A Distributed Algorithm for Computing the Node Search Number in Trees

David Coudert, Florian Huc, Dorian Mazauric

A Distributed Algorithm for Computing the Node Search Number in Trees

David Coudert

Mascotte, INRIA, I3S, CNRS, Université de Nice Sophia
Sophia Antipolis, France
david.coudert@inria.fr

Florian Huc

EPFL, Laboratory of Distributed Programming
Station 14, Bat INR
Lausanne, Switzerland
florian.huc@epfl.ch

Dorian Mazauric

Mascotte, INRIA, I3S, CNRS, Université de Nice Sophia
Sophia Antipolis, France
dorian.mazauric@inria.fr

Abstract

We present a distributed algorithm to compute the node search number in trees. This algorithm extends the centralized algorithm proposed by Ellis *et al.* [8]. It can be executed in an asynchronous environment, requires an overall computation time of $O(n \log n)$, and n messages of $\log_3 n + 4$ bits each.

The main contribution of this work lies in the data structure proposed to design our algorithm, called *hierarchical decomposition*. This simple and flexible data structure is used for four operations: updating the node search number after addition or deletion of any tree-edges in a distributed fashion; computing it in a tree whose edges are added sequentially and in any order; computing other graph invariants such as the process number and the edge search number, by changing only initialization rules; extending our algorithms for trees and forests of unknown size (using messages of up to $2 \log_3 n + 5$ bits).

1 Introduction

Treewidth and pathwidth have been introduced by Robertson and Seymour [19] as part of the graph minor project. Those parameters are very important since many problems can be solved in polynomial time for graphs with bounded treewidth or pathwidth. By definition, the treewidth of a tree is one, but its pathwidth might be up to $\log_3 n$, where n denotes the number of nodes. Centralized algorithms have been proposed to compute the pathwidth of a tree in linear time [8, 20, 21], but so far no distributed algorithm exists.

The algorithmic counterpart of the notion of pathwidth is the cops and robber game, also known as the graph searching problem [3, 7, 10, 13, 17]. This NP-hard problem [13] consists in finding an invisible and fast fugitive in a graph using agents. More precisely, in this two-player game, the first player can move at any time the fugitive from a node to another along a path in which no node is occupied by an

agent, and at each turn, the second player can either put an agent on a node, or remove an agent from a node. The fugitive is captured when an agent is located on the same node. The node search number is the minimum number of agents required to catch the fugitive. It was proved by Ellis *et al.* [8] that, for any graph G, the node search number of G is equal to the pathwidth of G plus 1. In addition, when the fugitive is visible the cops and robber game becomes the equivalent of the treewidth, and Fomin *et al.* [9] filled the gap between treewidth and pathwidth introducing in the cops and robber game a parameter controling the number of times the fugitive is visible.

Other closely related graph invariants (e.g., process number [4–6], edge search number [15]) have been proposed, but it is not known whether they are equivalent to the pathwidth or not. In other words, it is not known if given the value of a parameter, it is possible to compute the other one in polynomial time (with a time complexity independent of the known parameter's value), unless the graph is a tree [18]. In this later article, the authors show the correspondance between the edge search number and the node search number in trees, plus they propose a linear time algorithm to compute an optimal edge search strategy given an optimal node search strategy.

Another related parameter is the connected search number, which is similar to the search number except that the clean part of the graph, in which the fugitive cannot be, is a connected graph. For trees, this parameter has been proved to be within a factor two of the node search number [2], and a linear time distributed algorithm has been proposed in [1].

In this paper, we propose a distributed algorithm to compute the node search number, the edge search number and the proces number. Similarly to the algorithm of [1], our algorithm uses a convergecast and our main contribution is the intoduction of a new data structure called *hierarchical decomposition*.

In Section 2, we give a formal definition of the node search number and related parameters. In Section 3, we propose a distributed algorithm to compute the node search number in trees. In Section 4, we show how to update it in a forest after addition or deletion of any tree-edges. We deduce an incremental algorithm to compute the node search number in trees whose edges are added sequentially and in any order (Section 4). In Section 5, we show how to adapt these algorithms to compute other graph invariants such as the process number and the edge search number, and how to extend our algorithms to trees and forests of unknown size.

2 Definitions and Context

In this section, we present all the games and graph parameters studied in this paper.

2.1 Node Search Number

The node search number, denoted by $\mathtt{sn}$, is the minimum number of agents needed to catch an invisible and fast fugitive hidden in a graph in a cops and robber game. The rules of this two-player game are as follows: at any time, the first player can move the fugitive from a node to another along a path in which no node is occupied by an agent. At each turn, the second player can execute one of the following two actions:

(1) put an agent on a node

(2) remove an agent from a node

The fugitive is captured when an agent is located on the same node. Note that the second player does not know the position of the fugitive. A *p-search strategy* is a strategy which uses exactly p agents to capture the fugitive, regardless of its strategy. A *($\leq p$)-search strategy* is a strategy which uses at most p agents to capture the fugitive, regardless of its strategy. The *node search number* of a graph G, denoted by $\mathtt{sn}(G)$, is the smallest p such that a p-search strategy for G exists. For example, a star has node search number 2, a path has node search number 2, a cycle has node search number 3, and a $n \times n$ grid where $n \geq 2$ has node search number $n + 1$.

During a p-search strategy, nodes can be divided in three types: *guarded* nodes on which there is an agent, *unsafe* nodes on which the fugitive might be, and *safe* nodes standing for all other nodes.

Definition 1 (monotone p-search strategy)**.** *A p-search strategy is* monotone *if the unsafe part of the graph never grows. In other words, a node on which an agent has been put can never host the fugitive again after the removal of the agent.*

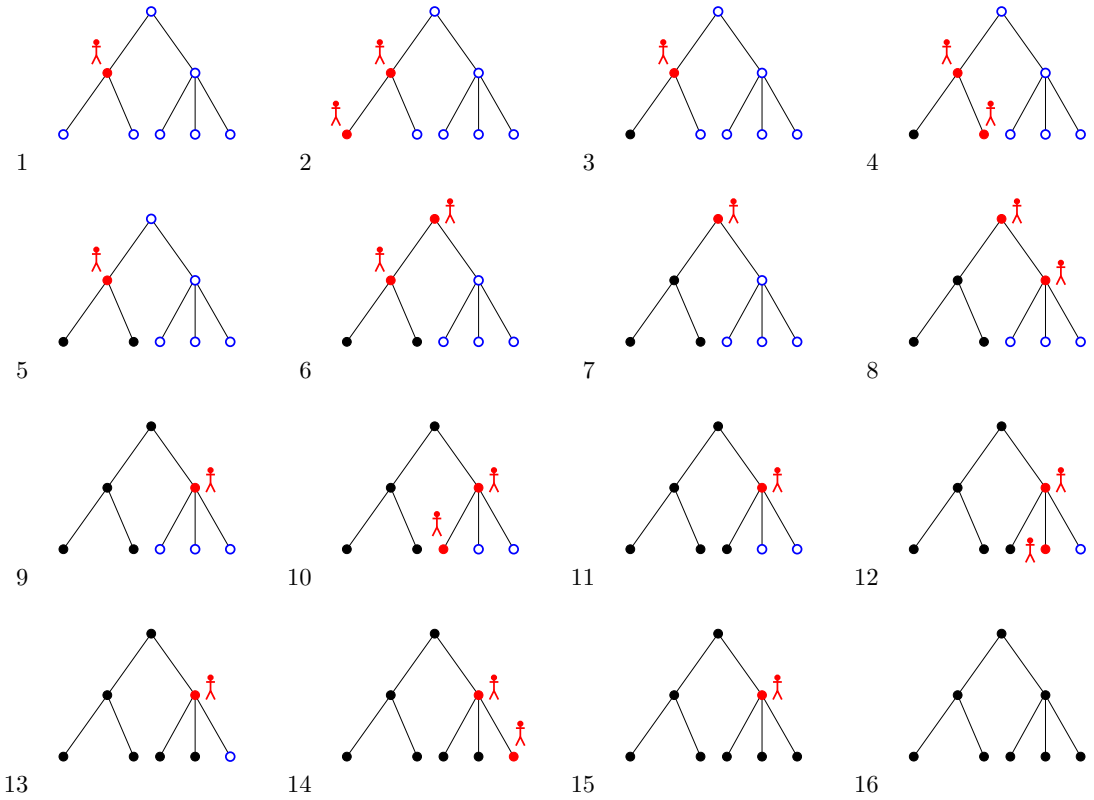

Figure 1: Monotone 2-search strategy for a simple tree in which filled nodes without agent represent safe nodes, and unfilled nodes represent unsafe nodes.

It has been proved by LaPaugh [14] that if a p-search strategy exists for a graph G, then there also exists a monotone p-search strategy for G. Thus we only consider monotone strategies from here on. We describe any monotone p-search strategy for $G = (V, E)$ by a sequence of $2n = 2|V|$ movements of the p agents: $m_1, \ldots, m_{2n}$, where m_i, $1 \leq i \leq 2n$, is one of the two actions *'put an agent on a node $u \in V$'* and *'remove an agent from a node $u \in V$'*. Note that an agent can be removed from a node only if it has previously been put on that node. Moreover, we consider only strategies that never put more than one agent per node. Indeed, it is always possible to transform a strategy placing at least two agents on the same node into a strategy placing only one agent on that node: from this strategy, we remove movements corresponding to *place an agent to a node* if such an event has already occured for this node, as well as the corresponding movements *remove an agent from a node*. We get a new strategy in which each node gets an agent on it once and the number of agents used does not increase. Thus, we assume that an agent is never put at a guarded node or at a safe node (recall that a safe node is a node where the fugitive cannot be and which is not guarded). Therefore, we consider only monotone strategies with the same number n of put and remove movements and such that all agents are removed from the graph at the end of the sequence S of movements. Figure 1 shows a monotone 2-search strategy for a simple tree.

Remark 1. *A search strategy with $2n$ movements that never puts more than one agent per node is monotone.*

Definition 2 (p-search strategy finishing at v)**.** *Given a graph $G = (V, E)$ and a node $v \in V$, we say that a p-search strategy* finishes at v *if the last movement m_{2n} is 'remove the agent from node v'.*

Definition 3 (p-search strategy starting at v)**.** *Given a graph $G = (V, E)$ and a node $v \in V$, we say that a p-search strategy* starts at v *if v is the first node with an agent on it.*

Remark 2. *As we consider only monotone search strategies, a p-search strategy starting at v ensures that in all the following steps, the fugitive cannot go to v.*

The following lemma shows that the two notions of starting and finishing at a node are equivalent for monotone search strategies. This result belongs to the folklore of the field, but we provide its proof for completeness.

Lemma 1. *Given a graph $G = (V, E)$ and a node $v \in V$, if there is a monotone p-search strategy starting at v, then there is a monotone p-search strategy finishing at v and vice-versa.*

Proof. Let S be a sequence of $2n = 2|V|$ movements $m_1, m_2, \ldots, m_{2n}$ describing a monotone p-search strategy for G finishing at v, i.e. with $m_{2n} =$ *'remove an agent from node v'*.

We define $\overline{m}_i$ as the movement *'put an agent on node $u \in V$'* (respectively *'remove an agent from node $u \in V$'*) when m_i is the movement *'remove an agent from node $u \in V$'* (respectively *'put an agent on node $u \in V$'*). Let $\overline{S}$ be the sequence of movements $\overline{m}_{\sigma(1)}, \overline{m}_{\sigma(2)}, \ldots, \overline{m}_{\sigma(2n)}$, where $\sigma(i) = 2n - i + 1$.

First note that if $\overline{S}$ gives a search strategy starting at v, then it is necessarily monotone (see Remark 1). Furthermore if $\overline{S}$ is a search strategy, then it is a p-search strategy. Indeed if we label the p agents, then any node $u \in V$ with agent i on it, $1 \leq i \leq p$, in S, also has agent i on it in $\overline{S}$.

We now prove that $\overline{S}$ gives a monotone p-search strategy starting at v, that is to say starting with the movement $\overline{m}_{2n} =$ *'put an agent on node v'*. Assume that $\overline{S}$ is not a search-strategy. Then it means that, in $\overline{S}$, there is an edge $(u, v) \in E$ such that

1. we first put an agent at u,
2. we then remove this agent from u,
3. we then put an agent at v,
4. and finally we remove the agent from v.

Note that these steps do not need to occur consecutively and that there are no more movements concerning u or v in $\overline{S}$ because of the correctness of the monotone search strategy S. Consequently it means that, in S,

1. we first put an agent at v,
2. we then remove this agent from v,
3. we then put an agent at u,
4. and finally we remove the agent from u.

A contradiction because S is a valid p-search strategy. □

Remark 3. *Given a graph $G = (V, E)$, for any node $v \in V$, a p-search strategy can be transformed into a $(\leq p + 1)$-search strategy finishing (or starting) at v by adding, if it is necessary, a $(p + 1)$th agent on v and letting it there during the whole p-search strategy. The "$\leq$" stems from the fact that the p-search strategy can already be a strategy finishing (or starting) at v.*

2.2 Pathwidth

The notion of pathwidth was introduced by Robertson and Seymour [19]. A *path decomposition* of a graph $G = (V, E)$ is a set system $(X_1, \ldots, X_r)$ of V such that

1. $\bigcup_{i=1}^{r} X_i = V$;
2. $\forall (x, y) \in E, \exists i \in \{1, 2, \ldots, r\} : \{x, y\} \subseteq X_i$;
3. $\forall (i_0, i_1, i_2) \in \{1, 2, \ldots, r\}^3, i_0 < i_1 < i_2 \Rightarrow X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$.

