resolved means all possible vertex-vertex pairs in the subtree are distinguished by $S(p(v), m) = S(p(v), m-1) \cup S(v, ch(v))$ combined with a sensor at distance $l(p(v), m)$ from $p(v)$. We separate two kinds of vertex-vertex pairs: The first is a pair where both vertices are from the same subtree (from the same subtree before merging, that is), the second is a pair where the vertices are in different subtrees. To make sure all vertex-vertex pairs where both vertices are from the same subtree are distinguished, we use the 'sub'-algorithm *Check-cases* (Section 3.7.3). *Check-cases* determines the largest value l' such that all vertex-vertex pairs in $T_{p(v),m-1}$ and all vertex-vertex pairs in $T_{v,ch(v)}$ are resolved by the sensors in the merged subtree $S(p(v), m)$ combined with a sensor outside the merged subtree at distance l' from the root of the merged subtree, $p(v)$. To summarize; *Check-cases* finds the largest value l' such that all vertex-vertex pairs where both vertices are from the same subtree are resolved by $S(p(v), m) = S(p(v), m-1) \cup S(v, ch(v))$ combined with a sensor outside $T_{p(v),m}$ at distance l' from $p(v)$. This means that any value larger than the value for l' found by *Check-cases* would certainly not result in a k-resolving set for $T_{p(v),m}$. To determine if all vertex-vertex pairs with the vertices in different subtrees are distinguished, we use another sub-algorithm: *Check-resolving* (Section 3.7.5). Given the set of sensors in the two subtrees and some value l', *Check-resolving* returns True if all vertex-vertex pairs with the vertices from different subtrees are distinguished by the combined set of sensors and a sensor outside the merged subtree $T_{p(v),m}$ at distance l' from the root $p(v)$, False if some such vertex-vertex pair is not distinguished.

The algorithm merges subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ as follows: As a first step, we set l' equal to the output of the sub-algorithm *Check-cases*. With this l', we perform *Check-resolving* on $S(p(v), m) = S(p(v), m-1) \cup S(v, ch(v))$ and l'. If *Check-resolving* returns True, the requirement that $S(p(v), m)$ combined with a sensor outside $T_{p(v),m}$ at distance l' from $p(v)$ resolves $T_{p(v),m}$ holds. We set the temporary l' as permanent $l(p(v), m)$ and we can continue with processing the next vertex in the endvertex list (Definition 2.5). If *Check-resolving* returns False, we need to try the next-largest value for l': We set $l' = l' - 1$ and run *Check-resolving* again. This is repeated until *Check-resolving* returns True or there are no more values for l' left to try (when $l' = 1$ failed). When the latter is the case we place a sensor on the vertex v which the algorithm is processing, and repeat the whole process explained in this paragraph.

3.7.2 Pseudocode for merging subtrees

Algorithm 2 Pseudo-code for merging subtrees

The algorithm is processing a vertex V with parent vertex P. It merges the subtrees $T_{P,m-1}$ and $T_{V,ch(V)}$ into the single subtree $T_{P,m}$.
$l' \leftarrow Check-cases(S(P, m-1), l(P, m-1), DistS(P, m-1), S(V, ch(V)), l(V, ch(V)), DistS(V, ch(V)))$
if $Check-resolving(S(P, m-1), l(P, m-1), DistS(P, m-1), S(V, ch(V)), l(V, ch(V)), DistS(V, ch(V)), l') = TRUE$ **then**
  $S(P, m) \leftarrow S(P, m-1) \cup S(V, ch(V))$
  $l(P, m) \leftarrow l'$
  Update all items corresponding to $T_{P,m}$.
else
  if $l' \neq null$ **then**
    $l' \leftarrow l' - 1$
    while $Check - resolving(S(P, m-1),\ l(P, m-1),\ DistS(P, m-1),\ S(V, ch(V)),\ l(V, ch(V)),\ DistS(V, ch(V)),\ l') = FALSE$ **do**
      if $l' \neq 1$ **then**
        $l' \leftarrow l' - 1$
        Continue the while-loop
      else
        $S(V, ch(V)) \leftarrow S(V, ch(V)) \cup \{V\}$.
        Update all items corresponding to $T_{V,ch(V)}$.
        Run this algorithm again for V and P.
      end if

```
17:       end while
18:     else
19:       l' ← k
20:       while Check − resolving(S(P, m − 1), l(P, m − 1), DistS(P, m − 1), S(V, ch(V)), l(V, ch(V)),
          DistS(V, ch(V)), l') = FALSE do
21:         if l' ≠ 1 then
22:           l' ← l' − 1
23:           Continue the while-loop
24:         else
25:           S(V, ch(V)) ← S(V, ch(V)) ∪ {V}.
              Update all items corresponding to T_{V,ch(V)}.
              Run this algorithm again for V and P.
26:         end if
27:       end while
28:     end if
29: end if
```

We call in two 'sub-'algorithms in lines 4,12 and 5,13. These algorithms, *Check-cases* and *Check-resolving* are explained in Section 3.7.3 and 3.7.5 respectively.

3.7.3 Heuristic explanation of Check-cases for merging subtrees

Suppose the algorithm is processing a vertex v and decides to merge subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ into a single subtree $T_{p(v),m}$. The *Check-cases* algorithm finds the 'temporary' value for $l(p(v), m)$, l', which the algorithm needs to proceed (Section 3.5). *Check-cases* assigns l' a value such that increasing it any further would certainly contradict the requirement for the algorithm that $S(p(v), m)$ combined with a sensor outside $T_{p(v),m}$ at distance l' from $p(v)$ is a k-resolving set for $T_{p(v),m}$. This value for l' is the largest value such that all vertex-vertex pairs in $T_{p(v),m}$ where both vertices are from the same subtree before merging are distinguished by $S(p(v), m)$ combined with a sensor outside $T_{p(v),m}$ at distance l' from $p(v)$. For any value larger than this, we know there would be a vertex-vertex pair in the merged subtree which is not distinguished, so we do not need to try it. If a subtree was allocated a value for l, then any value $l' < l$ would also resolve that subtree, as is stated in the following claim, proven in Section 4.1.2:

Claim 3.11. *Suppose the algorithm is processing a vertex v and allocated $S(v, ch(v))$ and $l(v, ch(v))$ such that $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from the root v is a k-resolving set for $T_{v,ch(v)}$. Then for any $l' \leq l(v, ch(v))$, $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance l' from the root v is also a k-resolving set for $T_{v,ch(v)}$.*

Because of this, any value for l' smaller than the first value for l' which *Check-cases* tries will always individually resolve each of the two subtrees. By individually resolve, we mean all vertex-vertex pairs where both vertices are from the same subtree are distinguished.

We separate seven cases, based on whether the subtrees are individually resolved by some sensor in the other subtree or not. The cases are as follows:

(Case 1) **$T_{p(v),m-1}$ is resolved by $S(v, ch(v))$ and $T_{v,ch(v)}$ is resolved by $S(p(v), m-1)$.**
This is the case when $l(p(v), m-1) \geq DistS(v, ch(v)) + 1$ and $l(v, ch(v)) \geq DistS(p(v), m-1) + 1$. Each of the subtrees individually is resolved by a sensor in the other subtree, so any value for l' might work.
We set $l' \leftarrow null$, if *Check-resolving* returns True, update all items and continue processing the next vertex in the endvertex list (Definition 2.5). If *Check-resolving* returns False, we set $l' \leftarrow k$, run *Check-resolving* again and continue this loop, decreasing l' on each iteration until *Check-resolving* returns True.

(Case 2) **$T_{p(v),m-1}$ is resolved by $S(v, ch(v))$ and $T_{v,ch(v)}$ is not resolved by $S(p(v), m-1)$.**
This is the case when $l(p(v), m-1) \geq DistS(v, ch(v)) + 1$ and $l(v, ch(v)) < 1 + DistS(p(v), m-1)$.
We know any sensor further than distance $l(v, ch(v)) - 1$ from $p(v)$ would not distinguish between all

vertex-vertex pairs in $T_{v,ch(v)}$.
We set $l' \leftarrow l(v, ch(v)) - 1$.
If *Check-resolving* returns False, we decrease l' by 1 and run *Check-resolving* again. This loop continues until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(Case 3) $T_{p(v),m-1}$ **is not resolved by** $S(v, ch(v))$ **and** $T_{v,ch(v)}$ **is resolved by** $S(p(v), m-1)$**.**
Analogous to Case 2; If $l(p(v), m-1) < 1 + DistS(v, ch(v))$ and $l(v, ch(v)) \geq DistS(p(v), m-1) + 1$. We know any sensor further than distance $l(p(v), m-1)$ from $p(v)$ would not distinguish between all vertex-vertex pairs in $T_{p(v),m-1}$.
We set $l' \leftarrow l(p(v), m-1)$.
If *Check-resolving* returns False, we decrease l' by 1 and run *Check-resolving* again. This loop continues until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(Case 4) $T_{p(v),m-1}$ **is not resolved by** $S(v, ch(v))$**,** $T_{v,ch(v)}$ **is not resolved by** $S(p(v), m-1)$**.**
This is the case when $l(p(v), m-1) < DistS(v, ch(v)) + 1$ and $l(v, ch(v)) < DistS(p(v), m-1) + 1$. To resolve both subtrees individually, we can not take l' to be any value larger than the smallest l of the two. Set $l' \leftarrow min(l(p(v), m-1), l(v, ch(v)) - 1)$.
If *Check-resolving* returns False, we decrease l' by 1 and run *Check-resolving* again. This loop continues until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(Case 5) $l(p(v), m-1) = null$ **and** $l(v, ch(v)) \neq null$**.**
We separate two cases;

