Non-deterministic graph searching in trees

Omid Amini, David Coudert, Nicolas Nisse

Non-deterministic Graph Searching in Trees*

Omid Amini[1], David Coudert[2,3], and Nicolas Nisse[2,3]

[1]CNRS - DMA, École Normale Supérieure, Paris, France
[2]Inria, France
[3]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

Abstract

Non-deterministic graph searching was introduced by Fomin *et al.* to provide a unified approach for pathwidth, treewidth, and their interpretations in terms of graph searching games. Given $q \geq 0$, the q-limited search number, $s_q(G)$, of a graph G is the smallest number of searchers required to capture an invisible fugitive in G, when the searchers are allowed to know the position of the fugitive at most q times. The search parameter $s_0(G)$ corresponds to the pathwidth of a graph G, and $s_\infty(G)$ to its treewidth. Determining $s_q(G)$ is NP-complete for any fixed $q \geq 0$ in general graphs and $s_0(T)$ can be computed in linear time in trees, however the complexity of the problem on trees has been unknown for any $q > 0$.

We introduce a new variant of graph searching called restricted non-deterministic. The corresponding parameter is denoted by rs_q and is shown to be equal to the non-deterministic graph searching parameter s_q for $q = 0, 1$, and at most twice s_q for any $q \geq 2$ (for any graph G).

Our main result is a polynomial time algorithm that computes $rs_q(T)$ for any tree T and any $q \geq 0$. This provides a 2-approximation of $s_q(T)$ for any tree T, and shows that the decision problem associated to s_1 is polynomial in the class of trees. Our proofs are based on a new decomposition technique for trees which might be of independent interest.

Keywords: Graph Searching; Treewidth; Pathwidth; Trees.

1 Introduction

Graph searching problems have been extensively studied for practical aspects such as pursuit-evasion problems [16], but also for their close relationship with fundamental structural parameters of graphs, namely pathwidth and treewidth, that serve as important tools in Robertson and Seymour's Graph Minor Theory [17]. In particular, many intractable problems can be solved in linear time when the input is restricted to graphs of bounded treewidth [5]. In this paper, $tw(G)$ and $pw(G)$ denote the treewidth and the pathwidth of a graph G, respectively.

Graph searching is a game in which a team of searchers is aiming at capturing a fugitive hidden in a graph. The searchers can be placed on or removed from the vertices of the graph. The fugitive stands at some vertex of the graph and can move arbitrary fast from its current vertex to another by following the paths in the graph as long as it does not cross any vertex occupied by a searcher. The fugitive has perfect knowledge about the position and future moves of searchers. The fugitive is caught when it occupies the same vertex as a searcher and has

*This project has been partially supported by GDR ASR ResCom, by ANR project Stint under reference ANR-13-BS02-0007 and by ANR program "Investments for the Future" under reference ANR-11-LABX-0031-01.

no way to escape. A vertex is *contaminated* if it may harbor the fugitive, and is *cleared* by placing a searcher on it. Once cleared, a vertex remains clear as long as every path from it to a contaminated vertex is guarded by at least one searcher. Otherwise, the vertex is *recontaminated*. The graph is clear as soon as all the vertices are simultaneously clear. Therefore, the fugitive is caught. A *node (search) strategy* is a sequence of searchers moves (place or remove), or *steps*, that guarantees the fugitive's capture. A strategy is *monotone* if no vertex is visited more than once by a searcher, i.e., if *recontamination* never occurs.

Two main variants of graph searching have been particularly studied: either the fugitive is *invisible*, meaning that the searchers do not know its position unless it is caught, or it is *visible*, i.e., at any step of the strategy, the searchers know the current position of the fugitive and they can thus adapt their strategy according to this knowledge. The *node search number* $s(G)$ (resp., the *visible search number* $vs(G)$) of a graph G is the minimum number of searchers for which a strategy capturing an invisible (resp., visible) fugitive exists for G [3, 18]. One important result of the field is that *recontamination does not help*. That is, for any graph G, there is a monotone strategy using the optimal number of searchers to capture an invisible (resp., visible) fugitive in G [3, 18]. In particular, it follows that the node search number and the visible search number of a graph are closely related to its pathwidth and treewidth, namely, for any graph G, $s(G) = pw(G) + 1$ and $vs(G) = tw(G) + 1$ (see [12] for a survey on graph searching).

In [11], Fomin *et al.* introduced a parametric variant called *non-deterministic graph searching*, and proved that the corresponding parameter establishes a link between invisible and visible search numbers, i.e., between pathwidth and treewidth. They proved that computing this parameter is NP-hard in general and asked whether it can be computed in polynomial time when the input is restricted to be a tree. In this paper, we study this latter problem.

In non-deterministic graph searching, the fugitive is invisible but the searchers have the possibility to query an oracle that knows the current position of the fugitive (a limited number of times). That is, given the set W of clear vertices, *performing a query* returns a connected component C of $G \setminus W$. The vertices of C remain contaminated and those of $G \setminus C$ become clear. Obviously, the number of searchers required to catch the fugitive cannot increase when the number of permitted performing-a-query steps increases.

A *non-deterministic (search) strategy* is a sequence of the three basic operations:

- Placing a searcher on a vertex,
- Removing a searcher from a vertex, and
- Performing a query.

Note that such a strategy corresponds to a decision tree so that the performing-a-query steps correspond to the forks in the decision-tree. A possible execution of this strategy is a sequence of such operations following a path of the decision-tree from its root to a leaf, corresponding to some choice for any query step, i.e., depending on the behavior of the fugitive. The strategy must result in catching the fugitive whatever it does. The number of query-steps, denoted by $q \geq 0$, is however fixed. The *q-limited search number* of a graph G, $s_q(G)$, is the smallest number of searchers required to catch a fugitive performing at most q query-steps. Mazoit and Nisse [14] generalized the monotonicity results of [3] and [18]. They proved that *recontamination does not help* neither in non-deterministic case: for any $q \geq 0$ and any graph G, there is a monotone strategy performing at most q queries that uses at most $s_q(G)$ searchers [14]. Hence, throughout this paper, we consider only monotone strategies. We moreover assume that useless moves such as placing a searcher on a clear or occupied node never occur.

The monotonicity result is also important because monotone non-deterministic graph searching realizes a link between treewidth and pathwidth through the notion of *q-branched tree decompositions* [11]. The definition of q-branched treewidth and its relationship with the q-limited search number are as follows.

Given a rooted tree $\mathfrak{T}$, with root $\mathfrak{r}$, a *branching node* of $\mathfrak{T}$ is a node with at least two children. Let $q \geq 0$. A *q-branched tree* $\mathfrak{T}$ is a rooted tree such that every path in $\mathfrak{T}$ from (root) $\mathfrak{r}$ to a leaf contains at most q branching nodes.

Let $G = (V, E)$ be a connected graph and let $q \geq 0$. A *q-branched tree decomposition* [11] of a graph G is a pair $(\mathfrak{T}, \mathcal{X})$ where $\mathfrak{T}$ is a q-branched tree on a set of nodes $\mathfrak{I}$, and $\mathcal{X} = \{X_i : i \in \mathfrak{I}\}$ is a collection of subsets of V, subject to the following three conditions:

1. $V = \cup_{i \in \mathfrak{I}} X_i$,
2. for any edge e in G, there is a set $X_i \in \mathcal{X}$ which contains both end-points of e,
3. for any triple i_1, i_2, i_3 of nodes of $\mathfrak{T}$, if i_2 is on the path from i_1 to i_3 in $\mathfrak{T}$, then $X_{i_1} \cap X_{i_3} \subseteq X_{i_2}$.

The *width* of $(\mathfrak{T}, \mathcal{X})$ is defined as $w(\mathfrak{T}, \mathcal{X}) = \max_{i \in \mathfrak{I}} |X_i| - 1$. The *$q$-branched treewidth* of a graph G, denoted by $tw_q(G)$, is the minimum width of any q-branched tree decomposition of G. Note that $tw_{q'}(G) \leq tw_q(G)$ for any $q \leq q'$. Obviously, for q large enough, $tw_q(G) = tw(G)$, where $tw(G)$ denotes the treewidth of G. In other word, $tw(G) = \min_{q \geq 0} tw_q(G) =: tw_\infty(G)$. Moreover, $tw_0(G) = pw(G)$, where $pw(G)$ denotes the pathwidth of G. In this way, the family of parameters $tw_q(G)$ can be regarded as an interpolating family of parameters between the pathwidth and the treewidth a graph G. The main theorem of [11] and the monotonicity result of [14] establish the link between q-limited search number and q-branched treewidth.

Theorem 1 ([11, 14])**.** *Let $q \geq 0$, and G a graph, $tw_q(G) = s_q(G) - 1$.*

1.1 Overview of the results of this paper

We first introduce a new variant of graph searching that we call *restricted non-deterministic graph searching*. The corresponding search parameter rs_q, parametrized by $q \in \mathbb{N} \cup \{0\}$, allows to go from pathwidth to treewidth as q goes from 0 to infinity. We prove (in Section 2) that restricted non-deterministic graph searching provides a 2-approximation for non-deterministic graph searching in any graph.

We study the problem of non-deterministic graph searching for trees. Our algorithms are dedicated to exact computation of restricted non-deterministic graph searching in trees. The main result of this paper is an algorithm which computes in time polynomial in n (independent of $q \geq 0$) the restricted q-limited search number rs_q of any tree on n vertices. This yields a 2-approximation polynomial time algorithm for the q-limited search number s_q of any tree, which turns out to be exact for $q \in \{0, 1\}$.

Let us now describe the main ingredients of our algorithms.

As this is the case for results of the same type concerning trees, our algorithm proceeds by labeling the vertices of the tree using dynamic programming. However, several difficulties arise, and we need to proceed in several steps. First, as a cornerstone of all our results, we need to consider the problem of graph searching where in addition the initial positions of the searchers are imposed. We generalize the algorithm of Skodinis [19] to design in Section 3.1 Protocol `InitPos` that computes in polynomial time the (invisible) search number of any tree when the initial positions of the searchers are imposed.

The second cornerstone of our main algorithm is the algorithm `TwoSearchers` that determines in polynomial-time the minimum number of queries needed to clear a tree using two searchers.

In Section 4, Algorithms `InitPos` and `TwoSearchers` are used (as black box) in the design of Algorithm `OneQuery` that computes $rs_1(T)$ for any tree T in polynomial-time. By Theorem 3, this shows that the problem of computing s_1 in the class of trees belongs to the complexity class P.

In technical Section 5, which forms the heart of the paper, we will generalize the ideas presented in Section 4 to obtain a polynomial time algorithm, called `Approx`, for determining rs_q, thus yielding a polynomial 2-approximation algorithm for computing $s_q(T)$ in any tree T. Note that the definition of Algorithm `Approx` is recursive (in q) and that algorithm `OneQuery` serves as the basis of the recursion (see Lemma 1). Beside the technicalities involved in the proof of our main result, as indicated above, the main new idea here is the notion of *k-good decomposition* of a labeled tree (see Section 5). In Proposition 3, we show that, if a labeled tree T admits such a decomposition, then $rs_q(T) \leq k$. The main technical difficulty is to show that Algorithm `Approx` actually computes a labeling compatible with a k-good decomposition in any tree T such that $rs_q(T) \leq k$.

It turns out that passing from two to three searchers drastically changes the behavior of the searching strategies. Sections 6 and 7 are devoted to the study of the case of two searchers. First, we present in Section 6 Algorithm `TwoSearchers`, which was used in the previous sections as a black box. Postponing the design of this algorithm has been done in order to increase the readability of the paper since Algorithm `TwoSearchers` is different from previous ones (in particular, it does not use the notion of good decomposition). Section 7, which is independent of the rest of the paper, shows that clearing an n-node tree using two searchers may require $\Omega(n)$ queries while using three searchers always requires $O(\log n)$ queries. Hence, there is an exponential gap on the number of queries required to clear a tree when the number of searchers passes from two to three.

1.2 Related work

Many versions of graph searching problems have been considered by allowing variations of the different parameters. For instance, the fugitive may be arbitrarily fast or its speed may be limited (e.g., cops and robber games). In each version, either the fugitive is invisible or it may be visible. Many other parameters can enter to the picture (e.g., the connectivity of the clear part [1], etc.). In general, these parameters reflect the relationship between the considered graph searching problem and the structural properties of the underlying graph.

Determining the pathwidth of a graph is NP-complete [15], even for the class of star-like graphs (graphs whose vertex-set can be partitioned into a clique and an independent set) [13]. However, it can be computed in linear time for bounded treewidth graphs [5]. Trees have been particularly studied for classical searching problems [6, 16, 15, 10]. Skodinis [19] obtained a linear time algorithm with small constant factor, that computes an optimal path decomposition of any tree. Barrière *et al.* [2] gave a distributed algorithm for computing the connected search number of trees in linear time. Coudert *et al.* [7] proposed a distributed algorithm for computing and updating the node, edge and process numbers of trees after any tree-edge addition or deletion. Ellis and Markov [9] gave a linear time algorithm for the class of unicyclic graphs. Bodlaender and Fomin [4] and Coudert *et al.* [8] use the weak dual of an outerplanar graph, that is a tree, to approximate its pathwidth.

Similarly, determining the q-limited search number of a graph is NP-complete in general [11]. However, the design of a polynomial time algorithm for computing the q-limited search number in bounded treewidth graphs, and even for trees, for any fixed q, is still an open problem. For fixed $q \geq 0$, Fomin *et al.* [11] proposed an exact exponential time algorithm, in time $O(2^n n \log n)$, that computes $s_q(G)$ and the corresponding non-deterministic strategy in any graph G on n vertices.

Note that for fixed $k \geq 1$, the decision version of the algorithm answers in time $O(n^{k+1})$ whether $s_q(G) \leq k$.

2 Restricted non-deterministic graph searching

We introduce in this section restricted non-deterministic graph searching. We prove that the corresponding parameter, denoted by rs_q, provides a 2-approximation of s_q. The main part of the paper will be then devoted to the design of a polynomial-time algorithm to compute rs_q in trees.

A *restricted (non-deterministic search) strategy* is a monotone non-deterministic search strategy such that the moves are ordered in the following particular way. Initially, the searchers are placed on some vertices, and the first query is performed. As long as there is still the possibility of performing a query, the strategy consists in first removing the searchers that occupy a vertex all the neighbors of which are clear, and then placing some (possibly all) of the free searchers on some vertices in the contaminated part, and then performing a query. When there is no query left, the strategy proceeds as usual. In other words, in this variant of graph searching, once a searcher is placed on some contaminated vertex, it cannot be removed as long as the next query has not been performed (unless no query remains).

The *restricted q-limited search number* of a graph G, denoted $rs_q(G)$, is the smallest number of searchers required to catch a fugitive performing at most q query steps, in a restricted non-deterministic way.