The *width* of the path decomposition $(X_1, \ldots, X_r)$ is $\max_{1 \leq i \leq r} |X_i| - 1$. The *pathwidth* of G, denoted by $\mathtt{pw}(G)$, is the minimum width over its path decompositions.

It was proved by Ellis *et al.* [8] that $\mathtt{sn}(G) = \mathtt{pw}(G) + 1$.

2.3 Vertex Separation

A *layout* (or *vertex-ordering*) L of a graph $G = (V, E)$ is a one-to-one correspondence between V and $\{1, \ldots, |V|\}$. The *vertex separation of* (G, L) is $\max_{1 \leq i \leq |V|} |M(i)|$ where

$$M(i) := \{v \in V : L(v) > i \text{ and } \exists u \in N(v) : L(u) \leq i\},$$

where $N(v)$ is the set of neighbors of v.

The *vertex separation of* G, denoted by $\mathtt{vs}(G)$, is the minimum of the vertex separation of (G, L) taken over all vertex-orderings L.

Kinnersley [12] proved that the pathwidth of a graph is equal to its vertex separation. Thus node search number, pathwidth, and vertex separation are equivalent: $\mathtt{sn}(G) = \mathtt{pw}(G) + 1 = \mathtt{vs}(G) + 1$. However, it is not known so far whether an equivalence also holds for the process number defined below, or not: if given the value of a parameter it is possible to compute the other one in polynomial time (with a time complexity independent of the known parameter's value), unless the graph is a tree [18].

2.4 Process Number

The process number was introduced as a cost function for rerouting strategies in connection oriented networks (e.g. optical networks). This parameter was originally defined for directed graphs [4–6]. It can be defined for symmetric digraphs in a cops and robber game manner on the underlying undirected graph. As for the node search number, the fugitive is captured when an agent is located on the same node but now it is also caught when it is surrounded by agents. This means that, at each turn, the second player (controlling the cops) can execute a third action:

(1) put an agent on a node

(2) remove an agent from a node

(3) clear a node if each of its neighbors has an agent on it.

Recall that the second player does not know the position of the fugitive. We say that a node is *processed* if the fugitive cannot be located at this node. A *p-process strategy* is a strategy which uses exactly p agents to capture the fugitive, regardless of its strategy. A *$(\leq p)$-process strategy* is a strategy which uses at most p agents to capture the fugitive, regardless of its strategy. The *process number* of a graph G, denoted by $\mathtt{pn}(G)$, is the smallest p such that a p-process strategy for G exists. For example, a star has process number 1, a path has process number 2, a cycle has process number 3, and a $n \times n$ grid where $n \geq 2$ has process number $n + 1$.

It was proved by Coudert *et al.* [5] that $\mathtt{vs}(G) \leq \mathtt{pn}(G) \leq \mathtt{vs}(G) + 1$.

2.5 Edge Search Number

For the node search number, the first player can move the fugitive from a node to another along a path in which no node is occupied by an agent but now the first player can move the fugitive on an edge. For the edge search number, the fugitive can hide anywhere, including on an edge, and there is an additional move allowed for the agents (a third action for the second player):

(1) put an agent on a node

(2) remove an agent from a node

(3') an agent on a node u can slide along an edge (u, v).

Nodes and edges can be divided in three types: *guarded* nodes on which there is an agent, *unsafe* nodes and edges on which the fugitive might be, and *safe* nodes and edges standing for all other nodes and edges. An edge (u, v) becomes safe if an agent slides along (u, v) and if the fugitive cannot go to (u, v) (there is no path from the current position of the fugitive to (u, v) composed only of unsafe nodes and unsafe edges). The agent is then located at node v but it is possible to remove it just after. For monotone strategies, it may be necessary to keep the agent at v. For instance consider a path of 3 nodes

u_1, u_2, and u_3. Assume that all nodes and edges are unsafe but u_1 which is guarded by an agent. The second player can use the third action to slide the agent along edge (u_1, u_2). After this, node u_1 and edge (u_1, u_2) are safe, node u_2 is guarded by an agent, and node u_3 and edge (u_2, u_3) are unsafe. In that case, we cannot remove the agent from u_2 because we consider monotone strategies. Then, the agent can slide along edge (u_2, u_3) to finish the strategy.

Recall that the second player does not know the position of the fugitive. A *p-strategy* is a strategy which uses exactly p agents to capture the fugitive, regardless of its strategy. A *($\leq p$)-strategy* is a strategy which uses at most p agents to capture the fugitive, regardless of its strategy. The *edge search number* of a graph G, denoted by $\texttt{es}(G)$, is the smallest p such that a p-strategy for G exists.

It was proved by Kirousis *et al.* [13] that $\texttt{sn}(G) - 1 \leq \texttt{es}(G) \leq \texttt{sn}(G) + 1$. Then in [18], Peng *et al.* characterized particular classes of trees in which equality holds, namely the sprout trees (each node is incident to a leaf) and the reduction trees (without vertices of degree 2).

2.6 Generic Construction

Theorem 2 ([5, 6, 17, 18]). *Given a tree T and an integer $p \geq 1$, $\texttt{es}(T) \geq p + 1$ iff T has a vertex v at which there are at least three branches T_i, $i = 1, 2, 3$, such that $\texttt{es}(T_i) \geq p$. The same holds for $\texttt{vs}(T)$ and $\texttt{pw}(T)$, but also for $\texttt{sn}(T)$ and $\texttt{pn}(T)$ when $p \geq 2$.*

This theorem has first been proved by Parson for the edge search number [17]. Later, the proof has been adpated for the node search number [18] and the process number [5, 6] (when $p \geq 2$). The same result also holds for $\texttt{pw}(T)$ and $\texttt{vs}(T)$. This theorem provides a construction which forces the edge search number to grow by 1. Node v of Theorem 2 is usually called a Parsons node [17]. In general, Theorem 2 implies that for any tree T; $\texttt{sn}(T)$, $\texttt{pw}(T)$, $\texttt{vs}(T)$, $\texttt{pn}(T)$, and $\texttt{es}(T)$ are less than $\log_3(n)$, where n is the number of nodes of T. This can be proved by induction on the value of the parameter. Indeed, the minimum size of a tree with the parameter equal to $p + 1$ is at least three times the minimum size of a tree with the parameter equal to p.

2.7 Contribution

In this paper, we propose an algorithm to compute all the parameters defined in the previous sections for trees (node search number, pathwidth, vertex separation, process number, and edge search number). We present the algorithm using the node search number. Changes performed on the algorithm for the other parameters are given in the last few Sections. The algorithm is based on the decomposition of a tree into subtrees forming a *hierarchical decomposition* (Section 3, Definition 6). Note that our algorithm is fully distributed and that it can be executed in an asynchronous environment, where each node is considered as a processor which knows its neighbors. Furthermore, the construction of the hierarchical decomposition requires only a small amount of information. Overall it requires $O(n \log n)$ operations and transmits at most $n(\log_3 n + 4)$ bits (Section 3). We then extend our algorithm to a fully dynamic algorithm (Section 4) allowing to add and remove edges. We also show how to adapt our algorithms when the total size of the tree is unknown, using messages of up to $2 \log_3 n + 5$ bits (Section 5). The main contribution of our algorithm compared to previous proposals by Scheffler [20], Ellis *et al.* [8], Peng *et al.* [18], or Golovach [11], is the use of the hierarchical decomposition. Indeed, previously proposed algorithms are centralized. Using the hierarchical decomposition, we are able to compute parameters using only local information without global knowledge of the tree structure. Furthermore, it is flexible enough to turn our algorithm into a fully dynamic algorithm allowing the update of the node search number (or edge search number, or process number, or pathwidth) after the addition or removal of any tree-edge at low computational cost. It could also be adapted to compute the mixed search number and other similar parameters.

Throughout this paper, we assume that each node u knows its set of neighbors $N(u)$, with $d(u) = |N(u)|$. However, knowledge of the size of the tree is not needed as explained in Section 5.

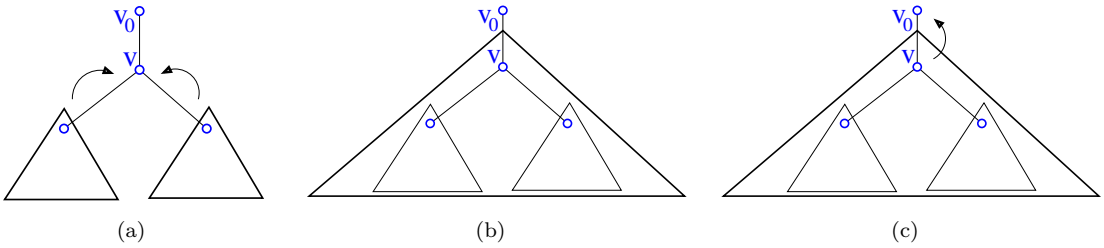

Figure 2: Node v collects information on subtrees rooted at each of its children (a), performs computations (b), and sends information to its parent v_0 (c).

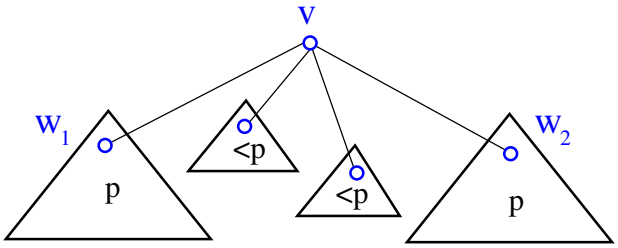

Figure 3: Generic unstable tree T_v rooted at node v, with $\mathtt{sn}(T_v) = p > 1$. There are two stable subtrees T_{w_1} and T_{w_2} rooted at w_1 and w_2 (two children of v) with $\mathtt{sn}(T_{w_1}) = \mathtt{sn}(T_{w_2}) = p$ and other subtrees whose node search numbers are at most $p - 1$.

3 Algorithm for the Node Search Number

Our algorithm performs a *convergecast* to compute the node search number $\mathtt{sn}(T)$ of a tree $T = (V, E)$. Starting from the leaves, each node $v \in V$ collects information about the subtrees rooted at its $d(v) - 1$ children (Figure 2(a)). Concretely this information is the hierarchical decompositions of these subtrees (Definition 6). Then, v computes a hierarchical decomposition of the subtree T_v (Figure 2(b)). Finally, v sends this hierarchical decomposition to its parent v_0 (last neighbor) (Figure 2(c)). The subtree T_v is the connected component containing v when removing the edge (v, v_0) from T.

To define a hierarchical decomposition, we introduce two specific structures of trees: stable and unstable trees.

3.1 Stable Tree vs Unstable Tree

Definition 4 (stable tree)**.** *A tree T_v rooted at node v, with $\mathtt{sn}(T_v) = p$, is called a* stable tree *if there is a p-search strategy which finishes (starts) at node v.*

Definition 5 (unstable tree)**.** *A tree T_v rooted at node v, with $\mathtt{sn}(T_v) = p > 1$, is called an* unstable tree *if there exist two stable subtrees T_{w_1} and T_{w_2} (respectively rooted at two children of v: w_1 and w_2) such that $\mathtt{sn}(T_{w_1}) = \mathtt{sn}(T_{w_2}) = p$ and all other subtrees $T_{w_3}, \ldots, T_{w_j}$, respectively rooted at each other child of v: $w_3, \ldots, w_j$, are such that $\mathtt{sn}(T_{w_i}) < p$, $3 \leq i \leq j$. (Figure 3 shows a generic unstable tree).*

Remark 4. *In the proof of Lemma 3 we will describe a p-search strategy for unstable trees.*

Remark 5. *There are trees that are neither stable nor unstable.*

Remark 6. *There is no unstable tree T with $\mathtt{sn}(T) = 1$.*

Property 1. *Given an unstable tree T_v rooted at node v, with $\mathtt{sn}(T_v) = p > 1$, there is no p-search strategy finishing (starting) at node v.*

Proof. Let $w_1, \ldots, w_j$ be the children of v, and let T_{w_1} and T_{w_2} be the two stable subtrees such that $\mathtt{sn}(T_{w_1}) = \mathtt{sn}(T_{w_2}) = p$. Assume that there exists a p-search strategy for T_v starting at v. It naturally gives a p-search strategy for $T_{w_1} \cup T_{w_2} \cup \{v\}$ starting at v. Such a strategy begins to clear one of the two subtrees T_{w_1} and T_{w_2}, say T_{w_1}. But while the p-search strategy is clearing T_{w_1}, by definition of a p-search strategy starting at v, it guarantees that the fugitive may not go to v, otherwise v would be recontaminated (*i.e. the fugitive may be on v, whereas an agent was already put on v*) wich would violate our hypothesis as we consider only monotone p-search strategies. Hence there must be an agent on v or on some nodes of T_{w_2}. Thus a search strategy starting at v needs at least $p+1$ agents. Recall that we consider only strategies that never put more than one agent per node (Section 2).

By Lemma 1, we get the same result for a p-search strategy finishing at v. □

Lemma 3. *Given a tree $T = (V, E)$ and an unstable subtree T_v rooted at node $v \in V$, with $\mathtt{sn}(T_v) = p > 1$, then $\mathtt{sn}(T) = p$ if and only if $\mathtt{sn}(T \setminus T_v) \leq p - 1$.*

Proof. Let T_{w_1} and T_{w_2} be the two stable subtrees (respectively rooted at two children of v: w_1 and w_2) such that $\mathtt{sn}(T_{w_1}) = \mathtt{sn}(T_{w_2}) = p$.