(5.1) When $l(v, ch(v)) \geq DistS(p(v), m-1) + 1$, the subtree $T_{v,ch(v)}$ is resolved by some sensor in $S(p(v), m-1)$. Naturally, the sensors $S(p(v), m-1)$ already resolve $T_{p(v),m-1}$, so both subtrees are resolved individually and the new value for $l(p(v), m)$ can be any value. Then we use *Check-resolving* to find the optimal l'. First, we set $l' \leftarrow null$ and run *Check-resolving*. If *Check-resolving* returns True, update all items and continue processing the next vertex in the endvertex list (Definition 2.5). If *Check-resolving* returns False, we set $l' \leftarrow k$, run *Check-resolving* again and continue this loop, decreasing l' on each iteration until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(5.2) When $l(V, ch(V)) < 1 + DistS(p(v), m-1)$, the subtree $T_{v,ch(v)}$ is not resolved by any sensor in $S(p(v), m-1)$. Then the merged subtree still needs a sensor at distance no greater than $l(v, ch(v)) - 1$ from $p(v)$ to resolve all vertex-vertex pairs in $T_{v,ch(v)}$.
We set $l' \leftarrow l(v, ch(v)) - 1$. If *Check-resolving* returns False, we decrease l' by 1 and run *Check-resolving* again. This loop continues until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(Case 6) $l(p(v), m-1) \neq null$ **and** $l(v, ch(v)) = null$**.**
This is analogous to Case 5, so we separate two cases again;

(6.1) When $l(p(v), m-1) \geq DistS(v, ch(v)) + 1$, the subtree $T_{p(v),m-1}$ is resolved by some sensor in $S(v, ch(v))$. Naturally, the sensors $S(v, ch(v))$ already resolve $T_{v,ch(v)}$, so both subtrees are resolved individually and the new value for $l(p(v), m)$ can be any value. Then we use *Check-resolving* to find the optimal l'. First, we set $l' \leftarrow null$ and run *Check-resolving*. If *Check-resolving* returns True, continue. If *Check-resolving* returns False, we set $l' \leftarrow k$, run *Check-resolving* again and continue this loop, decreasing l' on each iteration until *Check-resolving* returns True or until *Check-resolving* returns False for $l' = 1$.

(6.2) When $l(p(v), m-1) < 1 + DistS(v, ch(v))$, the subtree $T_{p(v),m-1}$ is not resolved by any sensor in $S(v, ch(v))$. Then the merged subtree still needs a sensor at distance no greater than $l(p(v), m-1)$ from $p(v)$ to resolve all vertex-vertex pairs in $T_{p(v),m-1}$.
We set $l' \leftarrow l(p(v), m-1)$.

(Case 7) $l(p(v), m-1) = null$ **and** $l(v, ch(v)) = null$**.**
Each of the subtrees individually is resolved. Because every vertex is measured by a sensor in its own subtree, we know *Check-resolving* will return True for $l' = null$. This statement will be elaborated on

in Claim 3.12 and proved in Section 4.1.2.
We set $l' \leftarrow null$.

3.7.4 Pseudocode for Check-cases algorithm

Algorithm 3 Check-cases for merging subtrees

The algorithm is processing a vertex V with parent vertex P. It merges the subtrees $T_{P,m-1}$ and $T_{V,ch(V)}$ into the single subtree $T_{P,m}$.
if $l(P, m-1) \geq DistS(V, ch(V)) + 1$ and $l(V, ch(V)) \geq DistS(P, m-1) + 1$ **then**
  $l' \leftarrow null$
else if $l(P, m-1) \geq DistS(V, ch(V)) + 1$ and $l(V, ch(V)) < 1 + DistS(P, m-1)$ **then**
  $l' \leftarrow l(V, ch(V)) - 1$
else if $l(P, m-1) < 1 + DistS(V, ch(V))$ and $l(V, ch(V)) \geq DistS(P, m-1) + 1$ **then**
  $l' \leftarrow l(P, m-1)$
else if $l(P, m-1) < DistS(V, ch(V)) + 1$ and $l(V, ch(V)) < DistS(V, ch(V)) + 1$ **then**
  $l' \leftarrow \min(l(P, m-1), l(V, ch(V)) - 1)$
else if $l(P, m-1) = null$ and $l(V, ch(V)) \neq null$ **then**
  if $l(V, ch(V)) \geq DistS(P, m-1) + 1$ **then**
    $l' \leftarrow null$
  else
    $l' \leftarrow l(V, ch(V)) - 1$
  end if
else if $l(P, m-1) \neq null$ and $l(V, ch(V)) = null$ **then**
  if $l(P, m-1) \geq DistS(V, ch(V)) + 1$ **then**
    $l' \leftarrow null$
  else
    $l' \leftarrow l(P, m-1)$
  end if
else if $l(P, m-1) = null$ and $l(V, ch(V)) = null$ **then**
  $l' \leftarrow null$
end if

With the explanation for the *Check-cases* algorithm complete, we move on to the second sub-algorithm: *Check-resolving*.

3.7.5 Heuristic explanation of Check-resolving for merging subtrees

The *Check-resolving* algorithm checks whether, when merging subtrees $T_{p(v),m-1}$ and $T_{v,ch(v)}$, the sensors in the two subtrees combined with a vertex at distance l' from the root $p(v)$ of the merged subtree is indeed a k-resolving set for $T_{p(v),m}$, as required to proceed with the algorithm. The natural way to do this is by checking for each vertex-vertex pair possible in the merged subtree $T_{p(v),m}$ if the pair is distinguished by the set of sensors in $T_{p(v),m}$ or not. In practice, the algorithm can skip certain vertex-vertex pairs because we know they are already distinguished, such as the vertex-vertex pairs where both vertices are from the same subtree. This is due to Claim 3.11 combined with the fact that the output of *Check-cases* is a value l' which is smaller or equal to any of the value(s) of l of the subtree(s) which are not resolved by any sensor in the other subtree. So when merging subtrees, the value of l' which *Check-cases* determines is such that each of the two subtrees individually are resolved by the set of sensors in the merged subtree $S(p(v), m)$ and a sensor at distance l' from the root of the merged subtree, $p(v)$. This implies that *Check-resolving* only needs to check vertex-vertex pairs in $T_{p(v),m}$ where the two vertices are not from the same subtree before merging.

For these vertex-vertex pairs in $T_{p(v),m}$ where the vertices are not from the same subtree before merging we separate four cases:

(Case 1) **x is measured by its own subtree $S(p(v), m-1)$, y is measured by its own subtree $S(v, ch(v))$.** We state the following claim, proven in Section 4:

Claim 3.12. *Suppose the algorithm is processing a vertex v and merges subtrees $T_{v.ch(v)}$ and $T_{p(v),m-1}$. Let $x \in V(T_{p(v),m-1})$ be measured by any sensor in $S(p(v), m-1)$ and let $y \in V(T_{v,ch(v)})$ be measured by any sensor in $S(v, ch(v))$. Then x, y are distinguished by $S(p(v), m) = S(p(v), m-1) \cup S(v, ch(v))$.*

This claim implies this type of vertex-vertex pair must always be distinguished from each other, so *Check-resolving* does not need to run any checks.

(Case 2) x is not measured by any sensor in its own subtree $S(p(v), m-1)$, y is not measured by any sensor in its own subtree $S(v, ch(v))$.
Within Case 2, we separate three subcases. If the checks of one of the subcases is true, we conclude x, y are distinguished and *Check-resolving* continues with checking the next vertex-vertex pair. If none of the subcases are true, x, y are not distinguished by $S(p(v), m) \cup \{s^*\}$, so *Check-resolving* returns False.

(2.1) x is measured by a sensor in $S(v, ch(v))$.
For this, we check if the closest sensor to v in $S(v, ch(v))$ can measure x.

$$d(x, p(v)) + 1 + DistS(v, ch(v) \leq k$$

(2.2) y is measured by a sensor in $S(p(v), m-1)$.
For this, we check if the closest sensor to $p(v)$ in $S(p(v), m-1)$ can measure y.

$$d(y, v) + 1 + DistS(p(v), m-1) \leq k$$

(2.3) x, y are distinguished by a vertex corresponding to l'.
A sensor on a vertex corresponding to l' would not distinguish x, y iff it would not measure either x or y, or if it measures both x and y to be equally far. We check if

$$min(k+1, d(x, p(v)) + l') \neq min(k+1, d(y, v) + 1 + l')$$

(Case 3) x is measured by its own subtree $S(p(v), m-1)$, y is not measured by any sensor in its own subtree $S(v, ch(v))$.
We state a claim which narrows down the vertex-vertex pairs within Case 3 and Case 4 which *Check-resolving* needs to check.

Claim 3.13. *Suppose the algorithm is processing a vertex v and merges subtrees $T_{p(v),m-1}$ and $T_{v,ch(v)}$. If a vertex x is measured and $x \notin Pr_1(v, ch(v)) \cup Pr_f(v, ch(v)) \cup Pr_2(v, ch(v))$, then it is distinguished from any vertex $y \in NotM(p(v), m-1)$. Conversely, if a vertex y is measured and $y \notin Pr_1(p(v), m-1) \cup Pr_f(p(v), m-1) \cup Pr_2(p(v), m-1)$, then it is distinguished from any vertex $x \in NotM(v, ch(v))$.*

We prove this claim in Section 4.

Therefore, we can assume $x \in Pr_1(p(v), m-1) \cup Pr_f(p(v), m-1) \cup Pr_2(p(v), m-1)$. Remember that this means $(d(p(v), x), d(p(v), s))$ is known (Definition 3.3.1). Now, we separate four subcases:

(3.1) $x \in Pr_1(p(v), m-1)$.
Then x is measured only though $p(v)$. x, y are distinguished iff they do not have the same distance from $p(v)$. We check if

$$d(x, p(v)) \neq d(y, v) + 1$$

When this check fails, x, y are not distinguished and the algorithm returns False.