A *restricted q-branched tree* $\mathfrak{T}$ is a q-branched tree with the following property: for any $v \in V(\mathfrak{T})$ that belongs to a path between two vertices of degree at least three, either v is the root and has degree at least two, or v has degree at least three. That is, for any vertex v which has a unique child, the subrooted tree $\mathfrak{T}_v$ of $\mathfrak{T}$, rooted at v, is a path with endpoint v. A *restricted q-branched tree decomposition* of a graph G is a tree decomposition $(\mathfrak{T}, \mathcal{X})$ where $\mathfrak{T}$ is a restricted q-branched tree. The *restricted q-branched treewidth*, $rtw_q(G)$, of a graph G, is the minimum width of any restricted q-branched tree decomposition of G.

Theorem 2. *For any $q \geq 0$ and for any graph G, $rtw_q(G) = rs_q(G) - 1$.*

Proof. The proof is similar to the proof of Theorem 1 in [11]. We first show that $rtw_q(G) \geq rs_q(G) - 1$. Let $(\mathfrak{T}, \mathcal{X})$ be a restricted q-branched tree decomposition of G, with width rtw_q. Let r be the root of $\mathfrak{T}$. The strategy is the following: place a searcher on any vertex of the root-bag $X_r \in \mathcal{X}$ and perform the first query (if $q > 0$). Let H be the component of $G \setminus X_r$ in which the fugitive is revealed. There is a child $r' \in V(\mathfrak{T})$ of r such that $(\mathfrak{T}_r, \mathcal{X}')$ is a restricted $(q-1)$-branched tree decomposition of H (where $\mathcal{X}'$ is the restriction of $\mathcal{X}$ to the nodes of $\mathfrak{T}_r$). The strategy goes on by removing the searchers in $X_r \setminus X_{r'}$ and then placing searchers on all unoccupied vertices of $X_{r'}$. Then, if a query is still available, it is performed. And so on. When no queries are left, the searchers are at the vertices of X_t for some $t \in V(\mathfrak{T})$ such that $\mathfrak{T}_t$ is a path with t as an end. Therefore, the searchers are at the vertices of the first bag of a path decomposition of width $\leq rtw_q(G)$ of the remaining contaminated component. Hence, they can clear the remaining part of the graph with no more queries. Such a strategy is clearly restricted, uses at most q queries and $rtw_q(G) + 1$ searchers.

To prove the other inequality, let us consider a restricted strategy of G using $q \geq 0$ queries and k searchers. By definition, it starts by placing the searchers on the vertices of $X \subseteq V(G)$ and performs the first query. We prove by induction on $q \geq 0$ that there is a restricted q-branched tree decomposition of G with width $\leq rs_q(G) - 1$ and with root-bag X. If $q = 0$, the result holds because $rs_0(G) = s_0(G) = pw(G) = rtw_0(G)$. If $q > 0$, let $C_1, \cdots, C_i$ be the connected components of $G \setminus X$ (Note that we may assume that $i \geq 2$, because otherwise the

strategy may be modified). For any $j \leq i$, if the contaminated component after the first query is C_j, the searchers at vertices not adjacent to some vertex in C_j are removed and then some searchers are placed. Let X_j be the set of vertices occupied just before the second query (or before the first removal step of a vertex in X that is adjacent to a vertex of C_j). Applying the induction hypothesis for $C_j \cup X_j$ starting from X_j, we obtain a restricted $(q-1)$-branched tree decomposition of $C_j \cup X_j$ with width $\leq rs_q(G)-1$. Combining the tree decompositions obtained for each $j \leq i$ by making each bag X_j adjacent to the root-bag X allows to obtain the desired decomposition. □

The importance of these new parameters is given by the next theorem which provides a link between restricted and non-restricted q-branched treewidths.

Theorem 3. *For any $q \geq 2$ and for any graph G, $rtw_q(G) \leq 2\ tw_q(G) + 1$.*
For $q \in \{0, 1\}$ and for any graph G, $rtw_q(G) = tw_q(G)$.

Proof. Let $(\mathfrak{T}, \mathcal{X})$ be a q-branched tree decomposition of G of width $tw_q(G)$, and let r be the root of $\mathfrak{T}$. Let w be the root or a vertex of degree at least three in $\mathfrak{T}$. Let v be a descendant of w of degree at least three such that the unique path $\{w, u_1, \cdots, u_l, v\}$ between w and v has internal nodes of degree two, i.e., all the nodes $u_1, \ldots, u_l$ have degree two ($l \geq 1$). Modify $(\mathfrak{T}, \mathcal{X})$ in the following way. First remove the edge $\{w, u_1\}$ from $\mathfrak{T}$ and add an edge $\{w, v\}$. (The obtained tree is still rooted at r.) Then replace X_v by $X_v \cup X_w$; and for any u_i, replace X_{u_i} by $X_{u_i} \cup X_w$. Obviously, this results in a q-branched tree decomposition of G. By repeating this process, one obtains a restricted q-branched tree decomposition of width at most $2\ tw_q(G) + 1$. Indeed, for any $v \in V(\mathfrak{T})$, X_v is modified at most once.

For the other statement, note that in the case $q = 0$, the result is obvious, and for $q = 1$, the result follows by observing that there exists always a monotone non-deterministic search strategy using at most q queries and $s_q(G)$ searchers in which the first query step happens before any removing step, see Proposition 1 below. □

Note that the translation of the above theorem for the corresponding search parameters give the inequalities $rs_q(G) \leq 2\ s_q(G)$ for any $q \geq 2$, and the equalities $rs_0(G) = s_0(G)$ and $rs_1(G) = s_1(G)$.

3 Graph Searching with fixed initial positions

In this section we present some basic results and notations that will be used throughout the paper. In particular, we propose a polynomial time algorithm that computes the smallest number of searchers required to monotonously capture an invisible fugitive (without performing any query) in any tree with the extra constraint that initial positions of the searchers are imposed.

We start by making the following observations on non-deterministic search strategies.
First, we observe that any q-branched tree-decomposition of width k of G corresponds to a monotone (search) strategy in G with at most $(k+1)$ searchers which performs at most q queries. Obviously, we can assume that the root is a branching node.

Proposition 1. *Let G be a graph. For any $q \geq 1$, there exists a monotone non-deterministic strategy using at most q queries and $s_q(G)$ searchers, such that the first query step occurs before any removing step.*

The next proposition provides some useful information on connected components of the contaminated part in a monotone strategy after a query step.

Proposition 2. *Let G be a graph. Let $\mathcal{S}$ be a monotone non-deterministic search strategy for clearing G that uses at most k searchers. Let C be a connected component of the contaminated part after a query step. Then at most $k-1$ vertices adjacent to C are occupied by a searcher.*

Proof. For the sake of a contradiction, suppose there is a step of the strategy such that a non-empty connected component of the contaminated part is bordered by all the k searchers. Obviously, during the next move, which must be a removing step, recontamination will happen. This is in contradiction with the monotonicity of $\mathcal{S}$. □

The above proposition has the following corollary in the case of a search strategy which only uses two searchers (recall that useless moves such as placing a searcher on an already cleared vertex are forbidden).

Corollary 1. *Let $\mathcal{S}$ be a non-deterministic monotone search strategy using two searchers. Any placing step, but the first one, consists in placing a searcher on a neighbor of the occupied vertex.*

It is clear that having more than one searcher at a same vertex is irrelevant for a strategy. Thus, we assume throughout this paper that at each moment of a search strategy, each vertex is occupied by at most one searcher. At each step of a strategy, a *free searcher* is a searcher that does not occupy any vertex. If a searcher occupies a vertex v all the neighbors of which are clear, obviously the searcher can be removed from v. By an abuse of the notation, we call such a searcher *free* as well, this meaning that the searcher can immediately become free.

Let G be a graph and $X \subseteq V(G)$ be a (possibly empty) subset of vertices. A search strategy *starting from* X is a monotone (non-deterministic) strategy the first steps of which consist in placing searchers on every vertex in X. For an integer $q \in \mathbb{N}$, define $s_q\{X\}(G)$ as the smallest number of searchers required to catch a fugitive starting from X and performing at most q queries. From the point of view of tree-decompositions, $s_q\{X\}(G)$ is the smallest non-negative integer k such that there exists a q-branched tree-decomposition $(\mathfrak{T}, \mathcal{X})$ of G of width $k+1$ in which $X_{\mathfrak{r}} = X$, for the root $\mathfrak{r}$ of $\mathfrak{T}$. Obviously, $s_q\{X\}(G) \geq |X|$.

3.1 Graph searching with imposed initial searchers' positions

In this section we present a polynomial time algorithm, called `InitPos` and described in Algorithm 1, that for any tree T and any subset $X \subseteq V(T)$ computes $s_0\{X\}(T)$. This algorithm will be used as the cornerstone of the forthcoming algorithms in the upcoming sections.

Roughly speaking, Protocol `InitPos` works as follows. First, initializing $k = |X|$, a searcher is placed on any vertex in X. Then `InitPos` greedily tries to clear T by using k searchers. This is performed in the following way. As long as a new vertex can be cleared without any cost, the corresponding move is performed (Lines 7-11). More precisely, all the free searchers are removed (this consists in removing all the searchers that occupy a vertex all the neighbors of which are occupied by a searcher), and if a searcher A is occupying a vertex v with a single contaminated neighbor u, a free searcher (if any) is placed on u and so A, which is now free, is removed from v. We call such a consecutive sequence of moves a *greedy step* (PossibleGreedy in `InitPos`). When no such a greedy step is possible anymore, Protocol `InitPos` looks for a connected component of the contaminated part that can be cleared by using only the free searchers, i.e., without removing any non-free searcher at this moment (Lines 14-16). For clearing such a connected component of the contaminated part, we use the following result.

Theorem 4 ([19, 15])**.** *For any tree T on n vertices, $s_0(T) = s(T) \leq 1 + \log_3(n-1)$ and $s(T)$ can be computed in time linear in n.*

Algorithm 1 Protocol `InitPos`(T, X) that returns $s_0\{X\}(T)$

Require: A tree T, and a subset $X \subseteq V(T)$

1: **for** k from $|X|$ to $|V(T)|$ **do**
2: // At this step, all vertices of the tree are contaminated and no searchers stand in it //
3: NewClearedComponent $\leftarrow$ TRUE
4: Place a searcher on each vertex of X
5: **while** NewClearedComponent **do**
6: PossibleGreedy $\leftarrow$ TRUE
7: **while** PossibleGreedy **do**
8: **if** a searcher A stands at a vertex v all neighbors of which are clear **then**
9: Remove A
10: **else if** $\exists$ one *free* searcher B, and one searcher A standing at a vertex v, a single neighbor u of which is contaminated **then**
11: Place B on u and remove A
12: **else**
13: PossibleGreedy $\leftarrow$ FALSE
14: Let X' be the set of vertices occupied by a searcher
15: **if** $\exists$ a contaminated connected component S of $T \setminus X'$ and $s(S) \leq k - |X'|$ **then**
16: Clear S using the $k - |X'|$ free searchers
17: **else if** T is clear **then**
18: Return k
19: **else**
20: NewClearedComponent $\leftarrow$ FALSE
21: **end for**

Once such a component has been cleared, new greedy steps may be performed, and so on. If no such component exists, k is increased by one and the whole process restarts from the beginning with one more searcher.

Once the whole tree is cleared, the current value of $|X| \leq k \leq |V(T)|$ is returned. Since the protocol also produces a search strategy for clearing T with k searchers and starting from X, we obviously get $s_0\{X\}(T) \leq k$. To prove the equality $s_0\{X\}(T) = k$, we show that the steps of any strategy clearing T can be reordered so that greedy moves are performed first. Then, we show that if there is a step with no possible greedy move, there must be a connected component of the contaminated part that can be cleared using only free searchers at this step.

The formal statement and proof of our theorem are now as follows.

Theorem 5. *Let T be a tree and $X \subseteq V(T)$ a subset of vertices. Protocol `InitPos` computes $s_0\{X\}(T)$ and a corresponding strategy in polynomial time.*

Proof. Let $\kappa = s_0\{X\}(T) \leq |V(T)|$. For any $k \geq |X|$, by Theorem 4, the k^{th} execution of the *for-loop* of Protocol `InitPos` provides a sequence of placement and removal of k searchers, starting from the vertices of X, and such that no recontamination occurs. There are two cases: either k is large enough and the provided sequence of moves clears the whole tree (Lines 17-18), or the sequence achieves a configuration (positions of the searchers and subset of clear vertices) where no greedy move can be done (i.e., all searchers occupying some vertex have at least two contaminated neighbors) and no contaminated component can be cleared with the remaining searchers (Lines 19-20). The first case obviously occurs for $k = |V(T)|$ which proves that Protocol `InitPos` terminates. We show that it actually returns the integer $k = \kappa$.

Let $\mathcal{S}_0$ be any sequence of moves provided by the $(\kappa + 1 - |X|)^{th}$ execution of the *for-loop*, i.e., using κ searchers. Let N_0 be the number of steps of $\mathcal{S}_0$. Note that the choice of $\mathcal{S}_0$ obviously

depends on order of removal-placement moves in greedy steps and on the choices of the connected components of the contaminated part where the clearing procedure is performed in the protocol. Nevertheless, despite these different choices, we will show that $\mathcal{S}_0$ is a search strategy starting from X, i.e., it clears the whole tree. Therefore, the integer k returned by Protocol `InitPos` satisfies $k \leq \kappa$. Since obviously $\kappa \leq k$, the integer returned by the algorithm is κ and this will prove the theorem.

For the sake of a contradiction, suppose that $\mathcal{S}_0$ does not clear the whole tree. In other words, there are still some contaminated vertices after the N_0 steps of $\mathcal{S}_0$, and there is no possibility to proceed according to the protocol by perform a step $N_0 + 1$ (and thus the current value of $k = \kappa$ has to be increased by one by the *for-loop*). We will derive a contradiction. For this, to any monotone search strategy $\mathcal{S}$ for clearing T which starts from X and uses κ searchers, we associate an integer $\mathfrak{h}(\mathcal{S})$ defined as the first step at which the two strategies $\mathcal{S}$ and $\mathcal{S}_0$ are different (so all the previous steps are the same in $\mathcal{S}$ and $\mathcal{S}_0$). Among all the (monotone) search strategies $\mathcal{S}$ which clear T by starting from X and which use κ searchers, let $\mathcal{S}_*$ be the one which has the largest value of $\mathfrak{h}(\mathcal{S})$. Since the first $|X|$ moves consist of placing searchers on the nodes which belong to X, modulo a reordering, we can assume that the first $|X|$ steps in any search strategy $\mathcal{S}$ and in $\mathcal{S}_0$ are the same, i.e., $\mathfrak{h}(\mathcal{S}_*) \geq |X| + 1$.

We divide the proof into two main cases depending on whether $\mathfrak{h}(\mathcal{S}_*) \leq N_0$ or $\mathfrak{h}(\mathcal{S}_*) > N_0$.