If $\mathtt{sn}(T \setminus T_v) \geq p$, then there are three disjoint subtrees rooted at three different neighbors of v, each having node search number at least p. If one of the subtrees has node search number greater than p then $\mathtt{sn}(T) \geq p + 1$. Otherwise, the three subtrees have node search number p and, by Theorem 2, $\mathtt{sn}(T) = p + 1$.

Otherwise $\mathtt{sn}(T \setminus T_v) \leq p - 1$ and we describe a p-search strategy for T. We start by a p-search strategy for T_{w_1} finishing at w_1. It uses p agents and finishes with w_1 occupied by an agent. We then place an agent on v and remove the one from w_1. We continue with a $(\leq p-1)$-search strategy for $T_v \setminus (T_{w_1} \cup T_{w_2} \cup \{v\})$. Now, since $\mathtt{sn}(T \setminus T_v) \leq p - 1$, we continue with a $(\leq p-1)$-search strategy for $T \setminus T_v$. We then place an agent on w_2 and remove the one from v. It now only remains to use a p-search strategy for T_{w_2} starting at w_2 which exists by assumption. □

Given a tree $T = (V, E)$, from Lemma 3, if we have an unstable subtree T_u rooted at node $u \in V$, with $\mathtt{sn}(T_u) = p$, computing $\mathtt{sn}(T \setminus T_u)$ allows to decide whether $\mathtt{sn}(T) = p$ or not. When we compute $\mathtt{sn}(T \setminus T_u)$, if no other unstable subtree is found, the exact value of $\mathtt{sn}(T)$ can be deduced. But if another unstable subtree $T_{u'}$ is found, we have to solve the same decision problem. After that, we have to compute $\mathtt{sn}(T \setminus (T_u \cup T_{u'}))$, and so on. Figure 4 represents this problem recursively.

For example, consider the tree T_v rooted at node v of Figure 4. It is composed of 6 disjoint subtrees: T^1, T^2, T^3, T^4, T^5 are unstable while T^0 rooted at v is stable, with node search numbers 4, 5, 6, 7, 8, and 3 respectively. T^5 is the unstable subtree of largest node search number ($\mathtt{sn}(T^5) = 8$) and from Lemma 3, we know that $\mathtt{sn}(T_v) = 8$ if and only if $\mathtt{sn}(T_v \setminus T^5) \leq 7$. Thus consider $T_v \setminus T^5$. T^4 is the unstable subtree of largest node search number ($\mathtt{sn}(T^4) = 7$) and from Lemma 3, we know that $\mathtt{sn}(T_v \setminus T^5) = 7$ if and only if $\mathtt{sn}(T_v \setminus (T^5 \cup T^4)) \leq 6$. And so on. At the end, we get that $\mathtt{sn}(T^0) \leq 3$, and so $\mathtt{sn}(T_v) = 8$.

As our algorithm is distributed and nodes have local knowledge, nodes need to transmit sufficient information to one another in order to describe the structure of the explored subtrees. In order to do so, we introduce in the next Section the notion of *hierarchical decomposition* that formalizes the idea of the previous example.

3.2 Hierarchical Decomposition

Definition 6 (hierarchical decomposition)**.** *Given a tree T_r rooted at node r, a* hierarchical decomposition *of T_r, denoted by $HD(T_r)$, is a family of trees $\{T^i\}_{0 \leq i \leq k}$ such that:*

- *the set of the subtrees $\{T^i\}_{0 \leq i \leq k}$ of T_r forms a partition of the nodes of T_r;*
- *T^0 is either a stable or an unstable tree rooted at node $v_0 = r$;*
- *T^i is unstable and it is rooted at a node v_i, $1 \leq i \leq k$;*
- *if two trees T^i and T^j, $0 \leq i \leq k$, $0 \leq j \leq k$, $i \neq j$, are such that the path going from v_i to r in T_r goes through v_j, then $\mathtt{sn}(T^i) > \mathtt{sn}(T^j)$.*

We associate to $HD(T_r)$ the pair $(int, vect)$. The first component int is equal to $\texttt{sn}(T^0)$ if T^0 is a stable tree, and -1 otherwise. The second component is a vector of length $L(vect)$, where $L(vect)$ is the largest node search number among the unstable trees of $HD(T_r)$. The i-th element of vect, denoted $vect[i]$, contains the number of unstable trees of $HD(T_r)$ whose node search numbers are i.

For example, the hierarchical decomposition $HD(T_v)$ of the tree T_v rooted at node v of Figure 4, is represented by a pair $(int, vect)$, as shown in Table 1.

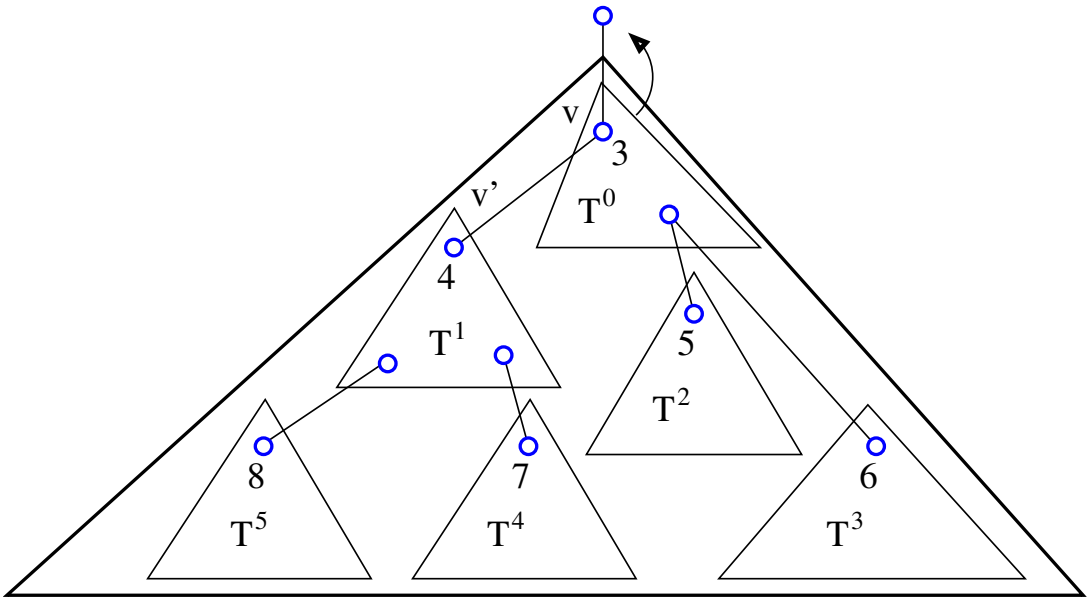

Figure 4: This minimal hierarchical decomposition $MHD(T_v)$ of tree T_v rooted at node v contains six disjoint trees: T^1, T^2, T^3, T^4, T^5 are unstable while T^0 rooted at v is stable. The node search numbers of these trees are indicated by integers located on roots. The disjoint trees respect the order constraint. For example, there is an edge between the root of T^4 and one node of T^1.

	int	$vect[1]$	$vect[2]$	$vect[3]$	$vect[4]$	$vect[5]$	$vect[6]$	$vect[7]$	$vect[8]$
$HD(T_v) = MHD(T_v)$	3	0	0	0	1	1	1	1	1
$HD(T_{v'}) = MHD(T_{v'})$	-1	0	0	0	1	0	0	1	1

Table 1: pairs $(int, vect)$ associated with the hierarchical decompositions $HD(T_v)$ and $HD(T_{v'})$ of trees T_v and $T_{v'}$ of Figure 4. In this example, the hierarchical decompositions are also the minimal hierarchical decompositions $MHD(T_v)$ and $MHD(T_{v'})$.

Remark 7. *Unstable trees with node search number 1 do not exist. Thus, given a pair $(int, vect)$ associated with a hierarchical decomposition, we will always have $vect[1] = 0$.*

Remark 8. *A hierarchical decomposition of a tree T_r rooted at node r is associated with a unique pair $(int, vect)$, but a pair $(int, vect)$ can be associated with several hierarchical decompositions. Indeed, edges connecting different trees of the partition of the hierarchical decomposition have no influence on the pair $(int, vect)$ as long as they do not violate the definition of the hierarchical decomposition. For example, consider the tree T_v shown in Figure 4. We observe the tree T^5, with $\texttt{sn}(T^5) = 8$, could be attached to T^3 having $\texttt{sn}(T^3) = 6$, which does not modify the representation of the hierarchical decomposition of T_v in Table 1.*

3.3 Minimal Hierarchical Decomposition

Definition 7 (minimal hierarchical decomposition)**.** *Given a tree T_r rooted at node r, a* minimal hierarchical decomposition *of T_r, denoted by $MHD(T_r)$, is a hierarchical decom-*

position of T_r (Definition 6) such that $\forall i, j \in [0, k]$, $i \neq j$, we have $\mathtt{sn}(T^i) \neq \mathtt{sn}(T^j)$.

The existence of a minimal hierarchical decomposition is guaranteed by Theorem 6 that will be proved in Section 3.4. More precisely, Theorem 6 implies that there is a vertex v for which a minimal hierarchical decomposition of T_v exists. But our algorithm can be slightly modified so as to obtain a minimal hierarchical decomposition of T at any vertex r. It suffices to add in our algorithm that r does not transmit any message.

Property 2. *Given the representation $(int, vect)$ of the minimal hierarchical decomposition $MHD(T_r)$ of a tree T_r rooted at node r, $\forall i \in [2 \dots L(vect)]$, we have $vect[i] \in \{0, 1\}$ and $vect[1] = 0$.*

This property is directly implied by Definition 7 and Remark 7.

Lemma 4. *Given the representation $(int, vect)$ of the minimal hierarchical decomposition $MHD(T_r)$ of a tree T_r rooted at node r, we have $\mathtt{sn}(T_r) = \max(int, L(vect))$.*

Proof. Recall that $L(vect)$ is the largest node search number among the unstable trees of $MHD(T_r)$. If $int \geq L(vect)$, then $MHD(T_r)$ is composed of a single stable tree $T^0 = T_r$ and $\mathtt{sn}(T_r) = int$.

If $int < L(vect)$, then we prove the assertion by induction on $L(vect)$. Since $MHD(T_r)$ is a minimal hierarchical decomposition of T_r, there is a unique unstable tree T^k such that $\mathtt{sn}(T^k) = L(vect)$. Thus considering $MHD(T_r)$ minus tree T^k, we get a minimal hierarchical decomposition $MHD(T_r \setminus T^k)$ of $T_r \setminus T^k$. Hence, the length of the vector associated with $MHD(T_r \setminus T^k)$ has length strictly less than $L(vect)$. By induction hypothesis $\mathtt{sn}(T_r \setminus T^k) < L(vect)$.

A $L(vect)$-search strategy for T_r is described as follows: start with a $L(vect)$-search strategy for T^k. There exists one which at some step has an agent on its root v_k and no other agent is located on another node. This is always possible using the generic optimal search strategy for unstable trees described in the proof of Lemma 3. At this step include a $(\leq L(vect) - 1)$-search strategy for $T_r \setminus T^k$. Recall that only the root v_k has an edge to a node of $T_r \setminus T^k$ because of the 4th property of hierarchical decompositions (Definition 6). Once it is done, finish the $L(vect)$-search strategy for T^k. □

3.4 Distributed Algorithm for the Node Search Number

We can now describe precisely algorithm `algoHD` (Algotithm 1) which constructs the minimal hierarchical decomposition and computes the node search number $\mathtt{sn}(T)$ of any given tree $T = (V, E)$. The main steps are as follows:

Algorithm 1 `algoHD`

- Each leaf sends the initialization message $(1, [\,])$, where $[\,]$ represents a vector of length 0, to its only neighbor which becomes its parent.
- A node $v \in V$ which has received messages from all neighbors but one, computes the pair $(int, vect)$ representing the minimal hierarchical decomposition $MHD(T_v)$ of T_v using Algorithm 2. Then v sends the message $(int, vect)$ to its last neighbor which thus becomes the parent.
- The last node $r \in V$ is called the root of T. When it has received a message from all neighbors, it computes the pair $(int, vect)$ representing the minimal hierarchical decomposition $MHD(T_r)$ of $T_r = T$ using Algorithm 2. Lemma 4 gives the node search number $\mathtt{sn}(T)$ of T. Remark 9 deals with the case in which two nodes receive messages from all their neighbors.

Remark 9. *It may happen that two adjacent nodes v and w receive a message from all their neighbors. It is the case when node v, after sending its message to its last neighbor w, receives a message from w. In this case, both v and w are potential candidates to be the root of the tree, but only one of them can be chosen. Several tricks can be used to ease the decision. The easiest is certainly to choose the vertex with largest identifier (assuming a total ordering on the nodes identifiers such as MAC address) thus avoiding the transmission of extra bits since we may assume that every node knows the identifier of its neighbors. Otherwise, a classical leader election mechanisms [14] can be used to determine the root. It can be done with $\log(n)$ bits.*

Algorithm 2 Computation of the representation $(int, vect)$ of the minimal hierarchical decomposition $MHD(T_v)$ of the tree T_v rooted at node v

Require: Representations $(int_1, vect_1), \ldots, (int_{d-1}, vect_{d-1})$ of the minimal hierarchical decompositions $MHD(T_{v_1}), \ldots, MHD(T_{v_{d-1}})$ of subtrees $T_{v_1}, \ldots, T_{v_{d-1}}$ rooted at children of v: $v_1, \ldots, v_{d-1}$.

Require: A vector $vect_{sum}$ such that $vect_{sum}[i] := vect_1[i] + \ldots + vect_{d-1}[i]$, $\forall i \in [2, \max_{1 \leq j \leq d-1} L(vect_j)]$ and $vect_{sum}[1] = 0$.

