# Distributed Chasing of Network Intruders by Mobile Agents 

Lélia Blin, Pierre Fraigniaud, Nicolas Nisse, Sandrine Vial

## To cite this version:

Lélia Blin, Pierre Fraigniaud, Nicolas Nisse, Sandrine Vial. Distributed Chasing of Network Intruders by Mobile Agents. Proceedings of the 13th Colloquium on Structural Information and Communication Complexity (SIROCCO 2006), 2006, Chester, United Kingdom. pp.70-84, 10.1007/11780823_7. hal-00342000

HAL Id: hal-00342000

## https://hal.science/hal-00342000

Submitted on 23 Jul 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Distributed Chasing of Network Intruders 

Lélia Blin*<br>IBISC<br>University of Evry<br>91000 Evry<br>lelia.blin@lami.univ-evry.fr

Pierre Fraigniaud ${ }^{\dagger}$<br>LRI<br>CNRS \& University of Paris Sud<br>91405 Orsay, France<br>pierre@lri.fr<br>Sandrine Vial*<br>IBISC<br>University of Evry<br>91000 Evry<br>sandrine.vial@lami.univ-evry.fr

Nicolas Nisse ${ }^{\dagger}$
LRI
University of Paris Sud
91405 Orsay, France
nisse@lri.fr


#### Abstract

Graph searching is one of the most popular tool for analyzing the chase for a powerful and hostile software agent (called the "intruder"), by a set of software agents (called the "searchers") in a network. The existing solutions for the graph searching problem suffer however from a serious drawback: they are mostly centralized and assume a global synchronization mechanism for the searchers. In particular: (1) the search strategy for every network is computed based on the knowledge of the entire topology of the network, and (2) the moves of the searchers are controlled by a centralized mechanism that decides at every step which searcher has to move, and what movement it has to perform.

This paper addresses the graph searching problem in a distributed setting. We describe a distributed protocol that enables searchers with logarithmic size memory to clear any network, in a fully decentralized manner. The search strategy for the network in which the searchers are launched is computed online by the searchers themselves without knowing the topology of the network in advance. It performs in an asynchronous environment, i.e., it implements the necessary synchronization mechanism in a decentralized manner. In every network, our protocol performs a connected strategy using at most $k+1$ searchers, where $k$ is the minimum number of searchers required to clear the network in a monotone connected way using a strategy computed in the centralized and synchronous setting.


Keywords: graph searching, distributed algorithm, network security.

[^0]
## 1 Introduction

Graph searching [26] is one of the most popular tool for analyzing the chase for a powerful and hostile agent, by a set of software agents in a network. Roughly speaking, graph searching involves an intruder and a set of searchers, all moving from node to node along the links of a network. The intruder is powerful in the sense that it is supposed to move arbitrarily fast, and to be permanently aware of the positions of the searchers. However, the intruder cannot cross a node or an edge occupied by a searcher without being caught. Conversely, the searchers are unaware of the position of the intruder. They are aiming at surrounding the intruder in the network. The intruder is caught by the searchers when a searcher enters the node it occupies. For instance, one searcher can catch an intruder in a path (by moving from one extremity of the path to the other extremity), while two searchers are required to catch an intruder in a cycle (starting from the same node, the two searchers move in opposite directions). Another typical example is the $n$-node square mesh, in which $\Theta(\sqrt{n})$ searchers are necessary and sufficient for catching an intruder. In addition to network security, graph searching has several other practical motivations, such as rescuing speleologists in caves [8] or decontaminating a set of polluted pipes [27]. It has also several applications to the Graph Minor theory as it provides a dynamic approach to the analysis of static graph parameters such as treewidth and pathwidth [6].

The main question addressed by graph searching is: given a graph $G$, what is the search number of $G$ ? That is, what is the minimum number of searchers, $\mathrm{s}(G)$, required to clear the graph $G$, i.e., to capture the intruder? This question is motivated by, e.g., the need for consuming the minimum amount of computing resources of the network at any time, while clearing it. The decision problem corresponding to computing the search number of a graph is NP-hard [26], and NP-completeness follows from [7, 24]. Computing the search number is however polynomial for trees [25, 26], and the corresponding search strategy can be computed in linear time [30]. In fact, the search number of a graph is known to be roughly equal to the pathwidth, $p w$, of the graph, and therefore the search number of an $n$-node graph can be approximated in polynomial time, up to multiplicative factor $O(\log n \sqrt{\log t w})$ where $t w$ denotes the treewidth of the graph (see [14], and use the fact that $p w / t w \leq O(\log n)$ ).