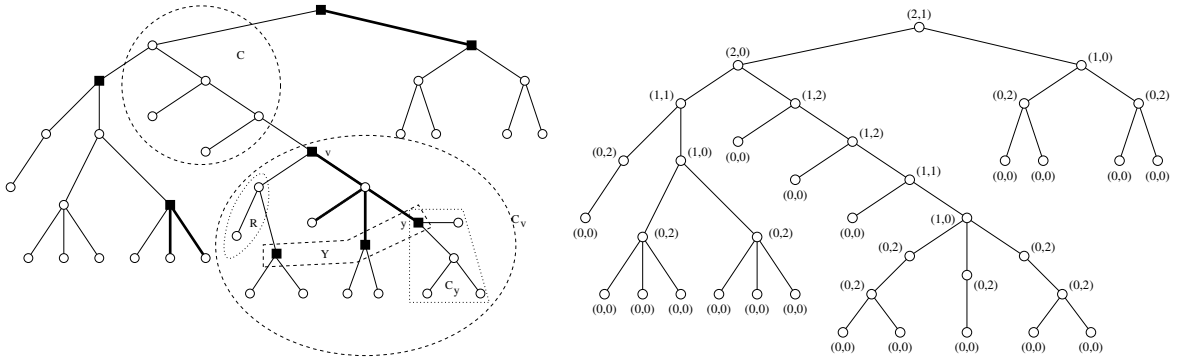
Case I. $\mathfrak{h}(\mathcal{S}_*) \leq N_0$. In other words, $\mathfrak{h}(\mathcal{S}_*)$ is a step in $\mathcal{S}_0$. We show that there is a search strategy $\mathcal{S}'$ with $\mathfrak{h}(\mathcal{S}') > \mathfrak{h}(\mathcal{S}_*)$, which obviously contradicts the choice of $\mathcal{S}_*$.

The proof is divided into three sub-cases depending on the different possibility for the step $\mathfrak{h}(\mathcal{S}_*)$ in $\mathcal{S}_0$. The first two sub-cases below correspond the greedy moves.

Case I-1. Assume first that the step $\mathfrak{h}(\mathcal{S}_*)$ of $\mathcal{S}_0$ consists of removing a searcher from a vertex u all neighbors of which are clear or occupied (Lines 8-9 or the second step in Line 11). Given that the first $\mathfrak{h}(\mathcal{S}_*) - 1$ steps are the same in $\mathcal{S}_0$ and $\mathcal{S}_*$, after step $\mathfrak{h}(\mathcal{S}_*) - 1$ of $\mathcal{S}_*$, there is a searcher at u and all neighbors of u are clear or occupied. Consider the search strategy $\mathcal{S}'$ obtained by modifying $\mathcal{S}_*$ as follows: i) apply the first $\mathfrak{h}(\mathcal{S}_*) - 1$ steps of $\mathcal{S}_*$, ii) remove the searcher from u, and iii) apply all the remaining steps of $\mathcal{S}_*$ (starting from step $\mathfrak{h}(\mathcal{S}_*)$) but possibly the step that removes the searcher from u if such a move is done in $\mathcal{S}_*$ (which is already performed). Clearly, $\mathcal{S}'$ is a monotone search strategy which clears the tree and $\mathfrak{h}(\mathcal{S}') > \mathfrak{h}(\mathcal{S}_*)$, a contradiction.

Case I-2. Assume now that the step $\mathfrak{h}(\mathcal{S}_*)$ of $\mathcal{S}_0$ consists of placing a searcher on a vertex u that is the single contaminated neighbor of an occupied vertex v (Lines 10-11). By the definition of $\mathfrak{h}(\mathcal{S}_*)$, after step $\mathfrak{h}(\mathcal{S}_*) - 1$ of $\mathcal{S}_*$, there is a searcher placed on v and v has a single contaminated neighbor u. Let $s > \mathfrak{h}(\mathcal{S}_*)$ be the step of $\mathcal{S}_*$ that places a searcher on u (such a step clearly exists since $\mathcal{S}_*$ clears T). Note that, because of the monotonicity of $\mathcal{S}_*$, a searcher must occupy v on any step between the step $\mathfrak{h}(\mathcal{S}_*) - 1$ and the step s. Let $\mathcal{S}'$ be the search strategy obtained from $\mathcal{S}_*$ by the following modifications: i) proceed as the first $\mathfrak{h}(\mathcal{S}_*) - 1$ steps of $\mathcal{S}_*$, ii) place a searcher on u and remove the one from v, and iii) apply all the remaining steps of $\mathcal{S}_*$ (starting from the step $\mathfrak{h}(\mathcal{S}_*)$) but the step s and possibly the step that removes later the searcher from u if such a move is done in $\mathcal{S}_*$. Clearly, $\mathcal{S}'$ clears the tree and $\mathfrak{h}(\mathcal{S}') > \mathfrak{h}(\mathcal{S}_*)$, again a contradiction.

Case I-3. In the only remaining sub-case, assume that the step $\mathfrak{h}(\mathcal{S}_*)$ of $\mathcal{S}_0$ is a step corresponding to Lines 15-16 of Protocol `InitPos`, that is an intermediate step in clearing a contaminated part of the tree. Let $s < \mathfrak{h}(\mathcal{S}_*)$ be the last greedy step of



(a) Proof of Theorem 5 (Case II) (b) Algorithm 4

Figure 1: (Left) Example of notations used in the fourth case of Proof of Theorem 5. The black squares represent searchers, bold edges induce the clear components, Y consists of three occupied nodes, $|V(C)| = 5, |V(C_v)| = 16, |V(C_y)| = 5$ and $|V(R)| = 2$. (Right) Example of the labeling computed by Protocol `TwoSearchers` (Algorithm 4) of a tree rooted in r.

$\mathcal{S}_0$ before step $\mathfrak{h}(\mathcal{S}_*)$ and let X' be the set of vertices occupied by the searchers at this step of $\mathcal{S}_0$. At step $\mathfrak{h}(\mathcal{S}_*)$, $\mathcal{S}_0$ clears a contaminated component C of $T \setminus X'$. Let $s' \geq \mathfrak{h}(\mathcal{S}_*) > s$ be the first step in $\mathcal{S}_0$ when the component C is cleared. Note that, $s' \leq N_0$ since otherwise Protocol `InitPos` would stop just after step s. According to Lines 15-16 of Protocol `InitPos`, after the step s, $\mathcal{S}_0$ clears the component C by using $\kappa - |X'|$ searchers.

Let $\mathcal{S}'$ be the monotone search strategy obtained by modifying $\mathcal{S}_*$ as follows: i) proceed as the first s steps in $\mathcal{S}_0$ (that are also the first s steps in $\mathcal{S}_*$ since $s < \mathfrak{h}(\mathcal{S}_*)$), ii) apply all the steps from $s+1$ to s' in $\mathcal{S}_0$ that clear C and perform the greedy moves consisting of removing all the searchers from the vertices in C and from the vertices of X' with only clear neighbors, iii) proceed according to $\mathcal{S}_*$ after its step $\mathfrak{h}(\mathcal{S}_*)$ by avoiding all the moves concerning C (placing/removing a searcher on/from a vertex of C).

Note that the phase iii) above is possible. Indeed, let s'' be the last step of $\mathcal{S}'$ before phase iii). After step s'' of $\mathcal{S}'$, when a move is performed in $\mathcal{S}'$, there are at least as many free searchers as when the corresponding step is done is $\mathcal{S}_*$, in other words the corresponding move can be performed in $\mathcal{S}'$ using κ searchers.

Finally, to get the contradiction, note that $\mathfrak{h}(\mathcal{S}') > s' \geq \mathfrak{h}(\mathcal{S}_*)$.

Case II. $\mathfrak{h}(\mathcal{S}_*) > N_0$, in other words $\mathfrak{h}(\mathcal{S}_*) = N_0 + 1$. Since by our assumption, $\mathcal{S}_0$ does not clear T, we have $\mathfrak{h}(\mathcal{S}_*) \leq |\mathcal{S}_*|$. That is, after step $\mathfrak{h}(\mathcal{S}_*) - 1$ of $\mathcal{S}_0$, there are no possible greedy moves and no contaminated components of $T \setminus X'$ can be cleared using $\kappa - |X'|$ searchers. In other words, after this step, the algorithm cannot continue with κ searchers while the tree is not yet clear (Lines 19-20). Both $\mathcal{S}_0$ and $\mathcal{S}_*$ have the same configuration after step $\mathfrak{h}(\mathcal{S}_*) - 1$. Let X' be the set of vertices occupied by a searcher at this step. Considering $\mathcal{S}_*$, we show that, after step $\mathfrak{h}(\mathcal{S}_*) - 1$, there is a contaminated component C of $T \setminus X'$ such that $s(C) \leq \kappa - |X'|$. This will obviously be in contradiction with the assumption that the algorithm cannot continue with κ searchers after step N_0 in $\mathcal{S}_0$.

Let C be the first connected component among all the connected components of the contaminated part of $T \setminus X'$ which becomes totally clear by the search strategy $\mathcal{S}_*$, and let $s \geq N_0 + 1$

be the step of $\mathcal{S}_*$ when all the vertices of C are clear. We claim that at any step of $\mathcal{S}_*$ between $\mathfrak{h}(\mathcal{S}_*)$ and s, at least $|X'|$ searchers are occupying the vertices of $T \setminus C$. In other words, at each intermediate step in $\mathcal{S}_*$ between N_0 and s, there are at most $\kappa - |X'|$ free searchers. This clearly implies that $s(C) \leq \kappa - |X'|$, and proves the existence of C as stated above.

To show this claim, we proceed as follows. For any vertex $v \in X'$, let $e_v = \{v, u\}$ be the edge incident to v which lies on the path which connect v to C in T (possibly, $u \in V(C)$). Let C_v be the connected component of $T \setminus e_v$ that contains v. We prove by induction on $|X' \cap V(C_v)|$ that at any step of $\mathcal{S}_*$ between steps $\mathfrak{h}(\mathcal{S}_*)$ and s, at least $|X' \cap V(C_v)|$ vertices in C_v are occupied by a searcher. This will clearly imply the claim. Indeed, considering the subset $W \subseteq X'$ of all the occupied vertices at step $\mathfrak{h}(\mathcal{S}_*) - 1$ which are incident by an edge to C, for any $v \in W$, at least $|X' \cap V(C_v)|$ searchers must occupy the vertices of C_v at any step between $\mathfrak{h}(\mathcal{S}_*) = N_0 + 1$ and s in $\mathcal{S}_*$. Since $|X'| = \sum_{v \in W} |X' \cap V(C_v)|$, this shows that $|X'|$ searchers must be places on some vertices of $T \setminus C$ at any step between $\mathfrak{h}(\mathcal{S}_*)$ and s, which is the assertion of the above claim.

To proceed by the induction, consider first the base case where $|X' \cap V(C_v)| = 1$. Since v has at least two contaminated neighbors (because by assumption no greedy move is possible after step N_0 in $\mathcal{S}_0$), there must exist a connected component $R \neq C$ of the contaminated part of $T \setminus X'$ such that $R \subseteq C_v$, and such that R has an edge incident to v. Since C is cleared before R and the strategy is monotone, at any step between $\mathfrak{h}(\mathcal{S}_*)$ and s, at least one searcher occupies a vertex of $R \cup \{v\} \subseteq C_v$ and this proves the base of induction.

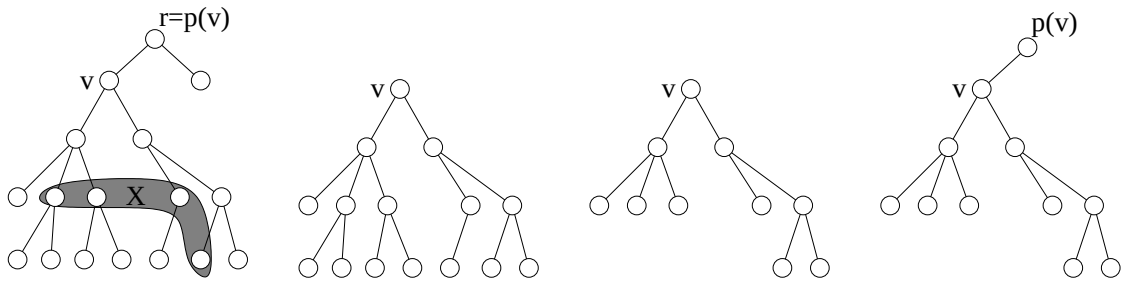
Let us assume now that $|X' \cap V(C_v)| > 1$ and for any integer strictly smaller than $|X' \cap V(C_v)|$ the induction hypothesis holds. This case is depicted in Figure 1(a). Consider the subset $Y \subseteq (X' \cap V(C_v)) \setminus \{v\}$ containing all the vertices $y \neq v$ that are in $X' \cap V(C_v)$ such that no other vertex of X' is on the unique path between y and v. Note that for any $y \in Y$, $|X' \cap V(C_y)| < |X' \cap V(C_v)|$ and so, according to the induction hypothesis, at any step between $\mathfrak{h}(\mathcal{S}_*)$ and s, there are at least $|X' \cap V(C_y)|$ searchers which occupy some vertices of C_y. Moreover, at least one connected components $R \neq C$ of the contaminated part of $T \setminus X'$ has an edge incident to v. Note that, by definition of Y, for any $y \in Y$, $R \cap C_y = \emptyset$. Since C is cleared before R and the strategy is monotone, at any step between $\mathfrak{h}(\mathcal{S}_*)$ and s, at least one searcher occupies a vertex of $R \cup \{v\} \subseteq C_v$. This shows that, at any step between $\mathfrak{h}(\mathcal{S}_*)$ and s at least $1 + \sum_{y \in Y} |X' \cap V(C_y)| = |X' \cap V(C_v)|$ searchers must occupy some vertices of C_v, and thus, the induction hypothesis also holds for $|X' \cap V(C_v)|$, and so the claim follows.

This finishes the proof of the first part of the theorem.

Note that at each execution of the *while-loop* (Line 5) of the protocol, either at least one vertex is cleared, or one searcher becomes free. Therefore, there are at most n^2 executions of this loop. Moreover, each execution of this loop first tests the neighborhood of the occupied nodes at most twice (Lines 8 and 10), and then it tests each contaminated component S by computing $s(S)$ (Line 15). Since, these components are disjoint then the sum of their sizes is less than n and, by Theorem 4, this latter computation is performed in linear time. Hence, each execution of the *for-loop* of Protocol `InitPos` is performed in polynomial time. □

3.2 Further notations

We now introduce some extra terminology and notations that will be used in the next sections. Let T be a tree rooted in $r \in V(T)$. For any $v \in V(T)$, let $p(v)$ denote the parent of v (we set $\{p(r)\} = \emptyset$), and let T_v denote the subtree of T rooted in v. By $\hat{T}_v$ we denote the subtree induced by $V(T_v) \cup \{p(v)\}$. For any subset $X \subseteq V(T_v) \setminus \{v\}$ with the property that no vertex of X is on the path between v and any other vertex of X, we denote by $S_X(v)$ the component of $T_v \setminus X$ that contains v, and denote by $C_X(v)$ (resp., $\hat{C}_X(v)$) the subtree of T_v induced by



(a) T (b) T_v (c) $C_X(v)$ (d) $\hat{C}_X(v)$

Figure 2: Notations.