Ensure: $(int, vect)$ is the representation of the minimal hierarchical decomposition $MHD(T_v)$ of T_v.

Let (p, p') be the pair computed by Algorithm 3 from the values of the stable trees (possibly empty) of $MHD(T_{v_1}), \ldots, MHD(T_{v_{d-1}})$: $int_1, \ldots, int_{d-1}$.
if $p < p'$ **then** {/* `The union of the stable trees is unstable` */}
  $L(vect) := \max(L(vect_{sum}), p)$; $vect[j] := 0$, $\forall j \in [1, L(vect)]$
  $vect := vect_{sum}$
  $vect[p] := vect[p] + 1$
  $vect[j] := 0$, $\forall j \in [2, p-1]$
  $(p, p') := (-1, -1)$
else {/* $p == p'$ `and so the union of the stable trees is stable` */}
  $L(vect) := L(vect_{sum})$
  $vect := vect_{sum}$
  $vect[j] := 0$, $\forall j \in [2, p-1]$
Let k be such that $vect[k] > 1$ and $vect[i] \leq 1$, $\forall i \in [k+1, L(vect)]$ /* `if` k `does not exist, then` $k := -1$ */
Let $k_1 > \max(k, p-1)$ be such that $vect[k_1] = 0$ and $vect[i] \geq 1$, $\forall i \in [\max(k, p), k_1 - 1]$
/* `we assume that there exists a virtual cell` $vect[L(vect)+1] = 0$ */
/* `if` k_1 `does not exist, then` $k_1 := -1$ */
if $k_1 > 1$ **then**
  $vect[i] := 0$, $\forall i \in [2, k_1]$
  $int := k_1$
else {/* `if` $k_1 == -1$`, then the hierarchical decomposition is minimal` */}
  $int := p$
  if $vect[i] == 0, \forall i \leq L(vect)$ **then**
    $vect :=$ [ ] /* [ ] `is a vector of length 0` */
return $(int, vect)$

Algorithm 3 Computation of (p, p')

Require: A list of integers $int_1, \ldots, int_{d-1}$ which corresponds to the search number of stable trees as follow: given a tree T_v^0 rooted at v whose children are roots of stable trees (possibly empty) of node search numbers $int_1, \ldots, int_{d-1}$.

Ensure: $p = \mathtt{sn}(T_v^0)$ and p' is the minimum value such that a p'-search strategy finishing (or starting) with an agent located on v exists for T_v^0. Recall that $p \leq p' \leq p+1$.

1: $int_{max} := \max_{1 \leq j \leq d-1}\{int_j\}$ /* $int_{max} := -1$ `if there is no stable tree` */
2: $I := \{i; int_i = int_{max}\}$ /* `all` i `such that` int_i `is maximum` */
3: **if** $int_{max} < 2$ **then**
4: $\quad (p, p') := \begin{cases} (1,1) & \text{when } int_{max} = -1 \\ (2,2) & \text{otherwise} \end{cases}$
5: **else** {/* `general cases` */}
6: $\quad$ **if** $|I| == 2$ **then** {/* T_v^0 `is unstable` */}
7: $\quad\quad (p, p') := (int_{max}, int_{max} + 1)$
8: $\quad$ **else** {/* T_v^0 `is stable` */}
9: $\quad\quad$ **if** $|I| > 2$ **then** {/* `Theorem 2` */}
10: $\quad\quad\quad (p, p') := (int_{max} + 1, int_{max} + 1)$
11: $\quad\quad$ **else** {/* $I = 1$ */}
12: $\quad\quad\quad (p, p') := (int_{max}, int_{max})$
13: return (p, p')

Algorithm 2 uses Algorithm 3 which computes the node search number of the subtree resulting from the merging of the (possibly empty) stable subtrees from each minimal hierarchical decomposition received and the minimum value such that a search strategy finishing at the root exists.

Lemma 5. *Let T_v^0 be a tree rooted at node v such that the children of v are roots of (possibly empty) stable trees having node search numbers $int_1, \ldots, int_{d-1}$. Algorithm 3 computes the pair (p, p') associated with the tree T_v^0 with $p = \mathtt{sn}(T_v^0)$ and if T_v^0 is a stable tree, then $p' = p$, otherwise $p' = p + 1$ (unstable tree).*

of Lemma 5. Lines 3 and 4 deal with the initialization, when the resulting tree T_v^0 is either a single vertex or a star:

- if $\forall i \in [1, d-1],\ int_i = -1$, it means that there are no stable trees, i.e. $\forall i \in [1, d-1], V(T_{v_i}^0) = \emptyset$. Then T_v^0 is a single vertex $\{v\}$. We indeed have $(p, p') = (1, 1)$;
- if $\forall i \in [1, d-1],\ int_i < 2$ and $\exists i \in [1, d-1],\ int_i = 1$, then T_v^0 is either a path of length two or a star with central vertex v. We have $(p, p') = (2, 2)$.

Lines 6 and 7 deal with the case when there are exactly two stable trees whose node search numbers are of maximum value $p > 1$. In this case, the resulting tree T_v^0 is an unstable tree with $\mathtt{sn}(T_v^0) = p$ (see Section 3.1). Thus Algorithm 3 returns $(p, p+1)$.

Lines 9 and 10 are for the case when there are more than two stable trees with maximum node search number of value $p > 1$. In this case, Theorem 2 states that $\mathtt{sn}(T_v^0) = p + 1$. Furthermore there exists a $(p+1)$-search strategy finishing at v: we put an agent on v, we continue with $(\leq p)$-search strategies for stable trees rooted at children of v (sequentially), and finally we clear v, removing the agent from it. Thus Algorithm 3 returns $(p+1, p+1)$.

Finally, when there is only one stable subtree, without loss of generality $T_{v_1}^0$, with maximum node search number $p > 1$, we still get a stable subtree T_v^0 with $\mathtt{sn}(T_v^0) = p$. Indeed a p-search strategy finishing at v for T_v^0 consists in a p-search strategy for $T_{v_1}^0$ finishing at v_1, putting an agent on v, removing the agent from v_1 and $(\leq p-1)$-search strategies for the other stable trees. Thus Algorithm 3 returns (p, p). □

Theorem 6. *Given a tree $T = (V, E)$,* `algoHD` *computes the pair $(int, vect)$ associated with the minimal hierarchical decomposition $MHD(T_v)$ of $T_v = T$ for some vertex $v \in V$.*

Proof. We prove Theorem 6 by induction on the number of nodes of the tree. The initialization is when T_v is a single vertex or a star with center v.

- Initialisation case 1: If T_v is a single vertex (a leaf of T), then v receives no information, i.e. $int_{max} = -1$, $I = \emptyset$, and $vect_{sum} = [\,]$. In this case, Algorithm 3 returns $(1, 1)$ and Algorithm 2 returns $(1, [\,])$ which is correct.

- Initialisation case 2: If T_v is a star with center v, then v receives information from some leaves of T. We still have $vect_{sum} = [\,]$, and $int_{max} = 1$. In this case, Algorithm 3 returns $(2, 2)$ and Algorithm 2 returns $(2, [\,])$ which is correct.

In the general case, by the induction hypothesis, v receives from its children, $v_1, \ldots, v_{d-1}$, pairs corresponding to the minimal hierarchical decompositions of $T_{v_1}, \ldots, T_{v_{d-1}}$. $T_{v_1}, \ldots, T_{v_{d-1}}$ are the trees respectively rooted at $v_1, \ldots, v_{d-1}$.

Given all these minimal hierarchical decompostions, we now prove that Algoithm 2 computes a minimal hierarchical decomposition of T_v. By Lemma 5, Algorithm 3 returns the vector (p, p') associated with the tree T_v^0 formed by stable trees (possibly empty) of $MHD(T_{v_1}), \ldots, MHD(T_{v_{d-1}})$ with node search numbers $int_1, \ldots, int_{d-1}$.

In a hierarchical decomposition, we have to respect a hierarchy between the trees. To guarantee this, the trees of the minimal hierarchical decompositions of $T_{v_1}, \ldots, T_{v_{d-1}}$ whose node search numbers are strictly smaller than p are added to T_v^0.

$MHD(T'_{v_1}), \ldots, MHD(T'_{v_{d-1}})$ represent the minimal hierarchical decompositions obtained from $MHD(T_{v_1}), \ldots, MHD(T_{v_{d-1}})$, respectively, keeping only the stable trees and the unstable trees whose node search numbers are strictly smaller than p. There are 3 cases:

- if $p = p'$ and if there is a unique $i \in [1, d-1]$ such that $int_i = p$, then there is a p-search strategy for $T_{v_i}^0 = T'_{v_i}$ finishing with an agent located at v_i. So, we put an agent at v and remove the agent from v_i. Then we use a $(\leq p-1)$-search strategy for each $MHD(T'_{v_j})$ $(j \in [1, d-1], j \neq i)$. Indeed $\mathtt{sn}(T'_{v_j}) \leq p - 1$ by definition.

- if $p = p'$ and if there are at least 3 stable trees with node search numbers $p-1$ (there is no tree with node search number p), then we put an agent at v, we use a $(p-1)$-search strategy for the stable trees with node search numbers $p-1$, and we use a $(\leq p-1)$-search strategy for each $MHD(T'_{v_j})$ not containing the stable trees with node search number $p-1$.

- if $p' = p + 1$, then there are exactly 2 stable trees with node search number p, say $T'_{v_1} = T^0_{v_1}$ and $T'_{v_2} = T^0_{v_2}$. Recall that by definition there are no unstable trees in T'_{v_1} and T'_{v_2}. We use a p-search strategy for T'_{v_1} finishing with an agent located at v_1. We then put an agent at v removing the agent from v_1. After, we use a $(\leq p-1)$-search strategy for each $MHD(T'_{v_j})$ $(j \in [3, d-1])$. We put an agent at v_2 removing the agent from v. Finally, we use a p-search strategy for T'_{v_2}.

This is accomplished in lines 6 and 11. The action performed in these lines is to delete the entry in $vect_{sum}$ of the trees merged with T_v^0. If T_v^0 is unstable with $\mathtt{sn}(T_v^0) = p$, then $vect_{sum}[p]$ is incremented, and (p, p') set to $(-1, -1)$.

$(p, vect_{sum})$ corresponds to the minimal hierarchical decomposition $MHD(T_v)$ of T_v and $MHD(T_v)$ is composed of T_v^0 and all other unstable subtrees not merged in T^0 if (line 17):

- $p = -1$, and $vect_{sum}$ contains only 0 and 1's or;

- $p \neq -1$, and $vect_{sum}$ contains only 0 and 1's, and cell p contains 0.

Otherwise the current hierarchical decomposition is not minimal since several trees have the same node search number. Lines 12 to 19 deal with this case. We define k as follows: k is the last cell of $vect_{sum}$ with an integer strictly greater than one, if such a cell exists, otherwise $k = -1$. We define k_1 as follows: if $k \neq -1$, then k_1 is the first cell with a zero after cell k; if $k = -1$, then k_1 is the first cell with a zero after cell p.

We now prove that adding to T_v^0 all trees of the various minimal hierarchical decompositions $MHD(T_{v_1}), \ldots, MHD(T_{v_{d-1}})$ whose node search numbers are at most k_1, gives a stable tree with node

search number k_1 (we keep calling this tree T_v^0). To prove this we need to expose a k_1-search strategy starting at v and then show that no (k_1-1)-search strategy exists.

We start to describe a k_1-search strategy starting at v. The first agent is put on v, the root of T_v. The k_1-search strategy consists in searching in each branch of T_v^0 one after the other. The ith branch is composed of the trees of $MHD(T_{v_i})$ whose search numbers are strictly less than k_1 (there are no trees whose node search numbers are k_1 since $vect_{sum}[k_1] = 0$). These trees form a minimal hierarchical decomposition of the ith branch, and so by Lemma 4, there exists a (k_1-1)-search strategy for it. Thus we have a k_1-search strategy for T_v^0 because we have to keep the agent located on v.

We now prove that no (k_1-1)-search strategy exists for T_v^0 by induction on $k_1 - \max(k,p)$.

- $k_1 - \max(k, p-1) = 1$:
  - $\max(k, p-1) = p-1$ implies $k_1 = p$. But then we know by definition of p that no $(k_1-1) = (p-1)$-search strategy exists for T_v^0.
  - If $\max(k, p-1) = k$, T_v^0 contains two unstable subtrees, say T_1 and T_2, whose node search numbers are k. Hence $sn(T_v^0) \geq k$. Furthermore, the node search number of $T_v^0 \setminus T_1$, which contains T_2, has also search number at least k. By Lemma 3, as T_v^0 contains an unstable tree T_1 with search number k and that the rest of the tree has not search number less than $k-1$, we now that $sn(T_v^0) \neq k$. As $sn(T_v^0) \geq k$, we obtain $sn(T_v^0) > k$. Consequently, no $(k_1-1) = k$-search strategy exists for T_v^0.

  the case $\max(k, p-1) = p-1$ was considered above, the current hierarchical decomposition is already minimal. If $\max(k, p-1) = k$, the tree composed of the two unstable subtrees whose node search numbers are k and a path joining them has node search number $k+1$. Indeed, by Lemma 3, the node search number is at least $k+1$, and we describe a $(k+1)$-search strategy. We first use a $(k+1)$-search strategy for one of the two unstable subtrees finishing with an agent at its root. Then we use a 2-search strategy for the path finishing with an agent at the root of the second unstable subtree. Finally we use a $(k+1)$-search strategy for this second unstable subtree (starting with an agent at its root). Thus $\texttt{sn}(T_v^0) \geq k_1$. Hence T_v^0 is a stable tree with $\texttt{sn}(T_v^0) = k_1$.