(3.2) $x \in Pr_f(p(v), m-1)$.
Let $s \in S(p(v), m-1)$ be a sensor measuring x, and $w(v, x)$ the pivotal vertex through which x is measured. Note that, because $x \in Pr_f(p(v), m-1)$, $d(p(v), w(v, x))$ and $d(x, w(v, x))$ are known (Definition 3.3.1). Then x, y are distinguished if they do not have the same distance from the pivotal vertex $w(v, x)$. We check if

$$\begin{aligned} d(y, s) =& d(y, p(v)) + d(p(v), s) \neq d(x, s) \\ =& d(y, v) + 1 + d(p(v), s) \neq d(x, s) \\ \iff & d(y, v) + 1 + d(p(v), w(v, x)) \neq d(x, w(v, x)) \end{aligned}$$

When this inequality does not hold, x, y can still be distinguished by a sensor on a vertex corresponding to l' if such a sensor measures y. We check if

$$d(v, y) + 1 + l' \leq k$$

If both checks fail, x, y are not distinguished and the algorithm returns False.

(3.3) $x \in Pr_2(p(v), m-1)$.
Then x is measured only through $w(x, p(v))$ (Definition 3.3.1). By definition of Pr_2, x, y are distinguished if y is not distance one from $p(v)$.

$$d(y, p(v)) \neq 1 \iff d(y, v) \neq 0$$

When this check fails, x, y are distinguished iff a sensor on a vertex corresponding to l' measures y, we check that

$$l' \leq k - 1$$

If both checks fail, x, y are not distinguished and the algorithm returns False. Both of these checks do not depend on x, so in the implementation of the algorithm and in the pseudocode we do not iterate over $Pr_2(p(v), m-1)$, but instead we check if $Pr_2(p(v), m-1)$ is empty or not. If it is not empty, we do the two checks explained in this case, if it is empty, we do not perform the checks.

(Case 4) **x is not measured by any sensor in its own subtree $S(p(v), m-1)$, y is measured by its own subtree $S(v, ch(v))$.**
This case is analogous to Case 3.

(4.1) $y \in Pr_1(v, ch(v))$.
Then y is measured only through v. x, y are distinguished if they do not have the same distance from v. We check if

$$d(y, v) \neq d(x, p(v)) + 1$$

When this check fails, x, y can still be distinguished by a sensor on a vertex corresponding to l' iff a sensor at distance l' from $p(v)$ measures x. We check that

$$d(x, p(v)) + l' \leq k$$

When both checks fail, x, y are not distinguished and the algorithm returns False.

(4.2) $y \in Pr_f(v, ch(v))$.
Let s be a sensor measuring y, and $w(v, y)$ the pivotal vertex through which y is measured. Because $y \in Pr_f(v, ch(v))$, $d(v, w(v, y))$ and $d(y, w(v, y))$ are known (Definition 3.3.1). Then x, y are distinguished if they do not have the same distance from the pivotal vertex $w(v, y)$. We check if

$$\begin{aligned} d(x, s) = d(x, p(v)) + 1 + d(v, s) &\neq d(y, s) \\ \iff d(x, p(v)) + 1 + d(v, w(v, y)) &\neq d(y, w(v, y)) \end{aligned}$$

When this check fails, x, y can still be distinguished by a sensor on a vertex corresponding to l' iff a sensor at distance l' from $p(v)$ measures x. We check that

$$d(x, p(v)) + l' \leq k$$

When both checks fail, x, y are not distinguished and the algorithm returns False.

(4.3) $y \in Pr_2(v, ch(v))$.
Then y is measured only through pivotal vertex $w(y, v)$ and this pivotal vertex has the property that $d(y, w(v, y)) = d(p(v), w(v, y))$ (Definition 3.3.1). By definition then, all vertices in $T_{p(v),m-1}$ except $p(v)$ are distinguished from y. We check if x is $p(v)$:

$$d(x, p(v)) \neq 0$$

When this check fails, x, y can still be distinguished by a sensor on a vertex correspondnig to l' iff a sensor at distance l' from $p(v)$ measures x. We check that

$$l' \neq null$$

When both checks fail, x, y are not distinguished and the algorithm returns False. Both of these checks do not depend on y, so in the implementation of the algorithm and in the pseudocode we do not iterate over $Pr_2(v, ch(v))$, but instead we check if $Pr_2(v, ch(v))$ is empty or not. If it is not empty, we do the two checks explained in this Case, if it is empty, we do not perform the checks.

3.7.6 Pseudocode for Check-resolving algorithm

Algorithm 4 Check-resolving for merging subtrees

The algorithm is processing a vertex V with parent vertex P. It merges the subtrees $T_{P,m-1}$ and $T_{V,ch(V)}$ into the single subtree $T_{P,m}$.
for $x \in NotM(P, m-1), y \in NotM(V, ch(V))$ **do**
  if $d(x, P+1+DistS(V, ch(V)) \leq k$ **then**
    Continue
  else if $d(y, V)+1+DistS(P, m-1) \leq k$ **then**
    Continue
  else if $min(k+1, d(x, P)+l') \neq min(k+1, d(y, V)+1+l')$ **then**
    Continue
  else
    x, y are not distinguished.
    return FALSE
  end if
end for
for $x \in Pr_1(P, m-1), y \in NotM(V, ch(V))$ **do**
  if $d(x, P) \neq d(y, V)+1$ **then**
    Continue
  else
    return FALSE
  end if
end for
for $x \in Pr_f(P, m-1), y \in NotM(V, ch(V))$ **do**
  if $d(y, V)+1+d(P, w(V, x)) \neq d(x, w(x, V)) \vee d(V, y)+1+l' \leq k$ **then**
    Continue
  else
    return FALSE
  end if
end for
for $x \in Pr_2(P, m-1), y \in NotM(V, ch(V))$ **do**
  if $d(y, V) \neq 0 \vee l' \leq k-1$ **then**
    Continue
  else
    return FALSE
  end if
end for
for $x \in NotM(P, m-1), y \in Pr_1(V, ch(V))$ **do**
  if $d(y, V) \neq d(x, P)+1 \vee d(x, P)+l' \leq k$ **then**
    Continue
  else
    return FALSE
  end if
end for

42: **for** $x \in NotM(P, m-1), y \in Pr_f(V, ch(V))$ **do**
43: **if** $d(x,P) + 1 + d(V, w(V,y)) \neq d(y, w(V,y)) \vee d(x,P) + l' \leq k$ **then**
44: Continue
45: **else**
46: **return** FALSE
47: **end if**
48: **end for**
49: **for** $x \in NotM(P, m-1), y \in Pr_2(V, ch(V))$ **do**
50: **if** $d(x,P) \neq 0 \vee l' \neq null$ **then**
51: Continue
52: **else**
53: **return** FALSE
54: **end if**
55: **end for**
56: After performing all checks, if the algorithm has not yet returned False, it returns True.
57: **return** TRUE

We prove the claims required to show this sub-algorithm works in Section 4.

3.8 Processing the root

When the algorithm is processing the root v of the tree which the algorithm was given as input, it is not possible to add the parent vertex of the root or to merge subtrees with the parent of the root as the root has no parent. The way we process the root is as follows: If $l(v, ch(v)) \neq null$, the algorithm places a sensor on v to make sure the output of the algorithm is a k-resolving set. If $l(v, ch(v)) = null$, the sensor set is already a k-resolving set for the tree so no sensor is placed.

3.9 Algorithm for finding k-truncated metric dimension

Finally, we put everything together for the pseudo-code for the full algorithm for finding the k-truncated metric dimension of a tree.

Algorithm 5 Algorithm for finding k-truncated metric dimension

Require: $k > 0, k \in \mathbb{N}$. EL and AL are the endvertex and its associated adjacency list, respectively. Set $I = 0$

STEP 2

1: $I \leftarrow I + 1$
2: $V \leftarrow EL(I)$
3: $P \leftarrow AL(I)$
4: **if** $V = P$ and $l(V, ch(V)) \neq null$ **then**
5: $S(V, ch(V)) \leftarrow S(V, ch(V)) \cup \{V\}$
$l(V, ch(V)) \leftarrow null$.
Go to Terminating Step (line 106).
6: **else if** V=P **then**
7: Go to Terminating Step (line 106).
8: **end if**
9: **if** $ch(V) = 0$ **then**
10: **if** $Seen(P) = 0$ **then**
11: $S(V,0) = S(P,1) \leftarrow \emptyset$
12: $l(V,0) \leftarrow k$, $l(P,1) \leftarrow k-1$
Update all items corresponding to $T_{P,1}$.
13: $Seen(P), Seen(V) \leftarrow 1$
14: Return to Step 2
15: **else if** $Seen(P) = 1$ **then**

16: $S(V,0) \leftarrow \emptyset$
$l(V,0) \leftarrow k$
Update all items corresponding to $T_{J,0}$.
17: $l' \leftarrow Check-cases(S(P,m-1), l(P,m-1), DistS(P,m-1), S(V,0), l(V,0), DistS(V,0))$
18: **if** $Check-resolving(S(P,m-1), S(V,0), l') = TRUE$ **then**
19: $S(P,m) \leftarrow S(P,m-1) \cup S(V,0)$
20: $l(P,m) \leftarrow l'$
21: Update all items corresponding to $T_{P,m}$.
22: **else**
23: **if** $l' \neq null$ **then**
24: $l' \leftarrow l' - 1$
25: **while** $Check-resolving(S(P,m-1), S(V,0), l(V,0), l') = FALSE$ **do**
26: **if** $l' \neq 1$ **then**
27: $l' \leftarrow l' - 1$
28: Continue the while-loop
29: **else**
30: $S(V,0) \leftarrow S(V,0) \cup \{V\}$.
Update all items corresponding to $T_{V,0}$.
Go back to line 20.
31: **end if**
32: **end while**
33: **else**
34: $l' \leftarrow k$
35: **while** $Check-resolving(S(P,m-1), S(V,ch(V)), l') = FALSE$ **do**
36: **if** $l' \neq 1$ **then**
37: $l' \leftarrow l' - 1$
38: Continue the while-loop
39: **else**
40: $S(V,0) \leftarrow S(V,0) \cup \{V\}$.
Update all items corresponding to $T_{V,0}$.
Go back to line 20.
41: **end if**
42: **end while**
43: **end if**
44: **end if**
45: **end if**
46: **end if**
47: *Now V is not a leaf*
48: **if** $Seen(P) = 0$ **then**
49: **if** $l(V, ch(V)) = 1$ **then**
50: $S(P,1) \leftarrow S(V, ch(V)) \cup \{P\}$
51: $l(P,1) \leftarrow k$
52: Update all items corresponding to $T_{P,1}$.
53: Return to Step 2
54: **else if** $2 \leq l \leq k$ **then**
55: $S(P,1) \leftarrow S(V, ch(V))$
56: $l(P,1) \leftarrow l(V, ch(V)) - 1$
57: Update all items corresponding to $T_{P,1}$.
58: Return to Step 2
59: **else if** $l(V, ch(V)) = null$ **then**
60: To determine $l(P,1)$, we check whether $S(V, ch(V))$ distinguishes P and $v, \forall v \in V(T_{V,ch(V)})$
61: **for** $(d(V,x), d(V,s)) \in Pr_1(V, ch(V))$ **do**
62: **if** $d(V,x) \neq 1$ **then**
63: x and P are distinguished.