$V(S_X(v)) \cup X$ (resp., $V(S_X(v)) \cup X \cup \{p(v)\}$) (see Figure 2). In other words, $C_X(v)$ is obtained from T_v by removing all vertices that have an ancestor in X.

For any subset $I \subseteq V(T)$ and any $v \in V(T) \setminus I$, the subset $X(v, I) \subseteq I$ denotes the set of all descendants u of v that are in I with the property that no internal vertices of the path between u and v belong to I. Finally, let us assume some vertices of a tree T are labeled with integers in $\{0, \cdots, q\}$, and a vertex $v \in V(T)$ is not labeled. For any $0 \leq i \leq q$, to simplify the notation, we denote by $X(v, i)$ the subset $X(v, I_i) \subseteq I_i$ where here I_i is the set of all labeled vertices of T with label larger or equal to i.

A *caterpillar* K is a tree that has a dominating path. That is, there is a path P in K such that any vertex in $V(K)$ either belongs to $V(P)$ or has a neighbor in $V(P)$.

4 To clear a tree by performing one query

The purpose of this section is to design an algorithm, called `OneQuery`, that computes $s_1(T)$, the smallest number of searchers required to clear a tree T when at most one query can be performed. Algorithm `OneQuery` uses, as a black box, Algorithm `TwoSearchers` (whose design is postponed to Section 6) which computes $q_2(T)$ the smallest number of queries required to clear a given tree T with two searchers.

Note that by Proposition 1, a *one-limited search strategy*, i.e., a search strategy with one allowed query, consists of the following three basic steps

(1) placing searchers on every vertex in $X \subseteq V(G)$, with $|X| \leq s_1(G)$,

(2) performing the query to locate the contaminated part C of the graph, and

(3) clearing the contaminated component C by starting from the vertices of X that are adjacent to C and by using $s_1(G)$ searchers.

More formally, for any tree T and any $k \geq 1$, $s_1(T) \leq k$ if and only if there exists a subset $X \subseteq V(T)$ such that $|X| \leq k$, and for any connected component C of $T \setminus X$ we have $s_0\{Y\}(C') \leq k$ where Y is the set of vertices in X that are adjacent to a vertex in C and C' is the connected component of T induced by $C \cup Y$.

We prove that for any tree T, Protocol `OneQuery` described in Algorithm 2 computes $s_1(T)$, and a corresponding strategy, in polynomial time. Note that, in `OneQuery` we first use Algorithm `TwoSearchers` to test if $s_1(T) = 2$ by checking if $q_2(T) = 1$.

Once this has been done, given an integer $k \geq 1$, the aim of Protocol `OneQuery` will be to find a subset $X \subseteq V(T)$ such that the connected components of $T \setminus X$ can be cleared in the way described above, and such that in addition, these components are as large as possible. Performing

Algorithm 2 Protocol `OneQuery`(T) that returns $s_1(T)$

Require: A tree T

1: **if** $|V(T)| = 1$ **then** return 1
2: **if** `TwoSearchers`(T) returns ≤ 1 **then** return 2
3: **for** k from 3 to $|V(T)|$ **do**
4: **for all** $r \in V(T)$ **do**
5: Let T be rooted in r
6: Label all leaves (vertices $\neq r$, with degree 1) with 0
7: **while** it remains an unlabeled vertex $v \in V(T)$ **do**
8: Let v be an unlabeled vertex every child of which has a label
9: **if** $s_0\{X(v,1) \cup p(v)\}(\hat{C}_{X(v,1)}(v)) \leq k$ **then**
10: label v with 0
11: **else**
12: label v with 1
13: Let b_1 be the number of vertices labeled 1 in T_v
14: **if** (v is not the root and $b_1 = k$) or $b_1 > k$ **then** Goto Line 4
15: Return k

in such a way allows to minimize the size of X. If such a subset X of size $|X| \leq k$ exists, we infer that $s_1(G) \leq k$ (otherwise, $s_1(G) > k$). Roughly speaking, Protocol `OneQuery` proceeds as follows. Let $k \geq 1$ be a fixed integer, and let T be rooted in $r \in V(T)$. Protocol `OneQuery` labels the vertices of T with labels 0 and 1 by starting from the leaves and by proceeding towards the root. At the end of this labeling procedure, the subset $X \subseteq V(T)$, that we are looking for, will consist of those vertices which are assigned label 1. Therefore, if at most k vertices are labeled 1, a one-limited strategy starts by placing the searchers on these vertices and performing a query. The way the labeling procedure has been performed ensures that for any maximal (with respect to inclusion order) connected subset C of vertices labeled 0, and for Y the set of vertices adjacent to C and labeled 1, we have $s_0\{Y\}(C \cup Y) \leq k$. Therefore, if Protocol `OneQuery` returns k, then $s_1(T) \leq k$.

The details of the labeling procedure are as follows and depicted in Figure 3. The procedure is done for T rooted in r for any possible root $r \in V(T)$ (Line 4). A vertex $v \in V(T)$ is labeled once all its children are assigned a label. $X(v,1)$ is defined as the set of descendants u of v that are labeled with 1, and such that the internal vertices of the path between u and v are labeled 0. (The notations are the same as in Section 3.2.) Knowing that a searcher has to be placed on each vertex of $X(v,1)$ before the first query (see the discussion in the previous paragraph), Protocol `OneQuery` tests whether it is necessary to place a searcher on v or not. The answer is yes if and only if by not placing a searcher on v before the query, the component which contains v and $X(v,1)$, and has internal nodes of label 0, creates a connected component that cannot be cleared with at most k searchers by starting from the position of the searchers just after the query, i.e. at its border. That is, under the (testing) assumption that a searcher is not placed on v, this connected component contains $\hat{C}_X(v)$, and Protocol `OneQuery` tests whether k searchers starting from $X(v,1) \cup \{p(v)\}$ can clear $\hat{C}_{X(v,1)}(v)$. v is labeled 0 if this is the case, and it is labeled 1 otherwise.

We notice that Lines 13-14 can be replaced by "If more than k vertices are labeled 1 Goto Line 4" without modifying the result achieved by Protocol `OneQuery`. However, we have presented Protocol `OneQuery` in this way to make it fully equivalent to Protocol `Approx` (for $q = 1$) described in Section 5 (see Lemma 1).

Figure 3 illustrates the execution of Protocol `OneQuery`(T) when T is rooted in r and with

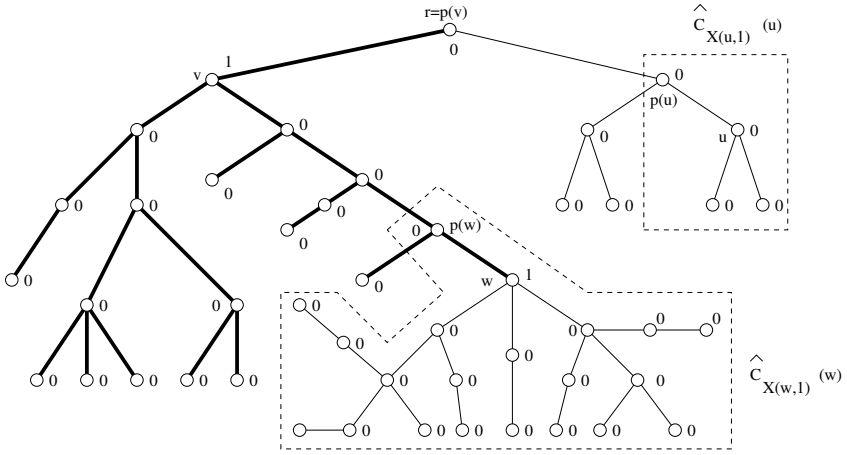


Figure 3: Result of Algorithm 2 on T rooted in r and $k = 3$. Bold edges are the edges of $\hat{C}_{X(v,1)}(v)$.

$k = 3$. During this execution, $X(u, 1) = \emptyset$ and u is labeled with 0 because $\hat{C}_{X(u,1)}(u)$ (with 4 nodes) can be cleared starting from $X(u, 1) \cup \{p(u)\} = \{p(u)\}$ using no query with 3 searchers. Similarly $X(w, 1) = \emptyset$ since there are no descendants of w labeled with 1, however w must be labeled 1 because it is not possible to clear $\hat{C}_{X(w,1)}(w)$ starting from $X(w, 1) \cup \{p(w)\} = \{p(w)\}$, using no query and 3 searchers. To see this, it is sufficient to see that, first a searcher can be placed on w and the one at $p(w)$ is then removed. Then, the searcher at w cannot be freed while two of its branches (components of $T_w \setminus \{w\}$) are cleared. Finally, since two of its branches are not caterpillar, the two remaining searchers cannot clear them without any query. The last example we describe is the one when labeling v. $X(v, 1) = \{w\}$ and $\hat{C}_{X(v,1)}(v)$ is the subtree induced by the bold edges. Again, it is easy to check that $\hat{C}_{X(v,1)}(v)$ cannot be cleared without query, using three searchers and starting from $X(v, 1) \cup \{p(v)\} = \{w, p(v)\}$. Therefore, v receives label 1. Finally, $s_1(T) \leq 3$ and a strategy consists of placing two searchers on v and w, performing the query and clearing the remaining contaminated component with 3 searchers, starting from v and w.

Theorem 6. *For any tree T, Protocol `OneQuery`(T) computes $s_1(T)$ and a corresponding one-limited strategy in polynomial time.*

Proof. The result clearly holds if $s_1(T) \leq 2$ by Lines 1-2 and by Theorem 9 (Section 6). In the following, we assume $s_1(T) \geq 3$. Let $k \geq 3$ be the integer returned by `OneQuery`(T). Let T be rooted in the vertex which gives the output of the algorithm. As we said before, the strategy consists in placing the searchers on the vertices labeled 1, and performing the query. Then, the connected component C that remains contaminated can be cleared with k searchers by starting from the vertices labeled 1 in the border of C (Lines 9-10). Thus, certainly $s_1(T) \leq k$ holds.

To prove the equality, let $\mathcal{S}$ be a monotone one-limited search strategy for clearing T that uses at most $k > 2$ searchers. By Proposition 1, we may assume that the first steps in $\mathcal{S}$ consist of placing at most k searchers on the vertices of a subset $I \subseteq V(T)$ ($I \neq \emptyset$), and then performing the query. We consider the labeling of the vertices of T obtained by `OneQuery`(T) (Lines 5-12) when T is rooted in a vertex $r \in I$. For any vertex $v \in V(T)$, define $j_v = |I \cap V(T_v)|$. We prove by induction on j_v that there are at most j_v vertices labeled 1 in T_v. Since $j_v < k$ for any $v \neq r$ (because $r \in I$) and $j_r = |I \cap V(T_r)| = |I| \leq k$, this proves that after the execution of Lines 5-12 (for the vertex v, when T is rooted in r), since $b_1 \leq j_v$, Line 14 is not executed, there is no return to Line 4. Hence, the output of `OneQuery`(T) is at most k, and this finishes the proof of

our theorem.

To prove the base of our induction, let v be a vertex with $j_v = 0$. Obviously, since $\mathcal{S}$ is monotone and there is no recontamination, for any $w \in V(T_v)$, one can derive from $\mathcal{S}$ a strategy $\mathcal{S}_w$ that clears $\hat{T}_w = T_w \cup \{p(w)\}$ by starting from $p(w)$ and by using at most k searchers, without performing any query. In other words, $s_0\{p(w)\}(\hat{T}_w) \leq k$. Thus, by the definition of our labeling procedure (Lines 9-10), all vertices of T_v are labeled 0.

Consider now a vertex v with $j_v > 0$, and suppose that for any u with $j_u < j_v$ the claim holds. We divide the proof into two parts depending on whether $v \in I$ or not.

If $v \in I$, the result can be easily obtained by applying the induction hypothesis to the children of v. Indeed, since $v \in I$, for any child u of v, we have $j_u < j_v$, and thus, by the hypothesis of our induction, there are at most $j_u = |I \cap T_u|$ vertices labeled 1 in T_u. This shows that there are at most $\sum_u j_u = |I \cap T_v| - 1$ vertices labeled 1 in $T_v \setminus \{v\}$, where the sum is over all the children of v. We infer that there are at most $j_v = |I \cap T_v|$ vertices labeled 1 in T_v.

Now, suppose that $v \notin I$. Let $X(v, I) = \{v_1, \cdots, v_\ell\}$ ($\ell \geq 1$) be the set of all descendants u of v that belong to I, and such that there is no other vertex of I on the path between v and u (the notations are the same as in Section 3.2). For any $i \leq \ell$ and any child z of v_i, we have $j_z < j_v$ and the induction hypothesis holds for z. That is, for any $i \leq \ell$ and any child z of v_i, there are at most j_z vertices labeled 1 in T_z.
Consider a vertex $w \in C_X(v)$ that does not lie on any path between v and v_i for any $i \leq \ell$. Obviously, for such a vertex we must have $j_w = 0$. By the base of our induction, all vertices in T_w are labeled 0. In other words, this shows that any vertex of $C_X(v)$ which is labeled 1 has to belong to a path between v and v_i for some value of $1 \leq i \leq \ell$. For any $i \leq \ell$, let u_i be the vertex (if any) of the unique path between v_i and v, including v_i and v, that is labeled 1 and which is closest to v_i. Let U be the set of all the vertices u_i, $1 \leq i \leq \ell$. It is clear that $|U| \leq \ell = |X(v, I)|$. We claim there is no other vertex of $C_X(v) \setminus U$ with label 1. For the sake of a contradiction, suppose this is not the case and let w be a vertex of $C_X(v) \setminus U$ which is assigned label 1 by Lines 6-12 of Protocol `OneQuery`(T). Let $U' = X(w, U)$, the set of descendants u of w which belong to U and verify the property that there is no other vertex of U on the unique path between w and u. By the definition of U, $\hat{C}_{U'}(w)$ is a subtree of $\hat{C}_X(v)$. In addition, since $U' \subset U$ and $\mathcal{S}$ is monotone, one can easily derive from $\mathcal{S}$ a strategy for clearing $\hat{C}_{U'}(w)$ which starts from $\{p(w)\} \cup U'$ and uses at most k searchers, and which does not perform any query. This shows that $s_0\{\{p(w)\} \cup U'\}(\hat{C}_{U'}(w)) \leq k$, and by the definition of the labeling scheme (Lines 9-10), the label assigned to w is 0, a contradiction. We infer that there are at most $|U| \leq |X(v, I)| = \ell$ vertices in $C_X(v)$ which are labeled 1.