- $k_1 - 1 - \max(k, p-1) \Rightarrow k_1 - \max(k, p-1)$: we suppose it is true for $k_1 - 1 - \max(k, p-1)$, we prove it is true for $k_1 - \max(k, p-1)$. Let T^1 be the unstable tree with node search number $k_1 - 1$. By induction hypothesis, the tree $T_v^0 \setminus T^1$ rooted at v has node search number $k_1 - 1$. Hence, by Lemma 3, T_v^0 has node search number k_1.

Line 15 consists in deleting from *vect* all entries corresponding to trees merged with T_v^0 (those whose node search numbers are at most $k_1 - 1$) and line 16 sets *int* to k_1. We have a new pair $(int, vect)$ with $\texttt{sn}(T_v^0) = int$ and *vect* the trees of the minimal hierarchical decompositions of $T_{v_1}^0, \ldots, T_{v_d}^0$ whose node search numbers are strictly greater than $\texttt{sn}(T_v^0) = k_1$. By choice of k_1, there are no two trees with equal node search number anymore, so that $(int, vect)$ corresponds to the minimal hierarchical decomposition $MHD(T_v)$ of T_v.

Lines 19 and 20 are there to satisfy the convention that a vector full of zeros is replaced by the empty vector [ ]. □

3.5 Examples

We present here an example of execution of `algoHD` for a tree T_u rooted at node u. As shown in Figure 5(a), T_u consists in three trees with minimal hierarchical decompositions MHD_1, MHD_2, and MHD_3 such that the two roots of MHD_1 and MHD_2 are linked via the node v, and v is linked to MHD_3 via u.

First, node v computes the minimal hierarchical decomposition $MHD(T_v)$ of the subtree T_v rooted at v. This is performed using the two minimal hierarchical decompositions MHD_1 and MHD_2 that it has received from Algorithm 2 (Figure 5(b)). The vector *vect* of $MHD(T_v)$ is obtained by summing the two vectors corresponding to MHD_1 and MHD_2 (see Table 2). As the two stable trees of MHD_1 and MHD_2 have node search number 2, we get an unstable tree in $MHD(T_v)$ with node search number 2 (Definition 5). Thus, we have $vect[2] = 1$ and $int = -1$ for the pair $(int, vect)$ associated with $MHD(T_v)$, since there is no stable tree in $MHD(T_v)$.

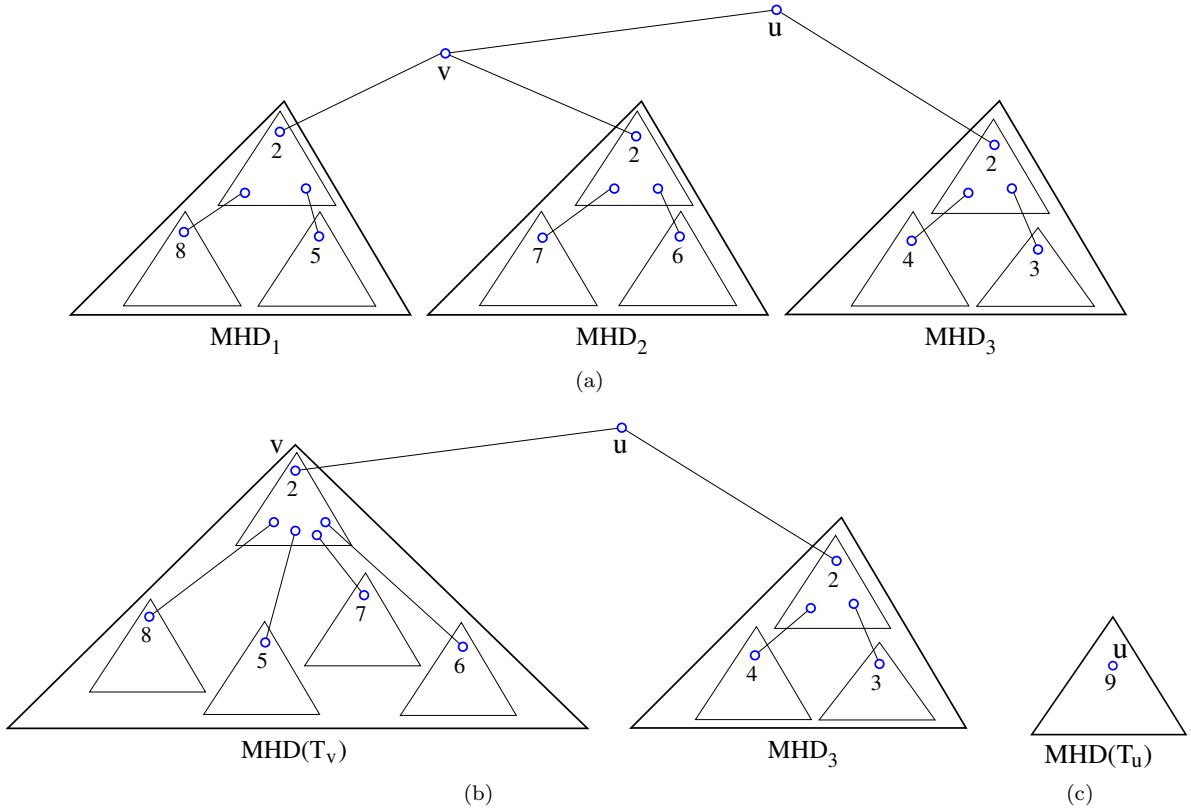

Figure 5: Example of execution of `algoHD` for a tree T_u rooted at node u.
(a) T_u consists in three trees with minimal hierarchical decompositions MHD_1, MHD_2, and MHD_3 connected via two nodes u and v. MHD_1 and MHD_2 contain one stable tree, and MHD_3 contain only unstable trees. The node search numbers of these trees are indicated by integers located on roots;
(b) the minimal hierarchical decomposition $MHD(T_v)$ of T_v rooted at v is obtained from MHD_1 and MHD_2 using Algorithm 2. $MHD(T_v)$ does not contain stable tree;
(c) the minimal hierarchical decomposition $MHD(T_u)$ of T_u rooted at u is obtained from $MHD(T_v)$ and MHD_3 using Algorithm 2. $MHD(T_u)$ is a single stable tree.

	int	$vect[1]$	$vect[2]$	$vect[3]$	$vect[4]$	$vect[5]$	$vect[6]$	$vect[7]$	$vect[8]$
MHD_1	2	0	0	0	0	1	0	0	1
MHD_2	2	0	0	0	0	0	1	1	
$MHD(T_v)$	-1	0	1	0	0	1	1	1	1
MHD_3	-1	0	1	1	1				
$HD(T_u)$	1	0	2	1	1	1	1	1	1
$MHD(T_u)$	9								

Table 2: pairs $(int, vect)$ associated with MHD_1, MHD_2, $MHD(T_v)$, MHD_3, $HD(T_u)$, and $MHD(T_u)$, minimal (but for $HD(T_u)$) hierarchical decompositions corresponding to trees of Figure 5.

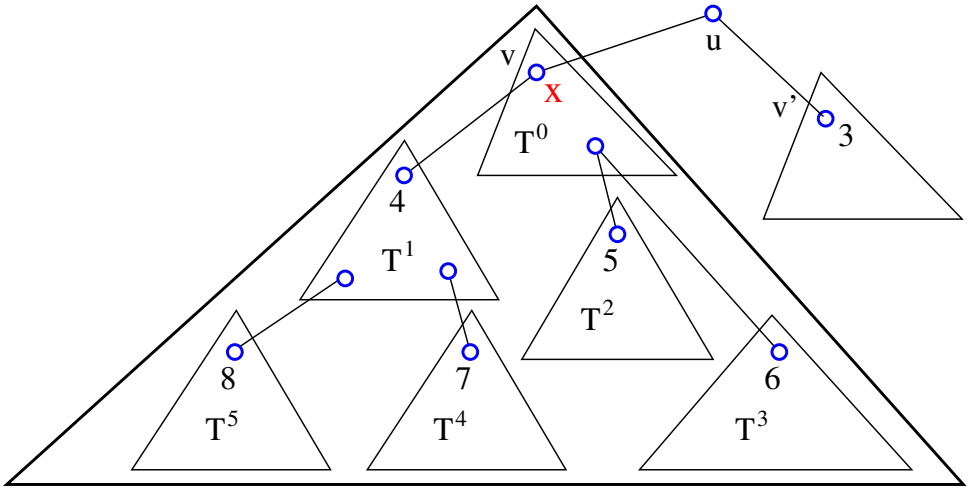

Figure 6: Computation of the minimal hierarchical decomposition $MHD(T_u)$ of tree T_u from $MHD(T_v)$ and $MHD(T_{v'})$ according to the node search number $\mathtt{sn}(T^0) = x \in \{2, 3\}$ of the unstable tree T^0. $MHD(T_v)$ is composed of 6 unstable trees $T^0, T^1, T^2, T^3, T^4, T^5$ with node search numbers $x, 4, 5, 6, 7, 8$ respectively. $MHD(T_{v'})$ is composed of a unique unstable tree with node search number 3.

Then, node u computes the minimal hierarchical decomposition $MHD(T_u)$ of the subtree T_u from $MHD(T_v)$ and MHD_3 using Algorithm 2 (Figure 5(c)). By summing the vectors of $MHD(T_v)$ and MHD_3, we obtain the vector for $HD(T_u)$ (see Table 2). Furthermore, the two integers of $MHD(T_v)$ and MHD_3 are -1 which yields a stable tree with node search number 1 in $HD(T_u)$. The minimal hierarchical decomposition $MHD(T_u)$ computed by Algorithm 2 is given in the last line of Table 2: it corresponds to a single stable tree and $\mathtt{sn}(T_u) = 9$. Over the execution of Algorithm 2, we have $k = 2$, $p = p' = 1$, and $k_1 = 9$.

3.6 Relevance of the notion of Hierarchical Decomposition

We will now show the relevance of the notion of minimal hierarchical decomposition. Consider the tree T_u rooted at node u depicted in Figure 6. The two minimal hierarchical decompositions of T_v and $T_{v'}$ respectively rooted at the two children of u: v and v' are represented. $MHD(T_v)$ is composed of 5 unstable trees T^1, T^2, T^3, T^4, T^5 with node search numbers $4, 5, 6, 7, 8$ respectively, and 1 unstable tree T^0 with node search number x. $MHD(T_{v'})$ is composed of a unique unstable tree with node search number 3. Consider two possible cases:

- if $\mathtt{sn}(T^0) = 2$, then by Theorem 6 $\mathtt{sn}(T_u) = 8$ and $MHD(T_u)$ is composed of 7 unstable trees with node search numbers $2, 3, 4, 5, 6, 7, 8$ and 1 stable tree with node search number 1.
- if $\mathtt{sn}(T^0) = 3$, then by Theorem 6 $\mathtt{sn}(T_u) = 9$ and $MHD(T_u)$ is composed of a unique stable tree.

This simple example shows that the knowledge of the node search number of the subtree T_v (that can either be stable or unstable) is not sufficient for node u to compute the node search number of the subtree T_u. Indeed, a more detailed description of T_u, and so of T_v, as provided by the hierarchical decomposition is needed.

3.7 Complexity

In this Section, we analyse the number of operations. The operations taken into account regarding memory access are read, write, add, subtract and compare.

Lemma 7. *Given a tree $T = (V, E)$ with n nodes, the time complexity of Algorithm 2 is $O(\log n)$. Thus,* `algoHD` *computes* $\mathtt{sn}(T)$ *in n steps and overall $O(n \log n)$ operations.*

Proof. Each node $v \in V$ of degree d_v has to compute int_{max} and I, which requires $O(d_v)$ operations, and the sum $vect_{sum}$ of all received tables, which requires $O(d_v \cdot L(vect_{sum}))$ operations. Finally it applies Algorithm 2 in which all operations are linear in $L(vect_{sum})$. As $\sum_{v \in V} d_v = 2(n-1)$ and $L(vect_{sum}) \leq \texttt{sn}(T) \leq \log_3 n$ (Section 2.6), we have $\sum_{v \in V} (d_v + d_v \cdot \log_3 n + \log_3 n) = O(n \log n)$. □

Lemma 8. *Given a tree $T = (V, E)$ with n nodes,* `algoHD` *sends n messages of $\log_3 n + 4$ bits each.*

Proof. Each node $v \in V$ sends $(int, vect)$ corresponding to the minimal hierarchical decomposition $MHD(T_v)$ of T_v to its parent. We know from Theorem 2 that $L(vect) \leq \log_3 n$, and from Property 2 that $vect$ contains only 0 and 1's. We transmit a vector $vect'$ and two bits ab to indicate the value of $(int, vect)$. Recall that it is not necessary to indicate the first value of $vect$ because there does not exist an unstable tree with node search number 1. We have four different codes:

(1) if $ab = 00$, then $int := -1$ and $vect := vect'$;

(2) if $ab = 01$, then $int := 1$ and $vect := vect'$;

(3) if $ab = 10$, then $int := i$ where $i > 1$ is the first cell with a 1 in $vect'$. Thus $\forall j \neq i, vect[j] := vect'[j]$ and $vect[i] := 0$;

(4) otherwise ($ab = 11$), the subtree is stable with $int := i$ where $i > 1$ is the unique cell with a 1 in $vect'$. If no such i exists, then $int = 1$.