64: **if** $Pr_2(V, ch(V)) = \emptyset$ **then**
65: $S(V, ch(V))$ is a k-resolving set for $T_{P,1}$.
$l(P, 1) \leftarrow null$.
Update all items corresponding to $T_{P,1}$.
$Seen(P) = 1$
66: **else**
67: $x \in Pr_2(V, ch(V))$ and P are not distinguished.
$l(P, 1) \leftarrow k$
$S(P, 1) \leftarrow S(V, ch(V))$
Update all items corresponding to $T_{P,1}$.
$Seen(P) = 1$
68: **end if**
69: **else**
70: x and P are not distinguished.
$l(P, 1) \leftarrow k$
$S(P, 1) \leftarrow S(V, ch(V))$
Update all items corresponding to $T_{P,1}$.
$Seen(P) = 1$
71: **end if**
72: **end for**
73: **end if**
74: **else if** $Seen(P) = 1$ **then**
75: $l' \leftarrow Check-cases(S(P, m-1), l(P, m-1), DistS(P, m-1), S(V, ch(V)), l(V, ch(V)), DistS(V, ch(V)))$

76: **if** $Check - resolving(S(P, m-1), S(V, ch(V)), l') = TRUE$ **then**
77: $S(P, m) \leftarrow S(P, m-1) \cup S(V, ch(V))$
78: $l(P, m) \leftarrow l'$
79: Update all items corresponding to $T_{P,m}$.
80: **else**
81: **if** $l' \neq null$ **then**
82: $l' \leftarrow l' - 1$
83: **while** $Check - resolving(S(P, m-1), S(V, ch(V)), l') = FALSE$ **do**
84: **if** $l' \neq 1$ **then**
85: $l' \leftarrow l' - 1$
86: Continue the while-loop
87: **else**
88: $S(V, ch(V)) \leftarrow S(V, ch(V)) \cup \{V\}$.
Update all items corresponding to $T_{V,ch(V)}$.
Run this algorithm again for V and P.
89: **end if**
90: **end while**
91: **else**
92: $l' \leftarrow k$
93: **while** $Check - resolving(S(P, m-1), S(V, ch(V)), l') = FALSE$ **do**
94: **if** $l' \neq 1$ **then**
95: $l' \leftarrow l' - 1$
96: Continue the while-loop
97: **else**
98: $S(V, ch(V)) \leftarrow S(V, ch(V)) \cup \{V\}$.
Update all items corresponding to $T_{V,ch(V)}$.
Run this algorithm again for V and P.
99: **end if**
100: **end while**
101: **end if**

end if
end if
Terminating Step
return $S(I, ch(I))$, the k-resolving set.

Now, having explained and given the pseudocode for the full algorithm, we continue in Section 4 with the analysis and proof of this algorithm.

4 Proving the algorithms validity

Proving the algorithm requires proving two statements. We give the first statement now as a proposition, then give the proof in Section 4.1. The second statement has not been proved yet, so we will present it as a conjecture.

Proposition 4.1. *For any subtree $T_{v,m}$ for which the algorithm has allocated a set of sensors $S(v,m)$ and a value for $l(v,m)$, $S(v,m)$ and a sensor outside $T_{v,m}$ at distance $l(v,m)$ from v fully resolve $T_{v,m}$.*

This proposition implies the output of the algorithm is a k-resolving set for the tree given as input. We repeat Conjecture 2.8 about optimality of the algorithm:

Conjecture. *Fix $k \in \mathbb{N}$. Given a tree T of size n, there is an $\mathcal{O}(n)$ algorithm which finds a k-resolving set for T of size at most one greater than the truncated metric dimension of T.*

We will present our reasons for this conjecture in Section 4.2.

4.1 Proving Proposition 4.1

We will prove Proposition 4.1 inductively, by showing that if Proposition 4.1 holds for some subtree $T_{v,ch(v)}$ (and $T_{p(v),m-1}$, if the algorithm has already allocated $S(p(v), m-1)$ and $l(p(v), m-1)$), the algorithm maintains that assumption after adding the parent vertex of v to the subtree or after merging $T_{v,ch(v)}$ together with $T_{p(v),m-1}$. To prove Proposition 4.1 inductively, we need a proof of the base case, which would be the leaves of the tree.

Claim 4.2. *Suppose the algorithm processes a leaf-vertex v. Then $T_{v,ch(v)}$ is resolved by a sensor at distance k from v.*

Proof. Any sensor at distance k from the leaf v measures v, and since v is the only vertex in $T_{v,0}$, that sensor fully resolves $T_{v,0}$. □

4.1.1 Proving Proposition 4.1 for adding the parent vertex

In the explanation for adding the parent vertex (Section 3.6.1), in Claim 3.10 we stated that when $l(v, ch(v)) = null$, it suffices to only check the vertices in $Pr(v, ch(v))$ are distinguished from $p(v)$. We now give a proof of this claim:

Proof of Claim 3.10. Let $x \notin Pr((v, ch(v))$. We will prove x must be distinguished from $p(v)$ by the sensors in the subtree $S(v, ch(v))$ by separating two cases:

1. If the pivotal vertex of x does not exist, there must be some $s_1, s_2 \in S(v.ch(v))$ such that $P_{xv} \cap P_{xs_1} \neq P_{xv} \cap P_{xs_2}$ (Definition 3.6). We define two vertices by where the path from x to v and the path from x to s_1, s_2 split, respectively:

$$P_{xw_1} \coloneqq P_{xv} \cap P_{xs_1}$$
$$P_{xw_2} \coloneqq P_{xv} \cap P_{xs_2}$$

We show either s_1 or s_2 distinguishes x and $p(v)$. Suppose s_1 does not distinguish x and $p(v)$, so $|P_{s_1x}| = |P_{s_1p(v)}|$. These paths split at vertex w_1, so this implies $|P_{w_1x}| = |P_{w_1p(v)}|$. We write the paths from s_2 to x and $p(v)$ as

$$\begin{aligned} P_{s_2x} &= P_{s_2w_2} \cup P_{w_2x} \\ P_{s_2p(v)} &= P_{s_2w_2} \cup P_{w_2p(v)} \end{aligned} \tag{4.1}$$

Both w_1 and w_2 are on the path from x to $p(v)$ and by assumption $w_1 \neq w_2$, therefore $|P_{w_2x}| \neq |P_{w_2p(v)}|$ (no two vertices on the same path can have the same distance to the endpoints of that path). Looking at Equation 4.1 it becomes clear this implies s_2 distinguishes x and $p(v)$.

2. If the pivotal vertex of x does exist, for all sensors the path from x to that sensor contains the pivotal vertex $w(x, v)$. We assumed $x \notin Pr(v, ch(v))$, so $w(x, v) \neq v$ and $d(w(x, v), x) \neq d(w(x, v), v) + 1$. Let s be any sensor measuring x. We write the paths from s to x and $p(v)$ respectively:

$$\begin{aligned} P_{sx} &= P_{sw(x,v)} \cup P_{w(x,v)x} \\ P_{sp(v)} &= P_{sw(x,v)} \cup P_{w(x,v)p(v)} \end{aligned}$$

The sensor s distinguishes x and $p(v)$ if $|P_{w(x,v)x}| \neq |P_{w(x,v)p(v)}|$, which holds by our assumption that $x \notin Pr(v, ch(v))$.

□

Remark 4.3. *This implies that in the situation that $l(v, ch(v)) = null$, in order to check if $p(v)$ is distinguished from all vertices in $V(T_{v,ch(v)})$ it suffices to check if all vertices in $Pr(v, ch(v))$ are distinguished from $p(v)$.*

We will prove inductively Proposition 4.1 holds for the add-parent operation by case distinction. We separate cases based on the value of $l(v, ch(v))$.