To conclude, the number of vertices labeled 1 in T_v is bounded above by $\ell + \sum_z j_z = \ell + \sum_z |T_z \cap I| = |T_v \cap I| = j_v$. Here, in the sums, z runs over all the children of a vertex $v_i \in X(v, I)$, for $1 \leq i \leq \ell$. The fact that `OneQuery` performs in polynomial time directly follows from Theorem 5. □

5 A polynomial time algorithm for rs_q in trees

This section is devoted to presenting Protocol `Approx`, formally described in Algorithm 3, that computes $rs_q(T)$ in polynomial time for any tree T and any $q > 0$. Combined with Theorem 3, this leads to a polynomial time 2-approximation algorithm for computing s_q in trees.

We start by characterizing the family of all trees with restricted q-limited search number two. This will be later used in the design of our algorithms. Recall that the *height* of a tree T is the smallest integer h such that there exist two vertices $u, v \in V(T)$ called the *centers* of T, so that either $u = v$, or u and v are adjacent, and such that any vertex $w \in V(T)$ is at distance

strictly smaller than h from u or v. A tree is called *q-simple* if it is obtained from any tree S of height at most q by attaching to any $w \in V(S)$ an arbitrary number of paths of arbitrary length.

Theorem 7. *Let T be a tree. Then $rs_q(T) = 2$ iff T is q-simple and $|V(T)| > 1$. Furthermore, this can be decided in linear time.*

Proof. Let T be q-simple with $|V(T)| > 1$. Since $|V(T)| > 1$, $rs_q(T) > 1$. Let S be a tree of height at most q from which T is obtained by the addition of some paths. We describe a strategy for clearing T using two searchers and at most q queries. Place the two searchers on the center(s) of S and perform the first query. Then, after each query, remove the searcher in the clear component and place it on the neighbor of the other searcher in the contaminated component and perform the next query. By definition of the height, when the last query has been performed, the contaminated part of the tree (if not empty) is a path an end of which is occupied by a searcher. The capture of the fugitive follows easily. Hence, $rs_q(T) = 2$.

By the definition of restricted non-deterministic graph searching and by Corollary 1, for any restricted strategy using two searchers, the set of vertices that have been occupied by the searchers until the j^{th} query must induce a path with j vertices. The result follows easily.

Clearly, one can decide whether a tree is q-simple in linear time. □

5.1 Good decomposition of a labeled tree

Given a subtree S of a tree T, define the *border* $\partial_T(S)$ of S in T as the set of all vertices of S that are adjacent in T to some vertices in $T \setminus S$.

Let T be a labeled tree in which each vertex has a label in $\{0, 1, \ldots, q\}$. For any $0 \leq \ell \leq q$, define the (unique) family $\mathcal{F}_\ell$ of subtrees (possibly reduced to one edge) of level ℓ as the family of all the (inclusion) maximal subtrees S of T with the property that all the internal nodes of S have a label smaller or equal to ℓ. Note that such a maximal subtree can be reduced to an edge with vertices labeled $> \ell$.

The following easy remarks are in order. First, note that $\mathcal{F}_q = \{T\}$. Let $\ell \in \{0, \cdots, q-1\}$. By definition, for any subtree of level ℓ, $S \in \mathcal{F}_\ell$, any internal vertex of S has a label $\leq \ell$, therefore, the border of S, $\partial_T(S)$, is simply the set of all leaves of S that have a label at least equal to $\ell+1$. Moreover, it is clear that the union of all the elements of $\mathcal{F}_\ell$ covers T, and any two distinct subtrees $S, S' \in \mathcal{F}_\ell$ intersect in at most one vertex $v \in \partial_T(S) \cap \partial_T(S')$ (in particular, the label of v is at least $\ell + 1$).

Definition 1. The collection $\{\mathcal{F}_0, \cdots, \mathcal{F}_q\}$ is a *k-good decomposition* of T if it satisfies the following two properties.

(I) For any $0 < \ell \leq q$, any subtree $S \in \mathcal{F}_\ell$ contains at most k vertices with label at least ℓ, and

(II) for any subtree $S \in \mathcal{F}_0$, we have $s_0\{\partial_T(S)\}(S) \leq k$.

Figure 4 illustrates a 4-good-decomposition of a tree T. In Figure 4, a triangle represents a subtree, a circle is the common vertex between two adjacent triangles and it is a leaf in both corresponding subtrees. The integer in a circle is the label of the corresponding vertex and any vertex not depicted by a circle is labeled with 0. $\mathcal{F}_0$ is the set of triangles, i.e., each triangle is a subtree that can be cleared without query by 4 searchers starting from its (at most three) leaves depicted by circles. $\mathcal{F}_3 = \{T\}$. The family $\mathcal{F}_2$ is represented by the width of the border of the triangles (red bold or black thin): two adjacent triangles with the same border-width belong to the same subtree of $\mathcal{F}_2$. $|\mathcal{F}_2| = 4$. The family $\mathcal{F}_1$ is represented by the colors (gray

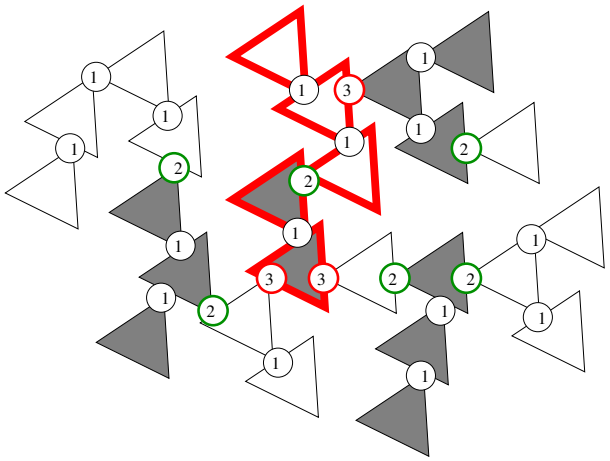


Figure 4: Shape of a 4-good-decomposition of a tree T. $\mathcal{F}_3 = \{T\}$, $|\mathcal{F}_2| = 4$ and $|\mathcal{F}_1| = 10$

or white) of triangles: two adjacent triangles with the same color belong to the same subtree of $\mathcal{F}_1$. $|\mathcal{F}_1| = 10$.

The significance of the above definition is given by the following proposition.

Proposition 3. *If a tree T with labels in $\{0, \cdots, q\}$ gives rise to a k-good decomposition $\{\mathcal{F}_0, \cdots, \mathcal{F}_q\}$, then $rs_q(T) \leq k$.*

Proof. The strategy consists in placing first the searchers on vertices labeled by q (at most k by Property (I) of Definition 1, for $T \in \mathcal{F}_q$), and then performing the first query. Proceeding by induction, for any $1 \leq \ell \leq q$, once the ℓ^{th} query has been performed, the contaminated part consists of a subtree $S_{q-\ell} \in \mathcal{F}_{q-\ell}$ such that

(1) the vertices in $\partial_T(S_{q-\ell})$ are all occupied by a searcher (by definition, they have label at least $q - \ell + 1$), and

(2) no other vertex in $S \setminus \partial_T(S)$ is occupied by a searcher (since they all have label at most $q - \ell$).

All those searchers that occupy a vertex in $V(T \setminus S_{q-\ell})$ are removed, and then placed on the set L of vertices in $S_{q-\ell}$ which are labeled $q - \ell$. By Property (I) in Definition 1, at most k searchers can occupy all the vertices in $|L \cup \partial_T(S)| \leq k$, so this is possible. The $(\ell+1)^{th}$ query is then performed next. Once the q^{th} query has been performed, by Property (II) of Definition 1, the remaining contaminated part $S_0 \in \mathcal{F}_0$ can be cleared starting from $\partial_T(S_0)$ with at most k searchers (since $s_0\{\partial_T(S_0)\}(S_0) \leq k$). This proves the proposition. □

In what follows next, we will prove that if Protocol `Approx`(T, q) returns an integer $k \geq 3$, then the labeling which results from its execution gives rise to a k-good decomposition of T.

5.2 General description of Protocol `Approx`.

The general scheme of `Approx` is as follows. Protocol `Approx` starts by checking first whether $rs_q(T) \in \{1, 2\}$ (Lines 1-2). This can be done by Theorem 7.
Then, fixing an integer $k \geq 3$, Protocol `Approx` aims at computing the smallest number $\tilde{q} \geq 0$ such that there exists a subset $X \subseteq V(T)$, with $|X| \leq k$, and such that for any connected component C of $T \setminus X$, there is a restricted $(\tilde{q}-1)$-limited strategy with the following constraint: Let Y be the set of vertices in X that are adjacent to a vertex in C, and let C' be the connected

Algorithm 3 Protocol `Approx`(T, q) that returns $rs_q(T)$, $q > 0$

Require: A tree T

1: **if** $|V(T)| = 1$ **then** return 1
2: **if** T is q-simple **then** return 2
3: **for** k from 3 to $|V(T)|$ **do**
4: **for all** $r \in V(T)$ **do**
5: Let T be rooted in r and all its vertices be unlabeled
6: Label all leaves (vertices $\neq r$, with degree 1) with 0
7: **while** it remains an unlabeled vertex $v \in V(T)$ **do**
8: Let v be an unlabeled vertex every child of which has a label
9: Let b_j be the number of vertices labeled $\geq j$ in $C_{X(v,j+1)}(v)$, $1 \leq j \leq q$
10: **if** $\exists j > 0$, $b_j > k$ **then**
11: Let m be the greatest integer such that $b_m \geq k$
12: Label v with $m + 1$
13: **else if** v is not the root and ($\exists j > 0$, $b_j = k$ or $|X(v, j)| = k - 1$) **then**
14: Let m be the greatest integer such that $b_m = k$ or $|X(v, m)| = k - 1$
15: Label v with $m + 1$
16: **else if** $s_0\{X(v, 1) \cup \{p(v)\}\}(\hat{C}_{X(v,1)}(v)) \leq k$ **then**
17: Label v with 0
18: **else**
19: Let $k' = k$ if v is the root and $k' = k - 1$ otherwise;
20: Let $m > 0$ be the smallest integer such that $b_m < k'$
21: label v with m
22: **if** a vertex is labeled larger than q **then** Goto Line 4
23: // test another root if possible and increase k by one otherwise
24: **return** k

component of T induced by $C \cup Y$. Then the strategy uses k searchers for clearing C', and starts by placing the searchers on the vertices of a superset of Y and performing the first query. If at some step of the algorithm $\tilde{q}$ becomes larger than q, the allowed number of queries, then `Approx` increases k by one and restart the whole procedure. The way this is done is similar to Protocol `OneQuery`: Protocol `Approx` aims at finding a subset $X \subseteq V(T)$ such that the connected components of $T \setminus X$ can be cleared in the way described above, and such that these components are as large as possible.

More precisely, the aim of Protocol `Approx`(T, q) is to produce a labeling of all the vertices of T with labels in $\{0, \cdots, q\}$ giving rise to a k-good decomposition $\{\mathcal{F}_0, \cdots, \mathcal{F}_q\}$ for T, and for the smallest value of k, such that in addition the subtrees in $\mathcal{F}_0$ are as large as possible. Once such a labeling has been found, Proposition 3 (and its proof) shows that $rs_q(T) \leq k$ and produces a strategy for clearing of T. The proof of the reverse inequality, yielding to the equality of $rs_q(T)$ with the output of the algorithm, is similar to the proof of Theorem 6 in the case $q = 1$. We show that $rs_q(T) \geq k$ by induction on q and next Lemma serves as basis of the induction.

Lemma 1. *For $q = 1$ Protocol `Approx` proceeds as `OneQuery`* (Algorithm 2)*.*

Proof. If $|V(T)| = 1$ or T is 1-simple, clearly both algorithms achieve the same result.

The only difference between both algorithms is that Protocol `OneQuery` labels the vertices only with 0 or 1, and stops if at some step, strictly more than k vertices have received label 1, while Protocol `Approx` increases the label used, i.e., it uses the label 2, when more than k vertices have received the label 1. But, at this step, Line 22 of `Approx` stops the execution as well.

More formally, consider an execution of both `OneQuery`(T) and `Approx`$(T, 1)$ when T is rooted in some $r \in V(T)$ and for some given k. Suppose that the vertex $v \in V(T)$ is the vertex which is going to be labeled, and assume that all the vertices in $V(T_v) \setminus \{v\}$ have already received the same labels by both algorithms.

To avoid confusion, note that the variable b_1 used by `Approx`$(T, 1)$ is the number of vertices labeled with 1 in T_v before labeling v (i.e., the number of vertices labeled with 1 in T_v without considering v) while the variable b_1 used by `OneQuery`(T) is the number of vertices labeled with 1 in T_v after labeling v (considering v).

If Protocol `Approx`$(T, 1)$ executes Line 12, i.e., the number of vertices labeled 1 in $T_v \setminus \{v\}$ is strictly larger than k, then v is labeled with 2 and Line 22 will initiate another execution (with another root, or by increasing k by one). But in this case, after labeling the vertex preceding v, Line 14 of `OneQuery` will also do the same.

If `Approx`$(T, 1)$ executes Line 15, again Line 22 will initiate another execution. In this case, v is not the root. Moreover, either strictly larger than k vertices are labeled 1 in T_v and Line 14 of `OneQuery` will do the same. Or $|X(v, 1)| = k - 1$. But in this case, by Proposition 2, $s_0\{X(v, 1) \cup \{p(v)\}\}(\hat{C}_{X(v,1)}(v)) > k$, and Line 12 of `OneQuery` labels v with 1, and then $b_1 = k$ (Line 13 of `OneQuery`). In the latter case, again, Line 14 of `OneQuery` initiates another execution.

Hence, we may assume that Line 16 of Protocol `Approx` and Line 9 of Protocol `OneQuery` are executed, which perform the same test.

If v is labeled with 0 by both algorithms, then Protocol `Approx` will try to label the next vertex. On the other side, `OneQuery` does the same. Indeed, for the sake of a contradiction, suppose instead, `OneQuery` initiates another execution. This means either, $b_1 > k$, or, $b_1 = k$, and v is not the root. But, since v is labeled 0, the variable b_1 considered at Line 13 of `OneQuery` has the same value as the variable b_1 considered at Line 9 of `Approx`$(T, 1)$. This means `Approx`$(T, 1)$ should have executed Line 12 or Line 15, which is clearly a contradiction.

Finally, let us assume that v is not labeled 0, and hence, it is labeled 1 by `OneQuery`. Suppose first that `Approx` labels v with an integer strictly larger than 1. This being the case, Line 22 will initiate another execution. In this case, it is easy to check that `OneQuery` will execute Line 14, i.e., initiate another execution. In the only remaining case, suppose that `Approx` labels v with 1. Then it tries to label the next vertex or stops if v is the root. This means that the variable b_1 of `Approx` is strictly smaller than k' where $k' = k$ if v is the root, and $k' = k - 1$ otherwise. But then, the variable b_1 of `OneQuery` is smaller or equal to k', and thus, `OneQuery` does the same as `Approx`. This finishes the proof of the lemma. □

The following lemma is straightforward from the definition of Protocol `Approx`.