We have 2 bits for ab and $\log_3 n - 1$ bits for $vect'$. We add 3 additional bits xyz as message prefix to indicate that the size of the tree is known ($x = 1$) and that the current algorithm is `algoHD` ($y = 1$ and $z = 0$). See Section 5.3 for a precise description of these bits.

If two nodes are potential candidates to be the root of the tree, the node with largest identifier is choosen, as explained in Remark 9. □

In this Section, we described a distributed algorithm to compute the node search number in trees. This algorithm allows for the design of a dynamic version to compute this parameter, as will be explained in next Section.

4 Dynamic and Incremental Algorithms

In this Section, we propose a dynamic algorithm that allows to compute the node search number for the tree resulting of the addition of an edge between two trees. It also allows to delete any edge. The efficiency of the algorithm relies on the main advantage of the hierarchical decomposition: the possibility to change the root (Lemma 9 of Section 4.1). Using this, we design an incremental algorithm which computes the node search number of a tree for which edges are added sequentially and in any order.

Clearly, joining two trees by adding an edge between their roots can be done directly using Algorithm 2, but for other cases, a preprocessing is needed to change the root of the trees. In this Section we propose such a preprocessing scheme. To apply this algorithm, we assume that each node v stores the information received from each of its $d(v) - 1$ neighbors ($d(v)$ if v is the root) and a vector $vect_{sum}$ which is the sum of the received vectors.

We now describe three functions that we will use in the dynamic version of `algoHD` before describing an incremental algorithm. We denote by D the *diameter* of a tree T. The number of *steps* of these functions corresponds to the number of nodes that have to perform computations.

4.1 Functions for Updating the Node Search Number

Lemma 9 (Change of the root)**.** *Given a tree $T = (V, E)$ of diameter D rooted at node $r_1 \in V$, and its minimal hierarchical decomposition $MHD(T_{r_1})$, we can choose a new root $r_2 \in V$ and update the minimal hierarchical decomposition $MHD(T_{r_2})$ in $O(D)$ steps of time complexity $O(\log n)$ each, using $O(D)$ messages of $\log_3 n + 4$ bits each.*

Proof. We describe an algorithm to change the root from r_1 to r_2.

First, r_2 sends a message to r_1 through the unique path between r_2 and r_1, ($r_2 = u_0, u_1, u_2, \ldots, u_k = r_1$), to notify the change. Then, r_1 computes the minimal hierarchical decomposition $MHD(T_{r_1})$ of T_{r_1}, considering that u_{k-1} is its parent and applies Algorithm 2 using $vect^{sum}_{r_1} - vect_{u_{k-1}}$ and all integers previously received but $int_{u_{k-1}}$. Then it sends a message to u_{k-1}.

Afterwards, u_{k-1} computes the minimal hierarchical decomposition $MHD(T_{u_{k-1}})$ of the subtree $T_{u_{k-1}}$ rooted at u_{k-1}, assuming that u_{k-2} is its parent. Then, u_{k-1} sends a message to u_{k-2}. We repeat it until r_2 receives a message from u_1. Finally, r_2 computes the node search number of T and becomes the new root. We have a new minimal hierarchical decomposition of T: $MHD(T_{r_2})$.

In this algorithm, u_i subtracts the vector $vect_{u_{i-1}}$ from $vect^{sum}_{u_i}$, and later adds $vect_{u_{i+1}}$, computes (p, p') corresponding to the merge of all stable trees of different minimal hierarchical decompositions (including the stable tree of $MHD(T_{u_{i+1}})$ and subtracting the stable tree of $MHD(T_{u_{i-1}})$) and finally applies Algorithm 2. Clearly, each such computation requires $O(\log n)$ operations. We add 3 bits xyz as message prefix to indicate that the size of the tree is known ($x = 1$), that the current algorithm is `IncHD` ($y = 0$), and whether the vector has to be added ($z = 1$) or subtracted ($z = 0$). □

Lemma 10 (Addition of an edge)**.** *Given two trees $T_{r_1} = (V_1, E_1)$ and $T_{r_2} = (V_2, E_2)$ respectively rooted at nodes r_1 and r_2 and whose minimal hierarchical decomposition are known by their respective root, we can add the edge $(w_1, w_2), w_1 \in V_1$ and $w_2 \in V_2$, and compute the node search number of $T = (V_1 \cup V_2, E_1 \cup E_2 \cup (w_1, w_2))$, in at most $O(D)$ steps where D is the diameter of T.*

Proof. First we change the roots of T_{r_1} and T_{r_2} respectively to w_1 and w_2 using Lemma 9. Then as described in Remark 9 one root out of w_1, w_2 is selected and computes the node search number of T. □

Lemma 11 (Deletion of an edge)**.** *Given a tree $T = (V, E)$ rooted at node r and an edge $(w_1, w_2) \in E$, after the deletion of (w_1, w_2), if r knows a minimal hierarchical decomposition of T_r, we can compute the node search number of the two disconnected trees in at most $O(D)$ steps.*

Proof. Without loss of generality we may assume that w_2 is the parent of w_1. Let T_{w_1} be the subtree rooted at w_1 and $T \setminus T_{w_1}$ be the tree rooted at r. Note that $T \setminus T_{w_1}$ includes w_2. While the node search number of T was computed by our algorithm, the minimal hierarchical decomposition $MHD(T_{w_1})$ of T_{w_1} was computed by node w_1. From this, $\mathtt{sn}(T_{w_1})$ can be computed using Lemma 4. Now, to compute $\mathtt{sn}(T \setminus T_{w_1})$, we apply the change root algorithm (Lemma 9) and node w_2 becomes the new root of $T \setminus T_{w_1}$. □

4.2 Incremental Algorithm

From Lemma 10, we obtain an incremental algorithm (`IncHD`) that, starting from a forest of n disconnected nodes with minimal hierarchical decomposition $(1, [\,])$, adds tree-edges one by one in any order and updates the node search number of each connected component. At the end, we obtain the node search number of T.

Although the average-case analysis is difficult, we can exhibit bad and good cases:

- bad case: T consists of two subtrees of size $n/3$ whose node search numbers are $\log_3(n/3)$, linked via a path of length $n/3$. The first edge to be inserted is located at the middle of the path. As explained in Lemma 10, the insertion of a new edge may force to change the root of the tree. By inserting new edges alternately at each opposite extremity of the already formed path, we force to change the root of the tree from one extremity of the path to the other one, and so to send messages from one side of the path to the opposite side. Once the path is formed, we insert edges alternately in each opposite subtree, thus imposing to change the root alternately in each subtree accordingly. Consequently, for each insertion of an edge of the tree, $O(D) = O(n)$ messages are exchanged, and so $O(n\ \mathtt{sn}(T)) = O(n \log(n))$ operations are required. Overall, `IncHD` requires $\Theta(n^2 \log n)$ operations;
- good case: any tree in which edges are inserted in the order induced by `algoHD` (inverse order of a breadth first search). `IncHD` needs an overall of $\Theta(n \log n)$ operations.

Actually, the overall number of messages is $O(nD)$ and the number of operations is $O(nD\mathtt{sn}(T))$. They are both strongly dependent on the edges' insertion order. Thus an interesting question is to determine the average number of messages and operations.

5 Improvements and Extensions

Our algorithms can be adapted to compute the process number (Section 2.4) or the edge search number (Section 2.5) of any tree with the same time complexity and transmission of information. For that, it is sufficient to change the values of the initial cases (lines 3 and 4) in Algorithm 3. Note that it gives the first polynomial algorithm to compute the process number in trees. Note also, that it is not possible to adapt our algorithm to weighted cases, since these later are NP-hard [16] (unless P=NP). We then show how to extend our algorithms to trees and forests of unknown size (Section 5.3).

For Section 5.1 and Section 5.2, we define a $(1,2)$-tree, which is particular tree used for the computation of the process number and the edge search number of trees.

Definition 8. *A tree T_v rooted at node v, with $\mathtt{pn}(T_v) = 1$ (respectively $\mathtt{es}(T_v) = 1$), is called a $(1,2)$-tree if and only if the smallest p such that a p-process strategy (respectively p-strategy) starting (or finishing) at v exists is $p = 2$.*

5.1 Computing the Process Number

Recall that the process number can be defined as the minimum number of agents to catch an invisible and fast fugitive in a graph in a cops and robber game (node search number) with the extra action that a node can be cleared when all its neighbors are occupied by an agent (Section 2.4). See [4–6] for more details. For example, a star has node search number 2 but has process number 1 as it can be cleared placing a single agent on its center. A path of length at least 4 requires 2 agents for the process number, as for the node search number.

Definition 9 (monotone p-process strategy)**.** *A p-process strategy is* monotone *if the unsafe part of the graph never grows. In other words, a node that has been processed can never host the fugitive again.*

Lemma 12. *For any graph G, there exists a monotone $\mathtt{pn}(G)$-process strategy for G.*

Proof. Let $G = (V, E)$ be a graph. We know from [5] that $\mathtt{sn}(G)-1 \leq \mathtt{pn}(G) \leq \mathtt{sn}(G)$. Clearly, Lemma 12 is true when $\mathtt{pn}(G) = \mathtt{sn}(G)$ as we can use the monotone $\mathtt{sn}(G)$-search strategy. When $\mathtt{pn}(G) = \mathtt{sn}(G)-1$, let us consider a non-monotone $\mathtt{pn}(G)$-process strategy and let $X \subseteq V$ be the set of nodes processed using the extra action (nodes with no agent on it). X is an independent set of G. Then, we build the graph $G' = (V', E')$, where $V' = V \setminus X$ and $E' = E \cup (\cup_{x \in X} \{(u_1, u_2),\ u_1, u_2 \in N(x) \text{ and } u_1 \neq u_2\}) \setminus (\cup_{x \in X} \{(u, x),\ u \in N(x)\})$. In other words, for each node $x \in X$, we create a clique between the neighbors $N(x)$ of x, and then we remove x from G and each incident edge. Now, from the $\mathtt{pn}(G)$-process strategy for G, we derive a $(\mathtt{sn}(G)-1)$-search strategy for G'. Recall that $\mathtt{pn}(G) = \mathtt{sn}(G)-1$. If we consider only the movements *'put an agent on a node $u \in V$'* and *'remove an agent from node $u \in V$'* of the process strategy for G, then the sequence of movements of the search strategy in G' is exactly the same than the sequence of movements of the process strategy in G. Since we have a $(\mathtt{sn}(G)-1)$-search strategy for G', then there exists a monotone $(\mathtt{sn}(G)-1)$-search strategy for G' [14]. Finally, from the movements of the agents of the monotone $(\mathtt{sn}(G)-1)$-search strategy for G', we deduce a monotone $\mathtt{pn}(G)$-process strategy for G.

Indeed the monotone $\mathtt{pn}(G)$-process strategy for G is composed of all the movements of the monotone $(\mathtt{sn}(G)-1)$-search strategy for G' plus $|X|$ steps that consist in processing each $x \in X$ without agent. Recall that any search strategy for a clique covers simultaneously all its nodes at some point. It means that node x can be processed when all its neighbors in G have an agent, that is when all the nodes of the corresponding clique in G' (forming by the neigbhors of x in G) have agents on them. Thus for each $x \in X$, we insert the processing step of x when the previous requirement is satisfied. □

Corollary 1. *If there exists a p-process strategy for a graph G, then there exists a monotone p-process strategy for G.*

Lemma 13. *Given a graph $G = (V, E)$ and a node $v \in V$, if there is a monotone p-process strategy starting at v, then there is a monotone p-process strategy finishing at v and vice-versa.*

Proof. Let S be a sequence of $x \leq 2|V|$ movements $m_1, m_2, \ldots, m_x$ describing a monotone p-process strategy for G finishing at v, that is finishing with the movement $m_x =$ *'remove an agent from node v'*. Note that, for the process number, a node $u \in V$ can be processed without an an agent on it. We denote

by V^{cov} the set of nodes with an agent during the p-process strategy defined by the sequence S, and we denote by $\overline{V^{cov}}$ the set of nodes that do not have an agent on them (i.e. that are processed using the extra action).

Let now $\overline{m}_i$ be the movement *'put an agent on a node $u \in V^{cov}$'* (respectively *'remove an agent from a node $u \in V^{cov}$'*) if m_i is the movement *'remove an agent from a node $u \in V^{cov}$* (respectively *'put an agent on a node $u \in V^{cov}$'*). Furthermore let $\overline{m}_i$ be the movement *'process a node $u \in \overline{V^{cov}}$'* if m_i is the movement *'process a node $u \in \overline{V^{cov}}$'*. Let $\overline{S}$ be the sequence of movements $\overline{m}_{\sigma(1)}, \overline{m}_{\sigma(2)}, \ldots, \overline{m}_{\sigma(x)}$, where $\sigma(i) = x - i + 1$.

We now prove that $\overline{S}$ gives a monotone p-process strategy starting at v, that is starting with the movement $\overline{m}_x =$ *'put an agent on node v'*. First note that if $\overline{S}$ gives a process strategy starting at v, then it is necessarily monotone. Indeed, in the obtained strategy, each node is processed only once. Furthermore if $\overline{S}$ is a process strategy, then it is a p-process strategy. Indeed if we label the p agents, then any node $u \in V^{cov}$ with agent i on it, $1 \leq i \leq p$, in S, also has agent i on it in $\overline{S}$.

Assume that $\overline{S}$ is not a process-strategy. There are four different cases:

A) in $\overline{S}$, there is an edge $(u, v) \in E$ such that *1)* we first put an agent at u, *2)* we then remove this agent from u, *3)* we then put an agent at v, *4)* and finally we remove the agent from v.