Claim 4.4. *Suppose the algorithm has processed all children of v and, while processing some vertex earlier (the last child of v to be exact), allocated $S(v, ch(v))$ and $l(v, ch(v)) = 1$ such that $S(v, ch(v)) \cup \{s^*\}$, where s^* is a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v, forms a k-resolving set for $T_{v,ch(v)}$. The algorithm now processes v and decides to add the parent vertex of v, $p(v)$, to the subtree. Then $T_{p(v),1}$ is resolved by $S(v, ch(v)) \cup \{p(v)\}$.*

Proof. The induction hypothesis states that the algorithm has allocated $S(v, ch(v))$ and $l(v, ch(v))$ such that $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v is a k-resolving set for $T_{v,ch(v)}$. We see that $l(v, ch(v)) = 1$, and $p(v)$ is the only vertex outside of $T_{v,ch(v)}$ with distance 1 from v, so by the induction hypothesis $T_{v,ch(v)}$ is resolved by $S(v, ch(v)) \cup \{p(v)\}$. When a sensor is placed on $p(v)$ it must be measured and distinguished from all vertices in $T_{v,ch(v)}$, so $S(v, ch(v)) \cup \{p(v)\}$ forms a k-resolving set for $T_{p(v),1}$. Thus $l(p(v), 1) = null$ and the induction is advanced for the new subtree $T_{p(v),1}$. □

Claim 4.5. *Suppose the algorithm has processed all children of v and, while processing some vertex earlier (the last child of v to be exact), allocated $S(v, ch(v))$ and $2 \leq l(v, ch(v)) \leq k$ such that $S(v, ch(v)) \cup \{s^*\}$, where s^* is a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v, forms a k-resolving set for $T_{v,ch(v)}$. The algorithm now processes v and decides to add the parent vertex of v, $p(v)$, to the subtree. Then $T_{p(v),1}$ is resolved by $S(v, ch(v))$ and a sensor at distance $l(v, ch(v)) - 1$ from $p(v)$.*

Proof. The induction hypothesis states that the algorithm has allocated $S(v, ch(v))$ and $l(v, ch(v))$ such that $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v (we will denote this sensor as s^*) is a k-resolving set for $T_{v,ch(v)}$. We know $d(p(v), s^*) = l(v, ch(v)) - 1$ because $p(v)$ must be on the shortest path from v to s^*, and because $l(v, ch(v))$ is not greater than k, we can conclude $p(v)$ is measured by s^*. Moreover, $p(v)$ is distinguished from all other vertices in $T_{v,ch(v)}$ by being the closest vertex to s^* in $T_{p(v),1}$. We conclude $T_{p(v),1}$ is resolved by $S(v, ch(v)) \cup s^*$, where s^* is a vertex outside $T_{p(v),1}$ at distance $l(v, ch(v)) - 1$ from $p(v)$. Thus $l(p(v), 1) = l(v, ch(v)) - 1$ and the induction is advanced for the new subtree $T_{p(v),1}$. □

Claim 4.6. *Suppose the algorithm has processed all children of v and, while processing some vertex earlier (the last child of v to be exact), allocated $S(v, ch(v))$ and $l(v, ch(v)) = null$ in such a way that $S(v, m)$ forms a k-resolving set for $T_{v,ch(v)}$. The algorithm now processes v and decides to add the parent vertex of v, $p(v)$, to the subtree. Then $T_{p(v),1}$ is resolved by $S(v, ch(v))$ if either of the following two cases is true:*

1. *$DistS(v, ch(v)) \leq k-1$ and $Pr(v, ch(v)) = \emptyset$. So when the parent of v, $p(v)$, is measured by a sensor in $S(v, ch(v))$ and there are no problematic vertices (Definition 3.3.1).*
2. *$DistS(v, ch(v)) \leq k-1$ and $Pr_2(v, ch(v)) = \emptyset$ and $Pr_1(v, ch(v)) \neq \emptyset$, $\forall (d(v, x), d(v, s) \in Pr_1(v, ch(v)) : d(v, x) \neq 1$. So when the parent of v, $p(v)$, is measured by a sensor in $S(v, ch(v))$ and there are only type-1 problematic vertices, and none of them are adjacent to v.*

If neither case is true, i.e. if $p(v)$ is not measured by any sensor in $S(v, ch(v))$, $Pr_2(v, ch(v))$ is non-empty or there is some type-1 problematic vertex which is adjacent to v, $T_{p(v),1}$ is resolved by $S(v, ch(v))$ and a sensor at distance at most k away from $p(v)$.

Proof. The induction hypothesis states that the algorithm has allocated $S(v, ch(v))$ and $l(v, ch(v))$ such that $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v is a k-resolving set for $T_{v,ch(v)}$. In the case that $l(v, ch(v)) = null$, $S(v, ch(v))$ is a k-resolving set for $T_{v,ch(v)}$.

1. Through Claim 3.10, it is clear that if the induction hypothesis, $DistS(v, ch(v)) \leq k-1$ and $Pr(v, ch(v)) = \emptyset$ all hold, $T_{p(v),1}$ is resolved by $S(v, ch(v))$, thus $l(p(v), 1) = null$ and the induction is advanced for the new subtree $T_{p(v),1}$.
2. Let $(d(v, x), d(v, s) \in Pr_1(v, ch(v))$ such that $d(v, x) \neq 1$. For all sensors in $S(v, ch(v))$ which measure $p(v)$, v is on the path from sensor to $p(v)$. By definition of type-1 problematic vertices, the same holds for x (Definition 3.3.1). For all $s \in S(v, ch(v))$ which measure x:

$$P_{sx} = P_{sv} \cup P_{vx}$$
$$P_{sp(v)} = P_{sv} \cup P_{vp(v)}$$
$$|P_{vx}| \neq 1 = |P_{vp(v)}|$$

   So x and $p(v)$ are distinguished by $S(v, ch(v))$. Combining this with the induction hypothesis $S(v, ch(v))$, it becomes clear $S(v, ch(v))$ is a k-resolving set for $T_{p(v),1}$. Therefore $l(p(v), 1) = null$ and the induction is advanced for the new subtree $T_{p(v),1}$.

If neither case is true either $p(v)$ is not measured by any sensor in $S(v, ch(v))$, there is a type-2 problematic vertex or $p(v)$ is not distinguished from some vertex $x \in Pr_1(v, ch(v))$ at distance 1 from v. Either way, we can measure and distinguish $p(v)$ from all vertices in $T_{v,ch(v)}$ with a sensor at distance at most k from $p(v)$. This sensor measures $p(v)$ to be the closest sensor to itself in the subtree $T_{p(v),1}$, thereby distinguishing $p(v)$ from any vertex in $T_{v,ch(v)}$. Combining this with the induction hypothesis, we see $T_{p(v),1}$ is resolved by $S(v, ch(v))$ and a sensor at distance at most k from $p(v)$. Therefore $l(p(v), 1) = k$ and the induction is advanced for the new subtree $T_{p(v),1}$. □

We combine all claims to prove that adding the parent vertex of v to the subtree $T_{v,ch(v)}$ maintains the assumption that Proposition 4.1 holds.

Claim 4.7. *Suppose the algorithm is processing a vertex v and, while processing some vertex earlier (the last child of v to be exact), allocated $S(v, ch(v))$, $l(v, ch(v))$ such that $S(v, ch(v))$ combined with a sensor outside $T_{v,ch(v)}$ at distance $l(v, ch(v))$ from v k-resolves $T_{v,ch(v)}$. Then after adding the parent vertex $p(v)$ to the subtree, the algorithm allocates sensors and a value for $l(p(v), 1)$ such that $S(p(v), 1)$ combined with a sensor outside $T_{p(v),1}$ at distance $l(p(v), 1)$ also k-resolves $T_{p(v),1}$.*

Proof. Combining Claims 3.10, 4.2, 4.4, 4.5, 4.6, it follows that $S(p(v), 1)$ combined with a sensor outside $T_{p(v),1}$ at distance $l(p(v), 1)$ k-resolves $T_{p(v),1}$. □

4.1.2 Proving Proposition 4.1 for merging subtrees

Proving Proposition 4.1 for merging subtrees relies on proving the *Check-cases* and *Check-resolving* algorithms work as intended. Before we can prove Proposition 4.1 for merging subtrees, we need to prove some claims which decrease the number of vertex-vertex pairs the *Check-resolving* algorithm needs to iterate over. We start by proving Claim 3.11 given in Section 3.7.3, which states that when the algorithm has allocated $S(v,m)$ and $l(v,m)$ for some subtree $T_{v,m}$ such that $S(v,m)$ combined with a sensor outside $T_{v,m}$ at distance $l(v,m)$ from v resolves $T_{v,m}$, $S(v,m)$ combined with a sensor outside $T_{v,m}$ at any distance smaller or equal to $l(v,m)$ also resolves $T_{v,m}$.

Proof of Claim 3.11. Denote a vertex outside $T_{v,m}$ at distance $l(v,ch(v))$ from v as s^*. Denote a vertex outside $T_{v,m}$ at distance l' from v as s' (s' can also be a sensor on the vertex v if we so choose). Because $l' \leq l(v,ch(v))$:

$$\forall x \in V(T_{v,m}) : P_{s^*x} = P_{s^*s'} \cup P_{s'x}$$

Let $x, y \in V(T_{v,m})$, then

$$d_k(x,s^*) \neq d_k(y,s^*) \implies d_k(x,s') \neq d_k(y,s') \tag{4.2}$$

By assumption $S(v,m) \cup \{s^*\}$ resolves $T_{v,m}$. Through (4.2), it becomes clear $S(v,m) \cup \{s'\}$ also resolves $T_{v,m}$. □

Remark 4.8. *This claim implies that when merging two subtrees together, both subtrees individually are resolved by the set of sensors inside that subtree and a vertex at distance l' away from the root of the merged subtree, with l' the output of the* Check-cases *algorithm. Therefore, the* Check-resolving *algorithm does not need to check vertex-vertex pairs where both vertices are from the same subtree.*

In Claim 3.12, we state that when the algorithm is processing a vertex v and merges subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$, if any vertex-vertex pair x, y exists where $x \in T_{p(v),m-1}$ and x is measured by some sensor in $S(p(v),m-1)$, $y \in T_{v,ch(v)}$ and y is measured by some sensor in $S(v,ch(v))$, then x and y are distinguished from each other by the sensors in the merged subtree. We now give a proof of this claim.

Proof of Claim 3.12. We repeat the proof which was given in [6] by Gutkovich and Yeoh. Denote the sensor in $S(p(v),m-1)$ which measures x as s_x, and the sensor in $S(v,ch(v))$ which measures y as s_y. We will show one of s_x, s_y distinguishes x and y.