Lemma 2. *Let T be a tree and $q > q' \geq 1$. Suppose that `Approx`(T, q') returns $\bar{k}$ during an execution when T is rooted in r. Then, when $k = \bar{k}$ and T is rooted in r, the two labelings obtained by Lines 5-21 of `Approx`(T, q) and `Approx`(T, q') are identical.*

5.3 Main Theorem.

We can now state and prove the main theorem of this paper.

Theorem 8. *For any tree T and any integer $q > 0$, `Approx`(T, q) computes $rs_q(T)$ and a corresponding restricted q-limited search strategy in time polynomial in $|V(T)|$ (independent of q).*

The rest of this section is devoted to the proof of the above theorem. Let $\bar{k}$ be the integer returned by `Approx`(T, q). To prove the theorem, we show that both the inequalities $rs_q(T) \leq \bar{k}$ and $rs_q(T) \geq \bar{k}$ hold.

Proof of the inequality $rs_q(T) \leq \bar{k}$. First note that if $rs_q(T) \in \{1, 2\}$ (Lines 1-2), the result is valid by Theorem 7. Let us assume that $\texttt{Approx}(T)$ returns an integer $\bar{k} \geq 3$, and let $r \in V(T)$ be the root during the iteration that returns $\bar{k}$. We prove that the resulted labeling gives rise to a $\bar{k}$-good decomposition of T. Hence, the inequality $rs_q(T) \leq \bar{k}$ follows from Proposition 3. To prove this, first note that, by Line 22 of $\texttt{Approx}(T)$, all vertices have received a label $\leq q$. Let $\{\mathcal{F}_0, \cdots, \mathcal{F}_q\}$ be the collection of families of subtrees of T defined in Section 5.1. We show that both the properties (I) and (II) in Definition 1 are satisfied.

Consider the base case $\ell = q$. Recall that $\mathcal{F}_q = \{T\}$. For the sake of a contradiction, suppose that more than $\bar{k}$ vertices are labeled with q. Consider the last execution of the *while-loop*, i.e., the one which assigns a label to r. If more than $\bar{k}$ vertices distinct from r are labeled with q, then at this step we have $b_q > \bar{k}$ (Line 10) and r has to be labeled with an integer $p > q$ (Line 12), which is a contradiction. Therefore, before r being assigned a label, we must have $b_q = \bar{k}$, and r will be labeled either 0 (Line 17) or $p \neq q$ (Lines 20-21). This means that exactly $\bar{k}$ vertices are labeled with q, which is again a contradiction. Therefore, (I) holds for $\ell = q$.

Let $0 \leq \ell < q$ and let $S \in \mathcal{F}_\ell$. Let $v \in V(S)$ be the vertex of S which is labeled last, i.e., S is a subtree of T_v. Note that $S = C_{X(v,\ell+1)}(v)$. There are two cases to consider:

- Either, v is labeled with an integer $\geq \ell + 1$. Then, v has to be a leaf of S. Let u be the child of v that belongs to S. If the label of u is at least $\ell + 1$, then $V(S) = \{u, v\}$ and (I) holds.

  So let us assume that u is labeled with an integer $\leq \ell$. If $\ell = 0$, Line 16 of `Approx` (when labeling u) ensures that (II) is valid. Thus, we may assume that u is labeled with a strictly positive integer.

  For the sake of a contradiction, let us assume that at least $\bar{k}$ vertices of $S \setminus \{v\} = C_{X(u,\ell+1)}(u)$ have a label at least ℓ. If at least $\bar{k}$ vertices of $S \setminus \{v, u\}$ were labeled with an integer $\geq \ell$, or if $|X(u, \ell)| = \bar{k} - 1$, then u would have been assigned a label $\geq \ell + 1$ by Lines 12 or 15, which is not the case. This shows that $(\bar{k} - 1)$ vertices of $S \setminus \{v, u\}$ have a label $\geq \ell$ and the label of u has to be at least ℓ, i.e., exactly ℓ by the previous discussion. However, in this case, Lines 20-21 label u (which is not the root) with $m \neq \ell$, which is a contradiction.

- Or, v is labeled with an integer $\leq \ell$. By the definition of $S \in \mathcal{F}_\ell$, v is the root of T. Let b_ℓ be the number of vertices labeled at least ℓ in $S \setminus \{v\} = C_{X(v,\ell+1)}(v) \setminus \{v\}$ before labeling v. If $b_\ell > k$, then v is labeled at least $\ell + 1$ by Line 12 which is not the case. Thus, $b_\ell \leq \bar{k}$. If $b_\ell < \bar{k}$, then obviously the number of vertices of label $\geq \ell$ in S is at most $\bar{k}$. If $b_\ell = \bar{k}$, then by Line 21, v receives a label $m \neq \ell$, and by our assumption $m \leq \ell$. This shows again that at most $\bar{k}$ vertices are labeled with an integer $\geq \ell$ in S.

We have proved that $\{\mathcal{F}_0, \cdots, \mathcal{F}_q\}$ is a $\bar{k}$-good decomposition, and by Proposition 3, $rs_q(T) \leq \bar{k}$. □

Proof of the inequality $rs_q(T) \geq \bar{k}$. We now prove that the converse inequality holds. Let $\mathcal{S}$ be a monotone q-limited search strategy for T that uses κ searchers. We prove below that Protocol $\texttt{Approx}(T, q)$ returns at most κ. We do this by showing that for the value of $k = \kappa$, and for some vertex r of T (that we will designate below), the labeling procedure described in Lines 5-21 of Protocol $\texttt{Approx}(T, q)$ produces a labeling which does not contain any vertex of label $> q$. This shows that the integer $\bar{k}$ returned by $\texttt{Approx}(T, q)$ is at most κ, which implies the desired inequality.

To do so, since we are going to proceed by induction, we need to consider a more general version of restricted graph searching (and Protocol $\texttt{Approx}(T, q)$), in which we impose that some leaves

of the tree are occupied by a searcher when the first query is performed. More precisely, let $\mathfrak{L}$ be a subset of vertices of degree one in T. Let $rs_q\{\mathfrak{L}\}(T)$ be the minimum number of searchers needed to restricted monotonously search a tree T with the extra constraint that $\mathfrak{L}$ is a subset of the set of occupied vertices when the first query is performed; we say that the strategy starts from $\mathfrak{L}$.
Protocol $\texttt{Approx}(T, q, \mathfrak{L})$ is defined by replacing Line 6 of $\texttt{Approx}(T, q)$ by the following:

"For any leaf v of T (different from r), if $v \in \mathfrak{L}$, then label v with q, and otherwise, label v with 0".

Note that $rs_q\{\emptyset\}(T) = rs_q(T)$, and that $\texttt{Approx}(T, q, \emptyset)$ is identical to $\texttt{Approx}(T, q)$.

In this more general setting, assume again that $\bar{k}$ is the result of Protocol $\texttt{Approx}(T, q, \mathfrak{L})$. We will prove below that $\bar{k} \leq rs_q\{\mathfrak{L}\}(T)$.

Let $q \geq 1$. Let $\mathfrak{L}$ be a subset of leaves of T. Let $\mathcal{S}$ be a monotone restricted q-limited search strategy for T that uses $\kappa > 2$ searchers and starts from $\mathfrak{L}$. Suppose that $I \subseteq V(T)$ is the non-empty subset of vertices such that $\mathcal{S}$ first places $|I| \leq \kappa$ searchers on vertices of I and then performs the first query. Note that, by definition, we have $\mathfrak{L} \subseteq I$. Obviously, we can assume that $\mathfrak{L}$ is the set of all vertices of degree one in I, since otherwise, there is no need to place a searcher on a leaf which does not belong to $\mathfrak{L}$ (since the first query does not provide any new information depending on whether that leaf belongs to I or not).

Fix a vertex $r \in I$. Let us consider the labeling of the vertices of T produced by Lines 5-21 of Protocol $\texttt{Approx}(T, q, \mathfrak{L})$ for the value of $k = \kappa$ and when T is rooted in r. For any $v \in V(T)$, define $j_v = |I \cap V(T_v)|$

We prove by induction on $q \geq 1$ that the following claim holds.

Claim 1. *For any $v \in V(T)$, there are at most j_v vertices labeled with q in T_v, and no vertex of T_v is labeled with an integer strictly larger than q.*

Clearly, once the claim has been proved, we infer that Protocol $\texttt{Approx}(T, q, \mathfrak{L})$ returns $\bar{k} \leq \kappa$, which concludes the proof of the theorem.

To show the above claim in the base case of our induction, $q = 1$, let us define Protocol $\texttt{OneQuery}(T, \mathfrak{L})$, $\mathfrak{L}$ being any subset of leaves of T, as the algorithm obtained from Protocol $\texttt{OneQuery}(T)$ by replacing Line 6 by

"For any leaf v (different from r) of T, if $v \in \mathfrak{L}$, then label v with 1, and label v with 0 otherwise".

A direct generalization of the proof of Lemma 2 allows to show that $\texttt{OneQuery}(T, \mathfrak{L})$ achieves the same result as $\texttt{Approx}(T, 1, \mathfrak{L})$. Moreover, a direct generalization of the proof of Theorem 6 proves that the execution of $\texttt{OneQuery}(T, \mathfrak{L})$ returns at most κ (in particular, as the proof of that theorem shows, when T is rooted in r). Hence, the proposition holds for $q = 1$.

By induction, let us assume that Claim 1 holds for any $1 \leq q' < q$. We proceed by a second induction on the value of j_v and show that it also holds for q. Let v be a vertex of T.

In the base case of our (second) induction, $j_v = 0$ (hence, $v \neq r$). In this case, one can easily derive a restricted $(q-1)$-limited strategy $\mathcal{S}_0$ from $\mathcal{S}$ that clears $\hat{T}_v = T_v \cup \{p(v)\}$, where $p(v)$ is the parent of v, and uses at most κ searchers. In addition, in $\mathcal{S}_0$ the first steps consist in placing searchers on a subset I_0, $p(v) \in I_0$, and then performing a query. The subset $I_0 \setminus \{p(v)\}$ is precisely the set of all occupied vertices in T_v according to the strategy $\mathcal{S}$ when the second query is performed, in the case when T_v is still contaminated after the first query (note that this can happen precisely because $I \cap T_v = \emptyset$). This in particular shows that $rs_{q-1}(\hat{T}_v) \leq \kappa$. By our

induction hypothesis on q, the execution of Lines 5-21 in $\texttt{Approx}(\hat{T}_v, q-1, \emptyset)$ when $\hat{T}_v$ is rooted in $p(v)$ and $k = \kappa$, labels any vertex w of $\hat{T}_v$ in such a way that at most $j_w^{q-1} = |I_0 \cap V(T_w)|$ vertices of T_w receive label $q-1$, and no vertex of $\hat{T}_v$ is labeled $> q-1$. By the observation made in Lemma 2, the execution of (Lines 5-21) in $\texttt{Approx}(\hat{T}_v, q, \emptyset)$ when $\hat{T}_v$ is rooted in $p(v)$ and $k = \kappa$ labels the vertices of $\hat{T}_v$ similarly. Finally, since $\mathfrak{L} \cap V(T_v) = I \cap V(T_v) = \emptyset$, obviously the execution of $\texttt{Approx}(T, q, \mathfrak{L})$ labels the vertices of T_v similarly. (Notice that, $p(v)$ may be labeled differently but this is not a concern here). This shows that no vertex of T_v is labeled with an integer $\geq q$, and so the claim holds for v.

Suppose now that $j_v > 0$, and assume that the induction hypothesis holds for any $j_u < j_v$, for any vertex u of T. We divide the proof into two parts depending on whether $v \in I$ or not.

Case $v \in I$.

If v is a leaf (different from the root), then $v \in \mathfrak{L}$ and the claim holds for v by the definition of $\texttt{Approx}(T, q, \mathfrak{L})$. Suppose that v is not a leaf. We show the result by applying the induction hypothesis on the children of v. Indeed, for any child u of v, since $v \in I$, we have $j_u < j_v$ and so the induction hypothesis applies. In other words, in T_u at most $j_u = |I \cap T_u|$ vertices are labeled q and no vertex is labeled $> q$. This shows that no vertex is labeled $> q$ in $T_v \setminus \{v\}$, and there are at most $\sum_u j_u = |I \cap T_v| - 1$ vertices labeled q in $T_v \setminus \{v\}$, where the sum is over all the children of v. Therefore, to show that Claim 1 holds for v we only need to show that v is not assigned a label $> q$. Consider the time when the labeling of v happens. Note that the variable b_q is at most $|I \cap T_v| - 1$, which (by our assumption $r \in I$) is at most $\kappa - 2$ if v is not the root, and at most $\kappa - 1$ if v is the root. This shows that v cannot be labeled by an integer strictly more than q, and Claim 1 follows.

Case $v \notin I$.

Let $X = X(v, I) = \{v_1, \cdots, v_\ell\}$ ($\ell \geq 1$) be the set of descendants u of v that are in I and such that there is no other vertex of I on the path between v and u (notations are the same as in Section 3.2). Note that for any $1 \leq i \leq \ell$ and any child z of v_i, we have $j_z < j_v$ and thus the induction hypothesis holds for z. That is, for any $1 \leq i \leq \ell$ and any child z of v_i, there are at most j_z vertices labeled q in T_z and no one is labeled $> q$.

We claim that any vertex labeled with $\geq q$ in $C_X(v)$ must belong to a path between v_i and v for some i, $1 \leq i \leq \ell$. For the sake of a contradiction, suppose this is not the case and consider a vertex $w \in C_X(v)$ that does not belong to any path between v and v_i for any $1 \leq i \leq \ell$. Note that $j_w = 0$. A similar reasoning as the case $j_v = 0$ above shows that all vertices in T_w are labeled at most $q-1$ (and the number of vertices labeled $q-1$ in T_w is at most the number j_w^{q-1} of occupied vertices in T_w when the second query is performed according to $\mathcal{S}$, and when the fugitive remains in this component after the first query), which is a contradiction.
Consider now for any $i \leq \ell$, the vertex (if any) u_i of label $\geq q$ which lies on the unique path between v_i and v, including the vertices v_i and v, which is the closest to v_i. Let U be the set of all the vertices u_i; obviously, $|U| \leq \ell = |X|$.