Note that these steps do not need to occur consecutively and that there are no more movements concerning u or v in $\overline{S}$ because of the correctness of the monotone process strategy S. Consequently it means that, in S, *1)* we first put an agent at v, *2)* we then remove this agent from v, *3)* we then put an agent at u, *4)* and finally we remove the agent from u. A contradiction because S is a valid p-process strategy.

B) in $\overline{S}$, there is an edge $(u, v) \in E$ such that *1)* we first put an agent at u, *2)* we then remove this agent from u, *3)* and we then clear v.

In the initial process strategy S, these movements correspond to *1)* process v, *2)* put an agent at u, *3)* and remove this agent from u. This is not a valid process strategy as we would clear v while one of its neighbors, u, is neither cleared (it will be processed later and we consider monotone process strategy) nor occupied. A contradiction because S is a valid p-process strategy.

C) in $\overline{S}$, there is an edge $(u, v) \in E$ such that *1)* we first clear v, *2)* we then put an agent at u, *3)* and we then remove this agent from u.

In S, it corresponds to *1)* put an agent at u, *2)* remove this agent from u, *3)* and then clear v. A contradiction because S is a valid p-process strategy.

D) in $\overline{S}$, there is an edge $(u, v) \in E$ such that *1)* we first clear v, *2)* and we then clear u.

In S, it corresponds to *1)* clear u, *2)* and then clear v. A contradiction because S is a valid p-process strategy. □

We now adapt previous definitions, properties, and lemmas for the computation of the process number.

Definition 10. *A tree T_v rooted at node v, with $\mathtt{pn}(T_v) = p$, is called a* stable tree *if there is a p-process strategy which finishes (starts) at node v.*

Definition 11. *A tree T_v rooted at node v, with $\mathtt{pn}(T_v) = p > 1$, is called an* unstable tree *if there exist two stable subtrees T_{w_1} and T_{w_2} (respectively rooted at two children of v: w_1 and w_2) such that $\mathtt{pn}(T_{w_1}) = \mathtt{pn}(T_{w_2}) = p$ and all other subtrees $T_{w_3}, \ldots, T_{w_j}$, respectively rooted at each other child of v: $w_3, \ldots, w_j$, are such that $\mathtt{pn}(T_{w_i}) < p$, $3 \leq i \leq j$.*

The definitions of stable (Definition 10) and unstable (Definition 11) trees for the process number are similar to Definitions 4 and 5 for the node search number. Notice that a $(1, 2)$-tree is neither stable nor unstable. For the process number, a $(1, 2)$-tree T_v rooted at v is a star with center $u \neq v$ (see Figure 8 for an example).

Property 3. *Given an unstable tree T_v rooted at node v, with $\mathtt{pn}(T_v) = p > 1$, there is no p-process strategy which finishes (starts) at node v.*

Proof. Let $w_1, \ldots, w_j$ be the children of v, and let T_{w_1} and T_{w_2} be the two stable subtrees such that $\mathtt{pn}(T_{w_1}) = \mathtt{sn}(T_{w_2}) = p$. Assume that there exists a p-process strategy for T_v starting at v. It gives naturally a p-process strategy for $T_{w_1} \cup T_{w_2} \cup \{v\}$ starting at v. Such a strategy begins to clear one of the two subtrees T_{w_1} and T_{w_2}, let say T_{w_1}. But while the p-process strategy clear T_{w_1}, by definition of a p-process strategy starting at v, it has to guarantee that the fugitive may not go to v, otherwise v would

if $int_{max} < 2$ **then**

$$(p, p') := \begin{cases} (0,0) & \text{when } int_{max} = -1 \\ (1,1) & \text{when } int_{max} = 0 \\ (1,2) & \text{when } |I| = 1 \text{ and } (p_i, p'_i) = (1,1), i \in I \\ (2,2) & \text{otherwise} \end{cases}$$

Figure 7: Initial cases for the process number (see Figure 8 for examples of computations), where int_{max} is defined in Algorithm 3, and (p_i, p'_i) are the new inputs of Algorithm 3, as explained in the proof of Theorem 16.

be recontaminated. This is impossible as we consider a monotone p-process strategy. Hence there must be an agent on v or on some nodes of T_{w_2}. Thus a process strategy starting at v needs at least $p+1$ agents.

By Lemma 13, we get the same result for a p-process strategy finishing at v. □

Lemma 14. *Given a tree $T = (V, E)$ and an unstable subtree T_v rooted at node $v \in V$, with $\mathtt{pn}(T_v) = p > 1$, then $\mathtt{pn}(T) = p$ if and only if $\mathtt{pn}(T \setminus T_v) \leq p - 1$.*

Proof. Let T_{w_1} and T_{w_2} be the two stable subtrees (respectively rooted at two children of v: w_1 and w_2) such that $\mathtt{pn}(T_{w_1}) = \mathtt{pn}(T_{w_2}) = p$.

If $\mathtt{pn}(T \setminus T_v) \geq p$, then v is a node with three branches, each having node search number at least p. If one of the branches has node search number greater than p then $\mathtt{sn}(T) \geq p + 1$. Otherwise, the three branches have process number p and, by Theorem 2, $\mathtt{pn}(T) = p + 1$.

Otherwise $\mathtt{pn}(T \setminus T_v) \leq p - 1$ and we describe a p-process strategy for T. We start by a p-process strategy for T_{w_1} finishing at w_1. It uses p agents and finishes with w_1 occupied by an agent. Then we place an agent on v and remove the one from w_1. We continue with a $(\leq p-1)$-process strategy for $T_v \setminus (T_{w_1} \cup T_{w_2} \cup \{v\})$. Now, since $\mathtt{pn}(T \setminus T_v) \leq p - 1$, we continue with a $(\leq p-1)$-process strategy for $T \setminus T_v$. We then place an agent on w_2 and remove the one from v. It now only remains to use a p-process strategy for T_{w_2} starting at w_2 which can be done with p agents by assumption. □

The definition of a hierarchical decomposition of a tree T_r rooted at node r (Definition 6) is slightly modified mainly as T^0 may be a $(1,2)$-tree.

Definition 12. *Given a tree T_r rooted at node r, a* process-hierarchical decomposition *of T_r, denoted by $HD(T_r)$, is a family of trees $\{T^i\}_{0 \leq i \leq k}$ such that:*

- *the set of the subtrees $\{T^i\}_{0 \leq i \leq k}$ forms a partition of the nodes of T_r;*
- *T^0 is either a stable, or a $(1,2)$-tree, or an unstable tree rooted at node $v_0 = r$;*
- *T^i is unstable and it is rooted at a node v_i, $1 \leq i \leq k$;*
- *if two trees T^i and T^j, $0 \leq i \leq k$, $0 \leq j \leq k$, $i \neq j$, are such that the path going from v_i to r goes through v_j, then $\mathtt{pn}(T^i) > \mathtt{pn}(T^j)$.*

With $HD(T_r)$ we associate the pair $((p, p'), vect)$ where $p = p'$ if T^0 is a stable tree ($p \geq 0$), $p = 1$ and $p' = 2$ if T^0 is a $(1,2)$-tree, and $p = p' = -1$ if T^0 is an unstable tree, and vect a vector of length $L(vect)$ where $L(vect)$ is the largest process number among the unstable trees of $HD(T_r)$. vect contains in cell i, denoted by $vect[i]$, the number of unstable trees of $HD(T_r)$ whose process numbers are i.

Definition 13 (minimal process-hierarchical decomposition)**.** *Given a tree T_r rooted at node r, a* minimal process-hierarchical decomposition *of T_r, denoted by $MHD(T_r)$, is a process-hierarchical decomposition of T_r (Definition 12) such that $\forall i, j \in [0, k]$, $i \neq j$, we have $\mathtt{pn}(T^i) \neq \mathtt{pn}(T^j)$.*

Lemma 15. *Given the representation $((p, p'), vect)$ of the minimal process-hierarchical decomposition $MHD(T_r)$ of a tree T_r rooted at node r, we have $\mathtt{pn}(T_r) = \max(p, L(vect))$.*

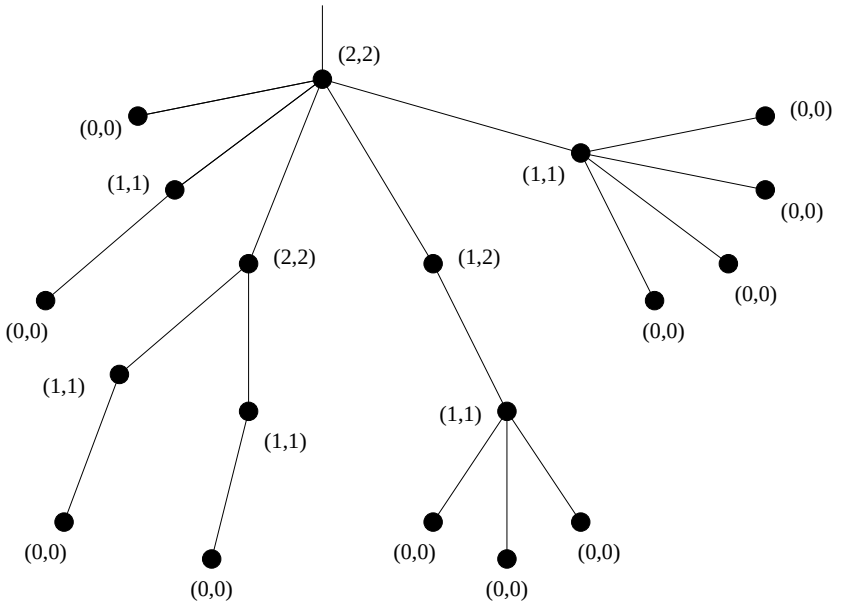

Figure 8: Examples of small cases for the process number. Pairs on nodes represent (p, p') for the corresponding subtrees.

Proof. Recall that $L(vect)$ is the largest process number among the unstable trees of $HD(T_r)$. If $p \geq L(vect)$, then $MHD(T_r)$ is composed of a single stable tree $T^0 = T_r$ and $\texttt{pn}(T_r) = p$.

If $p < L(vect)$, then we prove the assertion by induction on $L(vect)$. Since $MHD(T_r)$ is a minimal process-hierarchical decomposition of T_r, there is a unique unstable tree T^k such that $\texttt{pn}(T^k) = L(vect)$. Thus considering $MHD(T_r)$ minus tree T^k, we get a minimal process-hierarchical decomposition $MHD(T_r \setminus T^k)$ of $T_r \setminus T^k$. Hence, the length of the vector associated with $MHD(T_r \setminus T^k)$ has length strictly less than $L(vect)$. By induction hypothesis we have $\texttt{pn}(T_r \setminus T^k) < L(vect)$.

A $L(vect)$-process strategy for T_r is described as follows: start with a $L(vect)$-process strategy for T^k. There exists one which at some step has an agent on its root v_k and no other agent is located on another node. At this step include a $(\leq L(vect) - 1)$-process strategy for $T_r \setminus T^k$. Once it is done, finish the $L(vect)$-process strategy for T^k. □

We modify `algoHD` by using in Algorithm 2 and Algorithm 3 the initialization cases of Figure 7 to compute the process number of a tree $T = (V, E)$.

Theorem 16. `algoHD`*, using in Algorithm 2 and Algorithm 3 the initialization cases of Figure 7, computes the process number of a tree* $T = (V, E)$*.*

Proof. We first prove that Algorithm 3 modified returns the value of the union of all stable trees and all $(1, 2)$-trees. In this modified version, a node receives from its neighbors a pair $((p_i, p'_i), vect)$ instead of the pair $(int, vect)$.

We start the proof with the small cases (see Figure 8 for examples of computations):

A node $v \in V$, receiving pairs $(p_1, p'_1), \ldots, (p_{d-1}, p'_{d-1})$ from its neighbors $v_1, \ldots, v_{d-1}$, with $\forall i, p_i < 2$, computes the pair (p, p').

- If v is a leaf, it receives no message, and so $int_{max} = -1$. Then, with the initial cases of Figure 7 (line 1), the algorithm returns $(p, p') = (0, 0)$. This is correct since the process number of a single node is zero.
- If all neighbors sending information are leaves, then $int_{max} = 0$ and the modified algorithm returns $(p, p') = (1, 1)$ (line 2 of Figure 7). The process number of a star (with center v) is indeed one.
- If v receives a single message from a node which is the center of a star, then $|I| = 1$ and $(p_i, p'_i) = (1, 1), i \in I$. The modified algorithm returns $(p, p') = (1, 2)$ (line 3 of Figure 7). This is correct

if $int_{max} < 2$ **then**

$$(p, p') := \begin{cases} (1,1) & \text{when } |I| \leq 1 \\ (1,2) & \text{when } |I| = 2 \\ (2,2) & \text{otherwise} \end{cases}$$

Figure 9: Initial cases for the edge search number (see Figure 10 for examples of computations), where int_{max} is defined in Algorithm 3, and (p_i, p'_i) are the new inputs of Algorithm 3, as explained in the proof of Theorem 18.

since the process number of a star is one, but a search strategy finishing (starting) at v needs 2 agents.

- The last changes is when v receives information from a node which is in a star (but not the center) and from some leafs (with process number 0): $|I| = 1$ and $(p_i, p'_i) = (1,2), i \in I$; or when v receives information from at least two nodes with process number 1 and from some leafs: $|I| \geq 2$ and $p_i = 1, i \in I$. In both cases, due to the changes, the algorithm returns $(p, p') = (2,2)$ (line 4 of Figure 7). This is correct since 2 agents are needed, and are sufficient to finish (start) the strategy at v in these situations.