The proof is a proof by contradiction. Assume $d(x,s_x) = d(y,s_x)$ and $d(x,s_y) = d(y,s_y)$. Let $a = lca(x,s_x)$, $b = lca(y,s_y)$. *lca* stands for lowest common ancestor, a term first introduced in [1] by Aho, Hopcroft and Ullman. Due to the nature of a tree, any path between two vertices with no repeating vertex is the unique shortest path between two vertices. This allows us to express the distances from x,y to the sensors s_x, s_y as the following:

$$\begin{aligned} d(y,s_x) &= d(y,a) + d(a,s_x) \\ d(x,s_x) &= d(x,a) + d(a,s_x) \\ d(y,s_y) &= d(y,b) + d(b,s_y) \\ d(x,s_y) &= d(x,b) + d(b,s_y) \end{aligned}$$

Now, we are ready to show a contradiction.

$$\begin{aligned} 0 &= d(x,s_x) - d(y,s_x) \\ &= d(x,a) - d(y,a) \\ &= (d(x,s_y) - d(a,s_y)) - d(y,a) \\ &= d(y,s_y) - d(a,s_y) - d(y,a) \\ &= d(y,b) - d(a,b) - d(y,a) \\ &= -2d(a,b) < 0, \end{aligned}$$

a contradiction, as desired. □

Remark 4.9. *This claim implies that when merging subtrees together,* Check-resolving *does not need to iterate over any vertex-vertex pairs where the vertices are not from the same subtree but both are measured by their respective subtrees.*

In Claim 3.13 we state the following: Suppose the algorithm is processing a vertex v and merges subtrees $T_{p(v),m-1}$ and $T_{v,ch(v)}$. If a vertex x is measured and $x \notin Pr_1(v, ch(v)) \cup Pr_f(v, ch(v)) \cup Pr_2(v, ch(v))$, then it is distinguished from any vertex $y \in NotM(p(v), m-1)$. Conversely, if a vertex y is measured and $y \notin Pr_1(p(v), m-1) \cup Pr_f(p(v), m-1) \cup Pr_2(p(v), m-1)$, then it is distinguished from any vertex $x \in NotM(v, ch(v))$. We now give a proof of this claim.

Proof. Let $y \in NotM(p(v), m-1)$, x measured and $x \notin Pr_1(v, ch(v)) \cup Pr_f(v, ch(v)) \cup Pr_2(v, ch(v))$. This implies one of the following must hold, according to our definitions of Pr_1, Pr_2, Pr_f (Definition 3.3.1):

1. $P_{xv} \cap P_{xs} = \{x\}$.
   This means the sensors measuring x are in $T_{x,ch(x)}$. For all sensors s which measure x, $P_{sy} = P_{sx} \cup P_{xy}$, so clearly $|P_{xs}| \neq |P_{ys}|$, and x, y are distinguished.

2. $\exists s_1, s_2 \in S(v, ch(v)), |P_{xs_1}|, |P_{xs_2}| \leq k$ s.t. $P_{xv} \cap P_{xs_1} \neq P_{xv} \cap P_{xs_2}$.
   Then there are two distinct vertices w_1 and w_2 where the paths P_{xv}, P_{xs_1} and P_{xv}, P_{xs_2} split, respectively. One of s_1, s_2 must distinguish x and y. Suppose $|P_{xs_1}| = |P_{ys_1}|$, this is equivalent to $|P_{xw_1}| = |P_{yw_1}|$. We know w_2 is also on the path P_{xy} and there can not be two vertices on a single path with equal distance from its endpoints, so $|P_{xw_2}| \neq |P_{yw_2}|$, which implies s_2 distinguishes x and y.

3. $\forall s_1, s_2 \in S(v, m)$ s.t. $|P_{s_1x}|, |P_{s_2x}| \leq k : P_{xv} \cap P_{xs_1} = P_{xv} \cap P_{xs_2}$ and $d(w(x, v), x) \neq d(w(x, v), v) + 1$ and $\exists s \in S(v, m)$ s.t. $min(k+1, |P_{p(v)s}| + d(x, w(x, v)) - d(w(x, v), p(v))) \neq min(k+1, |P_{xs}|))$.
   This statement means x is measured only through its pivotal vertex $w(x, v)$, $w(x, v)$ does not have equal distance from $p(v)$ as to x and there is a sensor s in $S(v, m)$ which distinguishes x and a vertex at distance $d(w(x, v), x)$ away from $w(x, v)$ outside the subtree $T_{v,m}$. We show y and x are always distinguished. If $|P_{w(x,v)y}| \neq |P_{w(x,v)x}|$, the sensors measuring x distinguish x and y, because we can write the paths from any sensor which measures x, call this sensor s_1 to x and y respectively as

   $$P_{s_1x} = P_{s_1w(x,v)} \cup P_{w(x,v)x} P_{s_1y} = P_{s_1w(x,v)} \cup P_{w(x,v)y}$$

   Clearly s_1 distinguishes x and y by our assumption that $|P_{w(x,v)y}| \neq |P_{w(x,v)x}|$. If $|P_{w(x,v)y}| = |P_{w(x,v)x}|$, by assumption the sensor s distinguishes x and y.

For $x \in NotM(v, ch(v))$ and $y \notin Pr_1(p(v), m-1) \cup Pr_f(p(v), m-1) \cup Pr_2(p(v), m-1)$, the argument that x, y are distinguished is the same. □

Remark 4.10. *This Claim implies* Check-resolving *does not need to iterate over vertex-vertex pairs where one vertex is measured by the sensors in its own subtree but not in any of the problematic items and the other vertex is unmeasured by the sensors in its own subtree.*

We are now ready to prove Proposition 4.1 for merging subtrees.

Claim 4.11. *Suppose the algorithm is processing a vertex v and decides to merge the subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ together into a single subtree $T_{p(v),m}$. Suppose also that the algorithm has allocated $S(v, ch(v))$, $l(v, ch(v))$ and $S(p(v), m-1)$, $(l(p(v), m-1)$ such that Proposition 4.1 holds for each of the subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$ individually. Then the algorithm finds $S(p(v), m)$ and $l(p(v), m)$ such that Proposition 4.1 still holds for the merged subtree $T_{p(v),m}$.*

Proof. We show the *Check-resolving* algorithm only returns True if the sensor set of the merged subtree $S(p(v), m)$ and a sensor outside $T_{p(v),m}$ at distance $l(p(v), m)$ resolve $T_{p(v),m}$. We have shown in Claim 3.11 *Check-resolving* does not need to check vertex-vertex pairs when both vertices are from the same subtree. For the remaining vertex-vertex pairs where the vertices are in distinct subtrees, *Check-resolving* separates four cases based on whether each vertex is measured by the sensors in its own subtree.

(Case 1) **x is measured by its own subtree $S(p(v), m-1)$, y is measured by its own subtree $S(v, ch(v))$.** Through Claim 3.12, x and y must be distinguished.

(Case 2) **x is not measured by its own subtree $S(p(v), m-1)$, y is not measured by its own subtree $S(v, ch(v))$.**
x and y can be distinguished by a sensor from three different sources: $S(p(v), m-1)$, $S(v, ch(v))$ and l', as handled by subcases 2.1, 2.2 and 2.3. If x and y are not distinguished by any sensor, *Check-resolving* returns False.

(Case 3) **x is measured by its own subtree $S(p(v), m-1)$, y is not measured by any sensor in its own subtree $S(v, ch(v))$.**
Through Claim 3.13, we know *Check-resolving* only needs to check those vertex-vertex pairs in Case 3 where $x \in Pr_1(p(v), m-1) \cup Pr_f(p(v), m-1) \cup Pr_2(p(v), m-1)$. Each of these three subcases is handled by one of 3.1, 3.2 or 3.3, by first checking whether x and y are distinguished by the sensors which measure x. If they are not, we check if x and y are distinguished by a sensor corresponding to l'. No other sensors can distinguish x and y, so these two checks determine conclusively if x, y are distinguished. If some vertex-vertex pair fails both checks, *Check-resolving* returns False.

(Case 4) **x is not measured by its own subtree $S(p(v), m-1)$, y is measured by its own subtree $S(v, ch(v))$.**
Case 4 is analogous to Case 3.

We see *Check-resolving* iterates over all vertex-vertex pairs in $T_{p(v),m}$ which might not be distinguished from each other and returns False when some vertex-vertex pair is not distinguished from each other. Therefore, whenever *Check-resolving* returns True we know $S(p(v), m) \cup s^*$, where s^* is a sensor outside $T_{p(v),m}$ at distance $l(p(v), m)$ is a k-resolving set for $T_{p(v),m}$, and the induction is advanced for the new subtree $T_{p(v),m}$. □

4.1.3 Proof of Proposition 4.1

We are now ready to prove Proposition 4.1, which states that for any subtree $T_{v,m}$ for which the algorithm has allocated $S(v, m)$, $l(v, m)$, $S(v, m)$ combined with a sensor outside $T_{v,m}$ at distance $l(v, m)$ from v is a k-resolving set for $T_{v,m}$.

Proof of Proposition 4.1. Combining Claim 4.2, Claim 4.7 and Claim 4.11, it follows Proposition 4.1 must hold by induction. □

4.2 Conjecture of near-optimality

An important detail in Conjecture 2.8 is that we do not conjecture the algorithm always finds the k-truncated metric dimension of a tree. Instead, we conjecture the difference between the k-truncated metric dimension and the size of the k-resolving set the algorithm finds is always smaller or equal to one. The reason for this is that we have found examples where the algorithm is sub-optimal, but we have not been able to find an example where the difference in size between the smallest k-resolving set and the k-resolving set the algorithm finds is greater than one. We present an example where the algorithm produces a sub-optimal output:

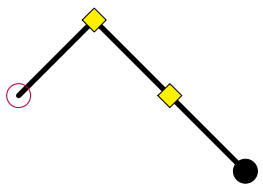


Figure 12: A counterexample against optimality of the algorithm for all $k \geq 3$. Square, yellow vertices are vertices where the algorithm would place sensors. The purple, circle vertex is a k-resolving set smaller than the k-resolving set the algorithm finds.

We have not been able to find an example where the difference between the k-truncated metric dimension and the size of the k-resolving set which the algorithm finds is greater than one. Repeating the structure of Figure 12 does not grow the gap between the size of the smallest k-resolving set and the size of the k-resolving set which the algorithm finds. We do not have an explanation for this phenomenon.