We claim that there is no other vertex in $C_X(v)$ which is labeled $\geq q$. For the sake of a contradiction, let us assume that a vertex $w \in C_X(v) \setminus U$ is labeled $\geq q$ by Protocol $\texttt{Approx}(T, q, \mathfrak{L})$. Let $U' = X(U, w)$, the set of descendants u of w that are in U with the property that there is no other vertex of U on the unique path between w and u in T (cf. Section 3.2 for notations). By the definition of U, $\hat{C}_{U'}(w)$ is a subtree of $\hat{C}_X(v)$ and U' is a subset of leaves of $\hat{C}_{U'}(w)$. One can easily derive from $\mathcal{S}$ a restricted $(q-1)$-limited strategy for clearing $\hat{C}_{U'}(w)$ starting from $\{p(w)\} \cup U'$ and using at most κ searchers. Hence, by the induction hypothesis on q, the execution of Lines 5-21 in $\texttt{Approx}(\hat{C}_{U'}(w), q-1, U')$ when $\hat{C}_{U'}(w)$ is rooted in $p(w)$ and $k = \kappa$

gives a label to w which is at most $q-1$. It is straightforward to verify that the execution of Lines 5-21 of $\texttt{Approx}(\hat{C}_{U'}(w), q-1, U')$ when $k=\kappa$ and the root is $p(w)$ provides the same labeling for the vertices of $C_{U'}(w) \setminus U'$ as the labeling obtained by Lines 5-21 of $\texttt{Approx}(T, q, \mathfrak{L})$ for $k=\kappa$ and the root r. This shows that w is labeled $< q$, which is a contradiction. Thus, the number of vertices in $C_X(v)$ of label $\geq q$ is exactly $|U|$.

We now prove that no vertex in U is labeled $> q$. The proof goes by induction, by starting from all $u \in U$ with no descendants in U and going towards the root. So let $u \in U$ be a vertex such that we have already proved that all the vertices in $(U \cap V(T_u)) \setminus \{u\}$ are labeled q. First note that since $v \notin I$, v is not the root and thus $j_v < \kappa$. This shows that at most $\kappa - 2$ vertices of $V(T_u) \setminus \{u\}$ are labeled with q, and no one is labeled $> q$ (either by induction or by the definition of U in the base case when u has no descendant in U). By the definition of the labeling procedure in `Approx`, u cannot be assigned a label $> q$, and we are done.

To conclude, note that by the hypothesis of our (second) induction (on j_v), the number of vertices labeled q in T_v is at most $|U| + \sum_z j_z \leq |X(v,I)| + \sum_z |I \cap V(T_z)| = |I \cap V(T_v)| = j_v$, where the sums are over the children z of v_i for any $1 \leq i \leq \ell$. □

Proof of the time complexity of Theorem 8. The time complexity of the *while-loop* of Protocol `Approx` is dominated by the execution of Line 14. By Theorem 5, `InitPos` performs the computation of $s_0\{X(v,I) \cup \{p(v)\}\}(\hat{C}_{X(v,I)}(v))$ in time polynomial in $n = |V(T)|$. Thus, `Approx` performs in time $O(n^3 t(n))$, where $t(n)$ is the time complexity of `InitPos`. Hence, Protocol `Approx` performs in polynomial time in the size of the input tree (independent of q). □

6 Search a tree with two searchers

This section is devoted to the design and the proof of Algorithm `TwoSearchers` used in the previous sections.

When using two searchers and no query, only caterpillars can be cleared. On the other hand, if an arbitrary number of queries can be performed, two searchers can clear any tree. In this section, we design an algorithm that computes the smallest number of queries required to clear a tree by using two searchers. Note that this was used in Section 4 as a way to check if $s_1(T) = 2$.

The proposed algorithm, called `TwoSearchers`, (formally described in Algorithm 4), computes $q_2(T)$, the smallest number of queries required to clear any tree T with two searchers. Since the treewidth of a tree is equal to one, this result provides information on the minimum number of queries in any search strategy that only uses two searchers. Translating the result to the tree-decomposition point of view, this provides information on the minimum of the maximum number of branching nodes that lie on a path from the root to a leaf of $\mathfrak{T}$ in any tree-decomposition $(\mathfrak{T}, \mathcal{X})$ of T of width one.

By Corollary 1, we may assume that after the first step, a monotone strategy using two searchers always places the free searcher on a contaminated neighbor of the vertex already occupied.

Roughly speaking, Protocol `TwoSearchers` consists in iteratively labeling the vertices of the tree T, from leaves toward an arbitrary root. The label of each vertex v consists of a pair of non-negative integers $(q(v), c(v))$. The significance of $q(v)$ and $c(v)$ are as follows. We will prove that $q(v) \geq 0$ is the smallest number of questions required for two searchers to clear the subtree T_v starting at v. The value of $c(v) \in \{0,1,2\}$ is a technical index which describes the first steps of a search strategy which clears T_v by starting from v and by using two searchers and $q(v)$ queries. The significance of $c(v) = 2$ is that in the search strategy which clears T_v and starts from v, there is no need to perform a query when v is occupied. In other words, v can become free of searchers when the first query in this strategy is performed. If $c(v) = 1$, this means that,

Algorithm 4 Protocol `TwoSearchers`(T) that returns $q_2(T)$

Require: A tree T

1: $q \leftarrow |V(T)|$
2: **for all** $r \in V(T)$ with degree at least 2 **do**
3: Let T be rooted in r
4: Label all leaves with $(0, 0)$
5: **while** it remains an unlabeled vertex $v \in V(T)$ **do**
6: Let v be an unlabeled vertex every child of which has a label
7: Let (ℓ, c) be the greatest label (in lexicographical ordering) of the children of v
8: **if** v has at most one non-leaf child **then**
9: *// if v has one non-leaf child w then $(\ell, c) = (q(w), c(w))$*
10: Label v with $(\ell, 2)$
11: **else if** v has at least two non-leaf children and exactly one of them is labeled (ℓ, c) and $c = 0$ **then**
12: Label v with $(\ell, 1)$
13: **else**
14: Label v with $(\ell + 1, 0)$
15: Let (q_r, c_r) be the label of the root
16: **if** $q_r < q$ **then**
17: $q \leftarrow q_r$
18: **return** q

in the search strategy that clears T_v by starting from v and by using $q(v)$ queries, there is exactly one child of v where we have to place the second searcher when the first query is performed. And finally, $c(v) = 0$ in the case $q(v) > 0$ means that the first query in the search strategy for clearing T_v is performed when a searcher is at v. That is, placing the second searcher before the first query does not help in decreasing the number of queries, i.e., whatever be the child w of v where the second searcher is placed, there is a contaminated component of $T \setminus \{v\}$ not containing w that will require $q(v)-1$ queries. More formally, to describe the significance of $q(v)$ and $c(v)$, consider a step of the labeling procedure when a searcher occupies a vertex $v \in V(T)$ and the fugitive stands at some vertex in T_v. Three cases might happen depending on the shape of T_v and on the labels of the descendants of v. We suppose that v is not a leaf (otherwise, $(q(v), c(v)) = (0, 0)$).

Case 1: v has at most one non-leaf child.

If v has only leaf-children, then T_v (which is a star with center v) can be cleared without performing any query. In that case, $(q(v), c(v)) = (0, 2)$. Or there exists a vertex $u \in V(T_v)$ such that $C_{\{u\}}(v)$ is a caterpillar with ends u and v (Line 8 of Protocol `TwoSearchers`). In this case, two searchers can easily clear the whole caterpillar $C_{\{u\}}(v)$ by starting from v and finishing at u, and without performing any query. (The subtree T_u will then be cleared according to the search strategy for T_u which starts from u and uses $q(u)$ queries.) This means $q(v) = q(u)$ and $c(v) = 2$ (Line 10).

Case 2: v has (at least two) non-leaf children $v_1, \cdots, v_d$ such that $q(v_1) \geq \cdots \geq q(v_d)$ and $q(v_1) > q(v_2)$ and $c(v_1) = 0$.

In this case (Line 11), exactly $q(v_1)$ queries are required to search T_v using two searchers and starting from v (Line 12). The fact that at least $q(v_1)$ queries are required is obvious, since already T_{v_1} requires this number of queries. To see the equality, note that the strategy

can proceed by placing the second searcher on v_1 and by performing the first query. Two situations can happen: Either, the contaminated part S is the one which is adjacent to v_1, or, the contaminated part is a tree T_u for u a child of v different from v_1.

In the first situation, we can proceed according to the search strategy which clears the subtree T_{v_1}. Indeed, since $c(v_1) = 0$, T_{v_1} can be cleared with $q(v_1)$ queries by starting from v_1 and by performing a query without placing the second searcher. Thus, the situation is the same as if the subtree T_{v_1} was being cleared. In other words, given that the searcher at v is now free, the remaining steps in the search strategy for T_{v_1} can be performed by performing at most $q(v_1) - 1$ queries.

In the second situation, the searcher that occupies v_1 can be removed and be placed on u. Then the searcher at v becomes free. We then follow the search strategy for clearing T_u using $q(u) \leq q(v_1) - 1$ queries.

Case 3: v has (at least two) non-leaf children $v_1, \cdots, v_d$ such that $q(v_1) \geq \cdots \geq q(v_d)$ and either $q(v_1) = q(v_2)$ or $c(v_1) > 0$.

Then we show that $q(v) = q(v_1) + 1$ (Line 14) and that placing the second searcher when the first query is performed does not help (i.e. $c(v) = 0$). In other words, the search strategy starts directly by performing the first query. Note that since the search strategy has to start from v, a searcher has to be placed on v, and in addition a query has to be performed.

If $q(v_1) = q(v_2)$, it does not matter to place the second searcher when the first query is performed. To see this, simply imagine the situation where the second searcher is placed on some vertex which does not belong to T_{v_1} (resp. T_{v_2}) and the fugitive stands somewhere in T_{v_1} (resp. T_{v_2}).

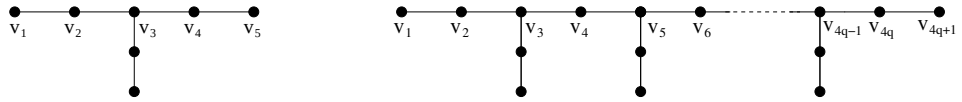
And if $c(v_1) > 0$, by the previous case, this means that in the search strategy that clears T_{v_1} by starting from v_1 and by performing at most $q(v_1)$ queries, one is required to first place the two searchers on v_1 and one of its children w before performing the first query. Since in the search strategy for T_v, v must be occupied at the beginning, w cannot be occupied when the first query is performed. This shows that placing the second searcher does not allow to gain on the number of queries, and one will still need $q(v_1)$ extra queries to clear the contaminated part in the case the contaminated part is in the subtree T_{v_1}. In other words, we can assume that the first query in the search strategy for clearing T_v is performed directly after placing a searcher on v, i.e., $c(v) = 0$. Obviously, the number of required queries is $q(v) = q(v_1) + 1$.

We are now in position to prove the following theorem.

Theorem 9. *For any n-node tree T, Protocol `TwoSearchers`(T) computes $q_2(T)$ in time $O(n^2)$.*

Proof. Consider an execution of the *for-loop* (Line 2) and suppose that T is rooted at some vertex v of the tree, and each vertex has been assigned a label by Protocol `TwoSearchers`. Let (q, c) be the label of the root v. It is easy to see that (q, c) is the greatest label (for the lexicographical order) of any vertex of T. By the discussion that preceded the theorem, q queries are enough to clear T starting from the root. Therefore, $q_2(T) \leq q$.

Let $\mathcal{S}$ be a monotone strategy that clears T with two searchers and at most $q \geq 0$ queries. Let us assume that the first step of $\mathcal{S}$ consists in placing a searcher on a vertex $r \in V(T)$. We prove by induction on $q \geq 0$ that there exists two non-negative integers $c \in \{0, 1, 2\}$ and $q' \leq q$ such that if T is rooted in r, then r will be eventually labeled (q', c) by the corresponding execution



(a) T_1, $q_2(T_1) = 1$ (b) T_q, $q_2(T_q) = \lceil |V(T_q)|/8 \rceil$

Figure 5: Tree T_q, $q \geq 1$, with $8q - 1$ nodes such that $q_2(T_q) = q = \lceil |V(T_q)|/8 \rceil$.

of the *for-loop* of Protocol `TwoSearchers` (when the root of the tree is r). This obviously proves the correctness of the algorithm.

For the base of our induction, let $q = 0$. Then T has to be a caterpillar and one of its end-points must be r, and obviously the claim holds in this case.

Let us assume that $q \geq 1$ and for all $q' < q$ the induction hypothesis holds. By Proposition 1 and Corollary 1, we may assume that after the first step where a searcher is placed on r, either the first query is performed, or the second searcher is placed on a child v of r and then the first query is performed. If T is a caterpillar, the theorem clearly holds, therefore we can assume that T is not a caterpillar and that none of v and r is a leaf. Indeed, otherwise, we can slightly modify $\mathcal{S}$ without increasing the number of queries such that in the new rooted tree and search strategy this condition holds. For any child $w \neq v$ of v or r, there is a strategy $\mathcal{S}_w$ (given by $\mathcal{S}$) that clears T_w with two searchers by starting to place a searcher on w and performing at most $q - 1$ queries. Thanks to the induction hypothesis, w is eventually labeled (q'', c') by Protocol `TwoSearchers`, with $q'' \leq q - 1$. Thus, r will eventually be labeled (q', c) by Protocol `TwoSearchers` with $q' \leq q$.

A simple analysis of the algorithm gives the running time $O(n^2)$ as stated in the theorem. □

7 On the number of non-deterministic steps

In this final section, independent from the rest of the paper, we provide some tight upper bounds on the number of queries required to clear any tree. Our results show that there is a gap on the number of queries required to clear a tree when the number of searchers passes from two to three. More precisely, we prove that $\Omega(n)$ queries might be necessary to clear a tree on n nodes using two searchers, whereas $O(\log n)$ queries are sufficient to clear any tree on n nodes using three searchers. For any tree T and $k \geq 2$, let $q_k(T)$ be the smallest number of queries required to clear T using at most k searchers.

It is well known that in any tree T on n nodes, there exists a vertex v, called a *centroid* of T, such that each of the connected components of $T \setminus \{v\}$ has size at most $\lfloor \frac{n}{2} \rfloor$. To see this, for the sake of a contradiction, suppose there is no such vertex. Then for any vertex v, there exists a connected component of $T \setminus v$ with size strictly larger than $\lfloor \frac{n}{2} \rfloor$. Let $e = \{v, u\}$ be the edge which connected v to this component, and orient e towards u ($v \to u$). In this way, we obtain a total of n orientations on the edges. Since the number of edges is $n - 1$, there exists an edge $e = \{u, v\}$ which is oriented both from u to v ($u \to v$) and from v to u ($v \to u$). The connected component of $T \setminus \{v\}$ which contains u does not intersect the connected component of $T \setminus \{u\}$ which contains v. This is certainly a contradiction since the total size of the tree will be strictly larger than n.