The rest of the proof of Algorithm 3 is unchanged as Theorem 2 is also valid for the process number.

Finally using previous modified Lemmas and Theorems, the proof of validity of `algoHD` can be adapted for the process number. □

5.2 Computing the Edge Search Number

Recall that the edge search number can be defined as the minimum number of agents to catch an invisible and fast fugitive in a graph in a cops and robber game (node search number). Here the fugitive can hide anywhere, including on an edge, and so an agent can slide along an edge (Section 2.5). For example, a path has edge search number 1 but has node search number 2. A star with at least three branches requires 2 agents for the edge search number, as for the node search number.

We now present the different changes of the previous definitions, lemmas, and theorems. For that, remark first that the edge search number of a path $(u_1, u_2, \ldots, u_k)$, $k \geq 3$, is 1 when the search strategy starts from an extremity (either u_1 or u_k), but it is 2 when it starts from any other vertex u_i, $2 \leq i \leq k-1$ (considering a monotone edge search strategy). The subtree attached to u_i is thus a $(1,2)$-tree. Consequently, we modify the definition of stable trees (Definition 4) by considering a $(1,2)$-tree as stable and with edge search number 2 unless the whole tree is a $(1,2)$-tree in which case the edge search number is 1. The definition of unstable trees (Definition 5) is unchanged. Property 1 is also unchanged. Lemma 3 can be adapted to Lemma 17. The proof explains how to clear edges linking v to $T \setminus T_v$. Note that, in `algoHD`, v is linked to $T \setminus T_v$ by a unique edge (by definition of `algoHD`).

Lemma 17. *Given a tree $T = (V, E)$ and an unstable subtree T_v rooted at node $v \in V$, with $\texttt{es}(T_v) = p > 1$, then $\texttt{es}(T) = p$ if and only if $\texttt{es}(T \setminus T_v) \leq p - 1$ (recall that a $(1,2)$-tree is considered as stable with edge search number 2).*

Proof. Let T_{w_1} and T_{w_2} be the two stable subtrees (respectively rooted at two children of v: w_1 and w_2) such that $\texttt{es}(T_{w_1}) = \texttt{es}(T_{w_2}) = p$.

If $\texttt{es}(T \setminus T_v) \geq p$, then v is a node with three branches, each having edge search number at least p. If one of the branches has edge search number greater than p then $\texttt{es}(T) \geq p + 1$. Otherwise, the three branches have edge search number p and, by Theorem 2, $\texttt{es}(T) = p + 1$.

Otherwise $\texttt{es}(T \setminus T_v) \leq p - 1$ and we describe a p-search strategy for T. We start by a p-search strategy for T_{w_1} finishing at w_1. It uses p agents and finishes with w_1 occupied by an agent. Then this agent slides along edge (w_1, v). We continue with a $(\leq p-1)$-search strategy for $T_v \setminus (T_{w_1} \cup T_{w_2} \cup \{v\})$. We assume that $T \setminus T_v$ is connected (otherwise we repeat sequentially the following strategy for the different trees). Thus there is a unique (v, u) such that $u \in T \setminus T_v$. Now, since $\texttt{es}(T \setminus T_v) \leq p - 1$, we continue with a $(\leq p-1)$-search strategy for $(T \setminus T_v)$ such that when an agent is located on u, we use an extra agent to clear the edge (u, v) and we remove it just after. It is always possible because if there is not extra agent, then either the strategy is accomplished or there are not enough agents. By assumption

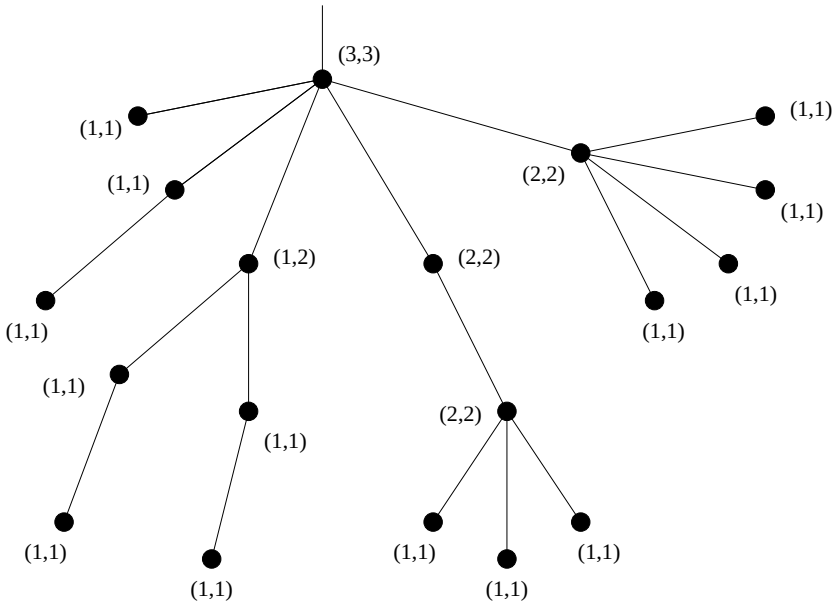

Figure 10: Examples of small cases for the edge search number. Pairs on nodes represent (p, p') for the corresponding subtrees.

there is one free agent when both u and v are occupied by agents, and so it is possible to clear edge (u, v), but if $T \setminus T_v$ is a $(1, 2)$-tree. This is a particular case because a single agent is used during the strategy for $T \setminus T_v$ and it cannot finish on node u (the root of $T \setminus T_v$). This is why this kind of tree is considered as stable tree with edge search number 2. To finish the description of the p-search strategy for T, the agent on v slides from v to w_2 along the edge (v, w_2). It now only remains to use a p-search strategy for T_{w_2} starting at w_2 which can be done with p agents by assumption. □

In the definition of a hierarchical decomposition of a tree T_r (Definition 6), T^0 is either a stable tree (including $(1, 2)$-tree considered as stable tree with edge search number 2) or an unstable tree. Other parts of this definition do not change. As for the process number and the node search number, with a hierarchical decomposition $HD(T_r)$ of T_r we associate $((p, p'), vect)$, where $p = p'$ if T^0 is a stable tree ($p \geq 1$), $p = 1$ and $p' = 2$ if T^0 is a $(1, 2)$-tree (considered as stable tree), or $p = p' = -1$ if T^0 is an unstable tree. Furthermore, the definition of the minimal hierarchical decomposition (Definition 7) does not change. Then Lemma 4 is modified replacing int by (p, p') and replacing $\max(int, L(vect))$ by $\max(p, L(vect))$.

We modify `algoHD` to compute the edge search number of a tree $T = (V, E)$.

Theorem 18. `algoHD` *by using in Algorithm 2 and Algorithm 3 initialization cases of Figure 9 plus the extra rules that all received pairs* $(1, 2)$ *are interpreted as if they were* $(2, 2)$ *and a node with* $(-1, -1)$ *cannot be in* I*, computes the edge search number of a tree* $T = (V, E)$*.*

Proof. The proof is similar to the proof of Theorem 6. We start to prove that Algorithm 3 is correct when using Figure 9 plus the extra rules described above. Recall that, as for the process number, the current nodes receives from its neighbors a pair $((p, p'), vect)$ instead of the pair $(int, vect)$.

We first consider the case when $int_{max} \leq 1$ (int_{max} corresponds to the int_{max} of Algorithm 3). It means that all received pairs are either $(0, 0)$ or $(1, 1)$. Indeed, recall that a received pair $(1, 2)$ is considered as $(2, 2)$. See Figure 10 for examples of computations.

- If the current node is a leaf, it receives no message, and then $I = \emptyset$. Algorithm 3 returns $(1, 1)$ which is correct since the tree consists in a single vertex.
- If $|I| = 1$, then T is a path finishing at the current node, a single agent is still sufficient to clear it, hence Algorithm 3 is correct as it returns $(1, 1)$.

- If $|I| = 2$, then T is a path and the current node is not a end vertex of this path. A single agent is sufficient to clear it, but a search strategy finishing at the current vertex needs two agents. Hence Algorithm 3 is correct as it returns $(1, 2)$.
- If $|I| \geq 3$, then T is a star with center the current node. Two agents are needed to clear it and sufficient for a search strategy finishing at the current vertex. Hence Algorithm 3 is correct as it returns $(2, 2)$.

When $int_{max} \geq 2$, since Theorem 2 is valid for the edge search number, the rest of the proof for Algorithm 3 is still valid.

For Algorithm 2, the use of modified lemmas and theorems completes the proof. □

5.3 Trees and Forests of Unknown Size

First of all, we prefix all messages with 3 bits xyz indicating which algorithm is currently used and if the size of the tree is known or not:

- $x = 1$ if the size of the tree is known and $x = 0$ otherwise;
- $y = 1$ for `algoHD` and $y = 0$ for `IncHD`;
- $z = 1$ for the addition of vectors in `IncHD`, and $z = 0$ for the subtraction of vectors in `IncHD`.

By setting bit z to 0 when `algoHD` is used ($y = 1$), we keep the prefix $xyz = 111$ for initialization purpose.

Now, when the size of the tree is unknown ($x = 0$), it remains to encode the rest of the message in order to detect its end. We can use simple rules to encode the pair $vect'$ used in the proof of Lemma 8: we replace bit 1 by the two bits 11 and we do not change bit 0. We add two bits to indicate ab as for the previous code. Furthermore we add the two bits 10 to indicate the end of the message. Thus the message is composed by (in order) xyz, ab, $vect'$, and 10. In this code the message requires at most $3 + 2 + 2L(vect') + 2$ bits. Since $L(vect') \leq \mathtt{sn}(T) - 1 \leq \log_3 n - 1$ (Lemma 8), the size of each message is up to $2\log_3 n + 5$ bits. Thus the receiver may decode the message without knowing n.

6 Conclusion

In this paper, we have presented the first distributed algorithm to compute the node search number in trees. This algorithm can be executed in an asynchronous environment, requires n steps, an overall computation time of $O(n \log n)$, and n messages of $\log_3 n + 4$ bits each. We have then proposed a distributed algorithm to update this graph invariant after addition or deletion of tree-edges. This second algorithm requires $O(D)$ steps, an overall computation time of $O(D \log n)$, and $O(D)$ messages of $\log_3 n + 4$ bits each, where D is the diameter of the modified connected component. From it, we have derived an incremental algorithm allowing to compute the node search number of trees for which edges are added sequentially and in any order. We have also shown how to adapt these algorithms to compute other graph invariants such as the process number and the edge search number, and how to extend our algorithms to trees and forests of unknown size using messages of size up to $2\log_3 n + 5$ bits.

In future work, we plan to extend further our algorithms on graphs with a shape similar to a tree. In particular, we plan to design distributed algorithms to compute the node search number of other classes of graphs such as trees of rings, and possibly outerplanar graphs.

Acknowledgments

We would like to thanks Philippe J. Giabbanelli, Nicolas Nisse and Hervé Rivano for fruitful discussions on this problem, and Carol and Dave Belski (Southwestern Cavers editors) and Richard Breisch for kindly provinding us a copy of [3].

This work has been partially supported by région PACA and ANR AGAPE.

References

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

[2] L. Barrière, P. Fraigniaud, N. Santoro, and D. M. Thilikos. Searching is not jumping. In Hans L. Bodlaender, editor, *29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 2880 of *Lecture Notes in Computer Science*, pages 34–45, Elspeet, The Netherlands, June 2003. Springer.

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

[4] D. Coudert, F. Huc, D. Mazauric, N. Nisse, and J-S. Sereni. Reconfiguration of the routing in WDM networks with two classes of services. In *13th Conference on Optical Network Design and Modeling (ONDM)*, Braunschweig, Germany, February 2009. IEEE.

[5] D. Coudert, S. Perennes, Q-C. Pham, and J-S. Sereni. Rerouting requests in WDM networks. In *7èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel'05)*, pages 17–20, Presqu'île de Giens, France, May 2005.

[6] D. Coudert and J-S. Sereni. Characterization of graphs and digraphs with small process number. *Discrete Applied Mathematics (DAM)*, 2011, to appear.

[7] J. Díaz, J. Petit, and M. Serna. A survey on graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

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

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

[10] F. V. Fomin and D. M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, 2008.

[11] P.A. Golovach. Search number, node search number, and vertex separator of a graph. *Vestn. Leningr. Univ., Math*, 24(1):8890, 1991.

[12] N. G. Kinnersley. The vertex separation number of a graph equals its pathwidth. *Information Processing Letters*, 42(6):345–350, 1992.

[13] M. Kirousis and C.H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, 47(2):205–218, 1986.

[14] A. S. LaPaugh. Recontamination does not help to search a graph. *J. Assoc. Comput. Mach.*, 40(2):224–245, 1993.

[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] R. Mihai and I. Todinca. Pathwidth is np-hard for weighted trees. In X. Deng, J. E. Hopcroft, and J. Xue, editors, *3rd International Workshop on Frontiers in Algorithmics (FAW)*, volume 5598 of *Lecture Notes in Computer Science*, pages 181–195, Hefei, China, June 2009. Springer.

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

[18] S.-L. Peng, C.-W. Hob, T.-S. Hsu, M.-T. Ko, and C. Y. Tanga. Edge and node searching problems on trees. *Theoretical Computer Science*, 240(2):429–446, June 2000.

[19] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *J. Combin. Theory Ser. B*, 35(1):39–61, 1983.

[20] P. Scheffler. A linear algorithm for the pathwidth of trees. In R. Henn R. Bodendiek, editor, *Topics in Combinatorics and Graph Theory*, pages 613–620. Physica-Verlag Heidelberg, 1990.

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