5 Proving the algorithms time complexity

In this section, we prove the time complexity of the algorithm is $\mathcal{O}(k^3 n)$, where k is the measuring range of the sensors and n is the size of the tree used as input for the algorithm. We do this by expressing the number of steps the algorithm takes when processing the i-th vertex as a polynomial function of k with degree three.

Before analyzing the time complexity, we need to prove some bounds on the size of the items the algorithm might iterate over.

Claim 5.1. *For any subtree $T_{v,m}$ for which the algorithm has allocated $NotM(v,m)$ (Definition 5), $|NotM(v,m)| \leq k$.*

Proof. The fact that $NotM(v,m)$ is allocated means $S(v,m)$ and $l(v,m)$ have also been allocated. Through Proposition 4.1, we know $S(v,m) \cup s^*$ forms a k-resolving set for $T_{v,m}$, with s^* a vertex outside $T_{v,m}$ at distance $l(v,m)$ from v. When $l(v,m) = null$, every vertex in $T_{v,m}$ must be measured so $|NotM(v,m)| = 0$. Otherwise, the vertices in $NotM(v,m)$ must be resolved by $S(v,m) \cup s^*$ and they are not measured by $S(v,m)$, so they must be measured by s^*. Since $l(v,m) \geq 1$, vertices in $NotM(v,m)$ can not be further than distance k ($k-1$ to be exact, but we write k for convenience) from the subtrees root v. Moreover, no two vertices in $NotM(v,m)$ can have the same distance from the root v, as this would contradict that s^* resolves these two vertices. Thus, $|NotM(v,m)| \leq k$. □

Claim 5.2. *For any subtree $T_{v,m}$ for which the algorithm has allocated $Pr_1(v,m)$ (Definition 3.3.1), $|Pr_1(v,m)| \leq k$.*

Proof. This proof is similar to the proof of Claim 5.1. The fact that $Pr_1(v,m)$ is allocated means $S(v,m)$ and $l(v,m)$ have also been allocated. By definition, vertices in $Pr_1(v,m)$ are measured through the root of the subtree v (Definition 3.3.1), so vertices in $Pr_1(v,m)$ can not be further than distance k from v. Two vertices in $Pr_1(v,m)$ can not have the same distance from v, as this would contradict Proposition 4.1, which states $S(v,m) \cup s^*$, where s^* is a sensor outside $T_{v,m}$ at distance $l(v,m)$ forms a k-resolving set for $T_{v,m}$. □

Claim 5.3. *For any subtree $T_{v,m}$ for which the algorithm has allocated $Pr_f(v,m)$ (Definition 3.3.1), $|Pr_f(v,m)| \leq k$.*

Proof. The fact that $Pr_f(v,m)$ is allocated means $S(v,m)$ and $l(v,m)$ have also been allocated. Let x and y be two distinct vertices in $Pr_f(v,m)$. We take p^x and p^y to be the vertex outside $T_{v,m}$ at distance $d(x, w(x,v))$ from $w(x,v)$ and $d(y, w(y,v))$ from $w(y,v)$ respectively. As a first step of the proof, we show there can not be two distinct vertices x, y in $Pr_f(v, ch(v))$ such that $p^x = p^y$. We will prove this by contradiction. Suppose x, y are such that $p^x = p^y$. We will derive a contradiction by distinguishing two cases.

1. If $w(x,v) = w(y,v)$, for lack of confusion we will refer to $w(x,v) = w(y,v)$ as simply w. Notice that our assumption that $p^x = p^y$ implies $d(x,w) = d(y,w)$. We will derive a contradiction by showing x and y violate Proposition 4.1, which states x, y are distinguished by $S(v,m) \cup s^*$, where s^* is a vertex outside $T_{v,m}$ at distance $l(v,m)$ from v. For all sensors which measure x and y, w is on the shortest path from x,y to that sensor (per definition of Pr_f, Definition 3.3.1), respectively. Therefore the sensors in $S(v,m)$ can not distinguish x and y. By definition of the pivotal vertex, w is on the paths P_{xv} and P_{yv} (Definition 3.6), so x, y can also not be distinguished by a sensor outside $T_{v,m}$ at distance $l(v,m)$ from v. Therefore, x, y violate Proposition 4.1, a contradiction as desired.

2. If $w(x,v) \neq w(y,v)$, we derive a contradiction by showing y violates the conditions for being in $Pr_f(v,m)$. The condition y violates is the condition that no sensor in $S(v,m)$ distinguishes p^y and y. We will show a sensor which measures x distinguishes p^y and y. Remember that for any sensor

s_x which measures x, $w(x,v)$ is on the path from that sensor to x. We know this sensor measures p^y through the following

$$d(s_x, p^y) = d(s_x, p^x) = d(s_x, w(x,v)) + d(w(x,v), x) \leq k$$

The first equality holds by the assumption that $p^x = p^y$. Moreover, due to the assumption that $w(x,v) \neq w(y,v)$, and per the definition of problematic vertices $w(y,v)$ is the unique vertex with the property that $d(w(y,v), y) = d(w(y,v), p^y)$ (Definition 3.3.1), so s_x must distinguish p^y and y, a contradiction as desired.

Now we have shown no two vertices in $x, y \in Pr_f(v,m)$ can have the same $p^x = p^y$.

In the second step of the proof, we show that for any $x \in Pr_f(v,m)$, $d(v, p^x) \leq k$ must hold. By how we defined p^x at the start of this proof, we know $d(s_x, p^x) = d(s_x, x) \leq k$. We also know that

$$d(s_x, p^x) = d(s_x, w(x,v)) + d(w(x,v), v) + d(v, p^x)$$

Therefore $d(v, p^x) \leq k$.

To finalise the proof, we combine the fact that for any $x \in Pr_f(v,m)$, $d(v, p^x) \leq k$ and that there is no other vertex y in $Pr_f(v,m)$ with distance from v to p^y equal to $d(v, p^x)$ in order to show that $|Pr_f(v,m)| \leq k$. □

Next, we examine the number of steps the algorithm takes when adding the parent vertex and when merging subtrees, independently.

5.1 Time complexity of adding the parent

Claim 5.4. *Suppose the algorithm is processing a vertex v in $T_{v,ch(v)}$ and adds the parent of v, $p(v)$, to the subtree. The number of computational steps the algorithm performs is upper-bounded by a function of k.*

Proof. We separate cases based on the value of $l(v, ch(v))$.

(Case 1) When $l(v, ch(v)) = 1$, the algorithm takes a fixed (not a function of k or n) number of steps before updating all items corresponding to $T_{v,ch(v)}$. Looking at Section 3.4.1, we see updating all items does not require iterating over anything, so updating items is constant time.

(Case 2) When $2 \leq l(v, ch(v)) \leq k$, the algorithm takes a fixed (not a function of k or n) number of steps before updating all items corresponding to $T_{v,ch(v)}$. Looking at Section 3.4.1, we see that updating all items requires iterating over $NotM(v, ch(v))$, $Pr_1(v, ch(v))$ and $Pr_f(v, ch(v))$.

(Case 3) When $l(v, ch(v)) = null$, the algorithm iterates over $Pr_1(v, ch(v))$ in order to determine the value for $l(p(v), 1)$. After this, it updates all items corresponding to $T_{v,ch(v)}$. Looking at Section 3.4.1, we see updating all items requires iterating over $NotM(v, ch(v))$, $Pr_1(v, ch(v))$, $Pr_f(v, ch(v))$.

Worst-case, the algorithm sees that $l(v, ch(v)) = null$ and it proceeds to iterates once over $NotM(v, ch(v))$, $Pr_f(v, ch(v))$ and twice over $Pr_1(v, ch(v))$. Using Claims 5.1, 5.2 and 5.3, we see the number of computational steps of adding the parent vertex to the subtree is smaller or equal to $4k$. □

5.2 Time complexity of merging subtrees

When the algorithm is processing a vertex v and decides to merge the subtrees $T_{v,ch(v)}$ and $T_{p(v),m-1}$, it performs the *Check-cases* and *Check-resolving* sub-algorithms until *Check-resolving* returns True. Therefore, before upper-bounding the number of computational steps of the merging subtrees operation, we require an upper-bound on these two sub-algorithms individually. Looking at Section 3.7.3, it is clear the *Check-cases* algorithm does not iterate over any item, so the number of computational steps it takes is upper-bounded by a constant function. We turn our attention to upper-bounding *Check-resolving*.

Claim 5.5. *The number of computational steps of the* Check-resolving *sub-algorithm is upper-bounded by a function of k.*

Proof. Looking at Section 3.7.5, we see the *Check-resolving* algorithm iterates over five types of vertex-vertex pairs:

1. $x \in NotM(p(v), m-1)$, $y \in NotM(v, ch(v))$
2. $x \in NotM(p(v), m-1)$, $y \in Pr_1(v, ch(v))$
3. $x \in NotM(p(v), m-1)$, $y \in Pr_f(v, ch(v))$
4. $y \in NotM(v, ch(v))$, $x \in Pr_1(p(v), m-1)$
5. $y \in NotM(v, ch(v))$, $x \in Pr_f(p(v), m-1)$

Using Claims 5.1, 5.2 and 5.3, we see the number of computational steps of the *Check-resolving* algorithm is upper-bounded by $5k^2$ because the size of each item listed above is upper-bounded by k. □

Now that we have bounded the number of computational steps of *Check-cases* and *Check-resolving*, we can prove an upper-bound on the number of computational steps it takes to merge two subtrees.

Claim 5.6. *Suppose the algorithm is processing a vertex v and decides to merge the subtrees $T_{p(v),m-1}$ and $T_{v,ch(v)}$. The number of computational steps the algorithm performs to merge the subtrees is upper-bounded by a function of k.*

Proof. Computationally, the worst-case scenario is *Check-cases* determines Case 1 or Case 5.1 hold, *Check-resolving* returns False for all $k+1$ possible values for l' and the algorithm places a sensor on v. It needs to update all items corresponding to $T_{v,ch(v)}$ as well, which, looking at 3.4.1, we see this can be done in constant time. After placing a sensor on v, worst-case the algorithm finds Case 6.1 holds and it proceeds to run *Check-resolving* k times, until finally *Check-resolving* returns true for $l' = 1$.