Theorem 10. *For any tree T on n nodes, $q_2(T) \leq \lceil n/8 \rceil$, and this bound is tight.*

Proof. First, we prove that the bound is achieved. Consider the family of trees T_q, $q \geq 1$ depicted in Figure 5. T_1 is the tree on seven nodes with a central path P_1 of length five and a pending path of length two from v_3. For $q \geq 2$, T_q is obtained from the central path P_q on vertices

$\{v_1, v_2, , \cdots, v_{4q+1}\}$ by attaching a path $\{u_{2i+1}, w_{2i+1}\}$ of length two at every node v_{2i+1} for any $1 \leq i \leq 2q-1$, i.e., u_{2i+1} is adjacent to v_{2i+1} (see Figure 5). Note that for $q \geq 1$, the number of vertices of T_q is $n_q = |V(T_q)| = 8q-1$. We show below that for any $q \geq 1$, $q_2(T_q) = q = \lceil n_q/8 \rceil$.

For $q = 1$, it is easy to verify the result. Indeed, given that the pathwidth of T_1 is two, performing a query is necessary for clearing T_1 with two searchers, and moreover, it is easy to check that one query is sufficient. So we assume in the following that $q \geq 2$. Let $\mathcal{S}$ be a monotone strategy that clears T_q using two searchers. According to Proposition 1 and Corollary 1, we may assume that the first steps in $\mathcal{S}$ consists of placing the searchers on two adjacent vertices. If the searchers occupy $\{u_{2i+1}, w_{2i+1}\}$ for some $1 \leq i \leq 2q-1$, the searcher at w_{2i+1} is then removed and placed on v_{2i+1}, then the searcher at u_{2i+1} must be removed and placed at v_{2i} or v_{2i+2}. Otherwise, the first step places a searcher on a node v_j, $1 \leq j \leq 4q+1$, and the second searcher is placed on one neighbor x of v_j. By symmetry, we can assume that $2q \leq j$. If $\{v_j, x\} = \{v_{4q}, v_{4q+1}\}$, the only possible first moves are to remove the searcher on v_{4q+1}, place it at v_{4q-1} and then remove the searcher at v_j and place it on a contaminated neighbor y of v_{4j-1}. Therefore, in all cases, after at most 6 moves, a searcher occupies a vertex v_j, $2q \leq j \leq 4q-1$, and the second searcher occupies a neighbor x of it, and at most 2 unoccupied nodes are cleared. We moreover assume that v_{j-1} is not occupied and contaminated (it is always possible, possibly by exchanging x with v_j). Finally, let $w = w_j$ if $j \equiv 1[2]$ and $w = w_{j+1}$ otherwise.

We now show that one query must be performed for v_{j-1} (and maybe v_{j-2}) to be occupied and cleared. Indeed, v_j and x are occupied, v_1 and w (or possibly v_{4q+1} before the first query) are contaminated. Therefore, both searchers are adjacent to a contaminated node and cannot be removed. After the query, we may assume that the contaminated component is the one containing v_1. Hence, the only possible strategy consists of removing the searcher from x, placing it at v_{j-1}, then removing the searcher at v_j and placing it on a neighbor of v_{j-1}. Therefore, we reach the same situation as before for $j' \geq j-2$. Since, initially, $j \geq 2q$, this shows that at least q queries must be performed to clear v_1. Hence, $q_2(T_q) \geq q$.

Let us now consider a strategy $\mathcal{S}$ that starts placing a searcher on v_{2q+1} and then performs a query. We may assume that one searcher occupies v_{2i+1}, $1 \leq i \leq q$, and that the contaminated component is the one containing v_1 (this may be assumed by symmetry). Then $\mathcal{S}$ performs a query. If the fugitive occupies u_{2i+1} or w_{2i+1} it will be captured without additional query. Otherwise, the free searcher is placed at v_{2i}, and after the searcher at v_{2i+1} is removed and placed on v_{2i-1}, and then a query is performed. Following such a strategy, q queries are sufficient to clear T_q. This shows that $q_2(T_q) = q$.

Roughly speaking, the above strategy consisted in choosing the initial position of searchers in a clever way so that after the first query is performed, the size of the contaminated part decreases by at least half the size of the tree. Then the sequence of steps in the strategy is carried on in such a way that each time a query is performed, the size of the contaminated part drops off by at least four. Obviously, if there is a way to follow a similar strategy in an arbitrary tree, then the upper bound given in the theorem has to hold. This is what we show now.

Now, perform the following strategy. Let r be a centroid of the tree and suppose that T is rooted at r. First place a searcher on r and perform a query. The result will be a connected component of $T \setminus \{r\}$, with size at most $n/2$, which designates the contaminated part. Then place the second searcher on the unique vertex v in this component which is adjacent to r. Now, consider a step when one searcher is at node v and the contaminated part is T_v. The second searcher may be free or occupy any node in $T \setminus T_v$. We will show that each query can decrease the size of the contaminated component by at least four. More precisely, after the next query, one searcher will occupy some node $v' \in V(T_v)$, the contaminated part will be $T_{v'}$ and $|V(T_{v'})| \leq |V(T_v)| - 4$. The following situations can happen (we use the terminology of

Section 3.2).

1. There is at most one child v' of v such that $|V(T_{v'})| \geq 2$. In this case, the free searcher can be used to clear all the leaves adjacent to v. Once this has been done, the free searcher can be placed on v', and v' can now play the role of v. In this way, the size of the contaminated part has been decreased by at least one, without performing any query.

2. There are at least three children of v_1, v_2, v_3 in T_v such that $|V(T_{v_i})| \geq 2$, for each $i \in \{1, 2, 3\}$.

   In this case, a query is performed. Let v' be the child of v such that $T_{v'}$ is the designated component by the query. Then the free searcher can be placed on v' and the searcher at v can be removed, so that the role of v is now played by v'. Obviously, in this case, the size of the contaminated part has been decreased by at least four.

3. v has exactly two children v_1 and v_2 such that $|V(T_{v_i})| \geq 2$. The following three situations can happen.

   (a) For the two children v_1, v_2 of T_v as above, we have $|V(T_{v_i})| \geq 3$.
   This case is exactly similar to case (2). (Note that in this case the size of the contaminated part decreases by at least four since one of the two T_{v_i} will be declared clear and the vertex v' in the contaminated part adjacent to v will be also cleared by the free searcher.)

   (b) One child v_1 of v has $|V(T_{v_1})| \geq 3$ and the other one v_2 has $|V(T_{v_2})| = 2$.
   In this case, the free searcher is placed on v_1 and a query is performed. If the contaminated part is T_{v_2}, this is the last query and the rest can be cleared without performing any query. Obviously, the size of the contaminated part decreases by at least four. Otherwise, the contaminated part is one of the components adjacent to v_1. Let v' be the vertex adjacent to v_1 in this component. The searcher at v can be removed and placed on v'. The size of the contaminated part decreases again by at least four (T_{v_2}, v_1 and v' are clear now), and v' plays the role of v from now on.

   (c) In the last case, we have $|V(T_{v_1})| = |V(T_{v_2})| = 2$.
   In this case, a query is performed. Since all the other children of v are leaves, this is the last query and the rest can be cleared without performing any query. Obviously, the size of the contaminated part has been dropped off by at least four again.

This finishes the proof of our Theorem 10. □

Theorem 11. *For any tree T on n nodes, and for any $k \geq 3$, $q_k(T) \leq 2\lceil \log_2 n \rceil$. Moreover, for any fixed k the bound is asymptotically tight: for any n_0, there exists $n \geq n_0$ and a tree T_n on n nodes such that $q_k(T_n) = \Omega(\log_2 n)$.*

Proof. To prove the upper bound, obviously, it will be enough to show that $q_3(T) \leq 2\lceil \log_2 n \rceil$ for any tree T (since for any $k \geq 3$, $q_k(T) \leq q_3(T)$). Consider the following strategy. The first step of the strategy consists in placing a searcher on the centroid r of T and performing a query. The result will be a connected component of $T \setminus \{r\}$, the contaminated part, with size at most $\lfloor |V(T)|/2 \rfloor$. The strategy proceeds inductively in such a way that the size of the contaminated component shrinks to at most $\lfloor |V(T)|/2^q \rfloor$ by performing a number of at most $q' \leq 2q$ queries. To show this, consider a step of the procedure where $q' \leq 2q$ queries are already performed and the size of the contaminated part is at most $\lfloor |V(T)|/2^q \rfloor$. We show how to proceed so that at most $q'' \leq 2q + 2$ queries will be performed and the size of the contaminated part will be shrunk to at most $\lfloor |V(T)|/2^{q+1} \rfloor$. There are two cases to be considered depending on whether one or two searchers are adjacent to the contaminated part after performing the q'-th query.

Case 1. Consider first the case where there is a unique searcher occupying a vertex adjacent to the contaminated part, that is there are two free searchers. Note in particular that this is the case after the first step. A free searcher is then placed on the centroid of the contaminated subtree and a query is performed. After such a step, the size of the contaminated part has been divided by 2, the number of performed queries q" is equal to $q' + 1$, and at least one searcher is still free (since in this case, two searchers were initially free). This shows the inductive claim in this case.

Case 2. Otherwise, assume that two searchers are occupying vertices u and v that are adjacent to the contaminated part and the third searcher is free. Let c be the centroid of the contaminated part and let w be the vertex separating u, v and c, i.e., either $w = c$, or u, v and c are in distinct components of $T \setminus \{w\}$. Then, the free searcher is placed on w and a query is performed.

Note that after the query, there must be at least one free searcher. Moreover, if the size of the connected component has not been divided by two after the query, it means that the contaminated part contains c (in particular $w \neq c$). Now proceed as in Case 1: place a searcher on the centroid of the contaminated part and perform a query. The number of performed queries is $q'' = q' + 2 \leq 2q + 2$ and the size of the contaminated component is at most $\lfloor |V(T)|/2^{q+1} \rfloor$. This shows that the inductive claim also holds in this case.

Thus, we have proved that $q_3(T) \leq 2\lceil \log_2 n \rceil$.

To prove the tightness, let $k \geq 3$ be a fixed integer. For any integer $h \geq 1$, let R^h be the rooted 3-regular tree of depth h: all the internal (non-leaf) nodes have three children and any leaf is at distance h from the root. Obviously, $|V(R^h)| = (3^{h+1} - 1)/2$. It is well known that, for any $k \geq 2$, $s_0(R^{k-1}) = k$, and moreover, there is a strategy with k searchers starting from the root [16, 15]. Let T_k^1 be the rooted tree obtained from $k+1$ copies of R^{k-1} by adding a new vertex r_1, the root of T_k^1, that is adjacent to the root of each copy of R^{k-1}. Recursively, for any $q \geq 2$, let T_k^q be the tree rooted in r_q obtained from $k+1$ copies of T_k^{q-1} by adding a new vertex r_q that is adjacent to the root of each copy of T_k^{q-1}.

A simple calculation shows that for any $q \geq 1$, $|V(T_k^q)| = \frac{k^{q+1}-1}{k-1} + \frac{3}{2}k^q(3^{k-1} - 1) = O(k^q 3^k)$. We will show by induction on q that $q_k(T_k^q) \geq q$. This proves the second part of the theorem: indeed, the above expression for $|V(T_k^q)|$ shows that asymptotically, for q sufficiently large, we have $q = \Omega(\log_2 |V(T_k^q)|)$, and so $q_k(T) = \Omega(\log_2 |V(T)|)$ for all the trees of the form T_k^q. By the usual trick one can exhaustively cover all the integers: simply fix a copy of T_k^q in T_k^{q+1} and remove a number of vertices from T_k^{q+1} to obtain a tree T_n of size $|V(T_k^q)| \leq n \leq |V(T_k^{q+1})|$. Obviously, $q_k(T_n) = \Omega(\log_2 n)$.

So we are left to prove the above claim. We proceed by induction on $q > 0$. Since T_k^1 admits R^k as a minor, $s_0(T_k^1) > k$. This shows that $q_k(T_k^1) \geq 1$ and so the claim holds for $q = 1$. Let $q > 2$ and assume that the result holds for $q - 1$. We show that it also holds for q. Let r_q be the root of T_k^q. Consider any strategy for clearing T_k^q that uses k searchers. By the construction of T_k^q, we know that $T_k^q \setminus \{r_q\}$ consists in $k+1$ disjoint copies of T_k^{q-1}. Therefore, at the moment when the first query is performed, at least one component of $T_k^q \setminus \{r_q\}$ contains no searchers, and can be contaminated. This component is a copy of T_k^{q-1} and so by our induction hypothesis, clearing this component needs at least $q - 1$ queries. This proves the claim. Actually, one has $q_k(T_k^q) = q$ for any $q \geq 1$ as the proof shows. □

References

[1] L. Barrière, P. Flocchini, F. V. Fomin, P. Fraigniaud, N. Nisse, N. Santoro, and D. M. Thilikos. Connected graph searching. *Inf. Comput.*, 219:1–16, 2012.

[2] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In *14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 200–209, 2002.

[3] D. Bienstock and P. Seymour. Monotonicity in graph searching. *J. Algorithms*, 12(2):239–245, 1991.

[4] H. L. Bodlaender and F. V. Fomin. Approximation of pathwidth of outerplanar graphs. *J. Algorithms*, 43(2):190–200, 2002.

[5] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[6] R. L. Breisch. An intuitive approach to speleotopology. *Southwestern Cavers*, VI(5):72–78, 1967.

[7] D. Coudert, F. Huc, and D. Mazauric. A distributed algorithm for computing the process number in trees. *Algorithmica*, 63(1-2):158–190, 2012.

[8] D. Coudert, F. Huc, and J.-S. Sereni. Pathwidth of outerplanar graphs. *J. Graph Theory*, 55(1):27–41, 2007.

[9] J. Ellis and M. Markov. Computing the vertex separation of unicyclic graphs. *Inform. and Comput.*, 192(2):123–161, 2004.

[10] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Inform. and Comput.*, 113(1):50–79, 1994.

[11] F. V. Fomin, P. Fraigniaud, and N. Nisse. Nondeterministic graph searching: From pathwidth to treewidth. *Algorithmica*, 53(3):358–373, 2009.

[12] F. V. Fomin and D. M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theor. Comput. Sci.*, 399(3):236–245, 2008.

[13] J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, 45(3):233–248, 1993.

[14] F. Mazoit and N. Nisse. Monotonicity of non-deterministic graph searching. *Theor. Comput. Sci.*, 399(3):169–178, 2008.

[15] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *J. Assoc. Comput. Mach.*, 35(1):18–44, 1988.

[16] T. D. Parsons. Pursuit-evasion in a graph. In *Theory and applications of graphs*, volume 642 of *Lecture Notes in Math.*, pages 426–441. Springer, Berlin, 1978.

[17] N. Robertson and P. Seymour. Graph minors ii, algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[18] P. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *J. Combin. Theory Ser. B*, 58:22–33, 1993.

[19] K. Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *J. Algorithms*, 47(1):40–59, 2003.