In this scenario, *Check-resolving* must return True after placing a sensor on v and trying $l' = 1$. The reasoning is as follows: There is a sensor on v, so $l(v, ch(v)) = null$ and $T_{v,ch(v)}$ is resolved by $S(v, ch(v))$. Then every vertex in $T_{v,ch(v)}$ is naturally measured by sensors in its own subtree, so the only vertex-vertex pairs *Check-resolving* still needs to check are $x \in NotM(p(v), m-1)$ and $y \in Pr_1(v, ch(v)) \cup Pr_f(v, ch(v)) \cup Pr_2(v, ch(v))$. We know from the definition of problematic vertices that all sensors measuring y measure y through its pivotal vertex $w(y, v)$ (Defintion 3.3.1)(Definition 3.6). Vertices x and y are not distinguished by the sensors in the merged subtree $S(p(v), m)$ if their distance from the pivotal vertex is equal, so if

$$d(x, w(y, v)) = d(y, w(y, v))$$

Supposing this holds, a sensor outside $T_{p(v),m}$ at distance 1 from $p(v)$ (denote this sensor s^*) would distinguish x and y, because

$$\begin{aligned} d(x, s^*) &= d(x, p(v)) + 1 \\ d(y, s^*) &= d(y, v) + 1 + 1 = d(y, w(y, v)) + d(w(y, v), v) + 2 \\ d(y, w(y, v)) &= d(x, w(y, v)) = d(x, p(v)) + d(p(v), w(y, v)) \\ &\implies d(y, s^*) = d(x, p(v)) + d(p(v), w(y, v)) + d(w(y, v), v) + 2 \\ &\implies d(x, s^*) < d(y, s^*) \end{aligned}$$

Therefore after placing a sensor on v, in this worst-case scenario *Check-resolving* returns True after trying k values for l'. After finding *Check-resolving* is True, the algorithm updates all items corresponding to the merged subtree $T_{p(v),m}$. Looking at 3.4.2, we see this requires iterating over $NotM(v, ch(v))$, $NotM(p(v), m-1)$, $Pr_1(p(v), m-1)$, $Pr_1(v, ch(v))$, $Pr_f(p(v), m-1)$ and $Pr_f(p(v), m-1)$. Using Claims 5.1, 5.2 and 5.3, the number of computational steps for updating all items is upper-bounded by

$$6k$$

Combining everything mentioned, we see in the worst-case scenario the algorithm runs *Check-resolving* $2k+1$ times. We know the number of computational steps for *Check-resolving* is upper-bounded by $5k^2$ (Claim 5.5), therefore the number of computational steps of merging subtrees is upper-bounded by

$$(2k+1)(5k^2) + 6k = 10k^3 + 5k^2 + 6k$$

□

Now that we have proven the time complexity of merging subtrees, we have all the tools we need to prove the time complexity of the algorithm in its entirety.

5.3 Proof of the time complexity of the algorithm

We are ready to prove the algorithm is $\mathcal{O}(k^3n)$.

Proposition 5.7. *The time complexity of the algorithm explained in this thesis is $\mathcal{O}(k^3n)$.*

Proof. We will prove the time complexity by showing the maximum number of computational steps possible when processing a vertex v is a function of k. Using Claim 5.4 and Claim 5.6, we see the maximum number of computational steps when processing a vertex v is

$$10k^3 + 5k^2 + 6k$$

Therefore, the number of computational steps for running the algorithm on a tree of size n is upper-bounded by

$$\sum_{i=1}^{n} 10k^3 + 5k^2 + 6k = (10k^3 + 5k^2 + 6k)n$$

Therefore we conclude the algorithm is $\mathcal{O}(k^3n)$. □

6 Conclusion

Source-detection is a large, active field of research with a great catalogue of problems. Finding the k-truncated metric dimension of trees is one of the open problems in this field [26]. In this thesis, we took the algorithm for k-truncated metric dimension of trees which was given by Gutkovich and Song Yeoh [6], modified their algorithm to improve it in both validity and time complexity, and proved the output of the algorithm as given in this thesis is a k-resolving set. Moreover, we proved the time complexity of the algorithm is $\mathcal{O}(k^3n)$.

7 Discussion

Further research could attempt to prove Conjecture 2.8 by proving the difference between the truncated metric dimension and the size of the output of the algorithm given in this thesis is never greater than one. After proving near-optimality of the algorithm, one could study the truncated metric dimension of other families of graphs. There is much work to be done concerning the truncated metric dimension of graphs, as to our knowledge, there is no known method of finding the truncated metric dimension of any (non-trivial) family of graphs.

A method to find the truncated metric dimension of a family of graphs could be a powerful tool to study the metric dimension of those graphs. A result by Tillquist et al. suggests that for any graph, the k-truncated metric dimension approaches the metric dimension as the measuring range of the sensors increases [26]. Researching a method to find the k-truncated metric dimension of general graphs could therefore lead to a breakthrough in the problem of finding the metric dimension of general graphs, which is NP-Hard [13]. As a first step, one could modify the algorithm for the truncated metric dimension of trees in order to use it as a heuristic for finding the truncated metric dimension and the metric dimension of graphs which could be considered 'locally tree-like'.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 253–265, 1973.

[2] Z. Bartha, J. Komjáthy, and J. Raes. Sharp bound on the threshold metric dimension of trees. 11 2021.

[3] Z. Chen, K. Zhu, and L. Ying. Detecting multiple information sources in networks under the sir model. *IEEE Transactions on Network Science and Engineering*, 3(1):17–31, 2016.

[4] A. Estrada-Moreno, J. A. Rodríguez-Velázquez, and I. G. Yero. The k-metric dimension of a graph, 2015.

[5] V. Fioriti, M. Chinnici, and J. Palomo. Predicting the sources of an outbreak with a spectral technique. *Applied Mathematical Sciences,HIKARI Ltd*, 8:6775–6782, 01 2014.

[6] P. Gutkovich and Z. Song Yeoh. Computing truncated metric dimension on trees. 2022.

[7] C. Hernando, M. Mora, P. J. Slater, and D. R. Wood. Fault-tolerant metric dimension of graphs. *Convexity in discrete structures*, 5:81–85, 2008.

[8] M. Imran, A. Q. Baig, S. A. U. H. Bokhary, and I. Javaid. On the metric dimension of circulant graphs. *Applied mathematics letters*, 25(3):320–325, 2012.

[9] I. Javaid, M. T. Rahim, and K. Ali. Families of regular graphs with constant metric dimension. *Utilitas mathematica*, 75(1):21–33, 2008.

[10] N. Karamchandani and M. Franceschetti. Rumor source detection under probabilistic sampling. In *2013 IEEE International Symposium on Information Theory*, pages 2184–2188, 2013.

[11] M. Karpovsky, K. Chakrabarty, and L. Levitin. On a new class of codes for identifying vertices in graphs. *IEEE Transactions on Information Theory*, 44, 09 2001.

[12] A. Kelenc, D. Kuziak, A. Taranenko, and I. G. Yero. Mixed metric dimension of graphs. *Applied Mathematics and Computation*, 314:429–438, 2017.

[13] S. Khuller, B. Raghavachari, and A. Rosenfeld. Landmarks in graphs. *Discrete applied mathematics*, 70(3):217–229, 1996.

[14] D. Kuziak and I. G. Yero. Metric dimension related parameters in graphs: A survey on combinatorial, computational and applied results, 2021.

[15] J.-B. Liu, M. F. Nadeem, H. M. A. Siddiqui, and W. Nazir. Computing metric dimension of certain families of toeplitz graphs. *IEEE Access*, 7:126734–126741, 2019.

[16] W. Luo, W. P. Tay, and M. Leng. Identifying infection sources and regions in large networks. *IEEE Transactions on Signal Processing*, 61, 04 2012.

[17] D. Nguyen, N. Nguyen, and M. Thai. Sources of misinformation in online social networks: Who to suspect? pages 1–6, 10 2012.

[18] B. Prakash, J. Vreeken, and C. Faloutsos. Efficiently spotting the starting points of an epidemic in a large graph. *Knowledge and Information Systems*, 38, 01 2014.

[19] H. Sha, M. Al Hasan, and G. Mohler. Source detection on networks using spatial temporal graph convolutional networks. In *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–11, 2021.

[20] D. Shah and T. Zaman. Rumor centrality: A universal source detector. *SIGMETRICS Perform. Eval. Rev.*, 40(1):199–210, jun 2012.

[21] M. Shahsavandi, J. Yazdi, M. Jalili Ghazizadeh, and A. Rashidi Mehrabadi. The use of graph theory for search space reduction in contaminant source identification. *Journal of Pipeline Systems Engineering and Practice*, 14(2):04023016, 2023.

[22] S. Shelke and V. Attar. Source detection of rumor in social network–a review. *Online Social Networks and Media*, 9:30–42, 2019.

[23] P. J. Slater. Leaves of trees. *Congr. Numer*, 14(549-559):37, 1975.

[24] P. J. Slater. Domination and location in acyclic graphs. *Networks*, 1987.

[25] C. W. Tan, P.-D. Yu, et al. Contagion source detection in epidemic and infodemic outbreaks: Mathematical analysis and network algorithms. *Foundations and Trends® in Networking*, 13(2-3):107–251, 2023.

[26] R. C. Tillquist, R. M. Frongillo, and M. E. Lladser. Truncated metric dimension for finite graphs, 2021.

[27] Y. Zhou, C. Wu, Q. Zhu, Y. Xiang, and S. W. Loke. Rumor source detection in networks based on the seir model. *IEEE Access*, 7:45240–45258, 2019.

[28] K. Zhu and L. Ying. Information source detection in the sir model: A sample-path-based approach. *IEEE/ACM Transactions on Networking*, 24(1):408–421, 2014.