The graph searching problem has given rise to a vast literature (cf. Section 1.2), in which several variants of the problem are discussed and solved. Nevertheless, from a distributed systems point of view, the existing solutions for the graph searching problem (cf., e.g., [25, 26, 30]) suffer from a serious drawback: they are mostly centralized. In particular, (1) the search strategy for every network is computed based on the knowledge of the entire topology of the network, and (2) the moves of the searchers are controlled by a centralized mechanism that decides at every step which searcher has to move, and what movement it has to perform. These two facts limit the applicability of the solutions. Indeed, as far as networking or speleology is concerned, the topology of the network is often unknown, or its map unprecise. The topology can even evolve with time (either slowly as for, e.g., Internet, or rapidly as for, e.g., P2P networks). Moreover, the mobile entities involved in the search strategy can hardly be controlled by a central mechanism dictating their actions. All these constraints make centralized algorithms inappropriate for many practical instances of the graph searching problem.

This paper addresses the graph searching problem in a distributed setting, that is the searchers must compute their own search strategy for the network in which they are currently running. This distributed computation must not require knowing the topology of the network in advance (not even its size), and the searchers must act in absence of any global synchronization mechanism, hence they must be able to perform in a fully asynchronous environment. Distributed strategies have been proposed for specific topologies only, such as trees [2], hypercubes [16], and rings and tori [15]. In this paper, we address the problem in arbitrary topologies.

The searchers are modeled by autonomous mobile computing entities with distinct IDs. More precisely, they are labeled from 1 to the current number $k$ of searchers in the network (if a new searcher has to join the team, it will take number $k+1$ ). Otherwise searchers are all identical, and run the same program. The network and the searchers are asynchronous in the sense that every action of a searcher takes a finite but unpredictable amount of time. Moreover, motivated by the fact that the intruder models a potentially hostile agent that can, e.g., corrupt the node memories, the search strategy must perform independently from any local information stored at nodes a priori, and even independently from the node IDs. We thus consider anonymous networks, i.e., networks in which nodes do not have labels, or these labels are not accessible to the searchers. The $\operatorname{deg}(u)$ edges incident to any node $u$ are labeled from 1 to $\operatorname{deg}(u)$, so that the searchers can distinguish the different edges incident to a node. These labels are called port numbers. Every node of the network has a whiteboard in which searchers can read, erase, and write symbols. (A whiteboard is modeling a specific zone of the local node memory that is reserved for the purpose of exchanging information between software agents). At every node, the local whiteboard is assumed to be accessible by the searchers in fair mutual exclusion. Since the content of the whiteboard at every node accessible by the intruder is corruptible, it is the role of the searchers to protect information stored at nodes' whiteboards.

The decisions taken by a searcher at a node (moving via port number $p$, writing the word $w$ on the whiteboard, etc.) is local and depends only on (1) the current state of the searcher, and (2) the content of the node's whiteboard (plus possibly (3) the incoming port number, if the searcher just entered the node).

The powerful intruder is assumed to be aware of the edge-labeled network topology, and thus it does not need the whiteboards to navigate. In fact, as mentioned before, when the intruder enters a node that is not occupied by a searcher, then it can modify or even remove the content of the local whiteboard.

All searchers start from the same node $u_{0}$, called the entrance of the network, or the homebase of the searchers. This node $u_{0}$ is also a source of searchers, in the sense that if the current team of searchers realize that they are not numerous enough for clearing the network, then they can ask for a new searcher, that will appear at the source. Initially, one searcher spontaneously appears at the source. The size of the team will increase until it becomes large enough to clear the network. Basically, the searchers are aiming at expanding a cleared zone around their homebase $u_{0}$, that is at expanding a connected sub-network of the network $G$, containing $u_{0}$, until the whole network is clear. In particular, as the entrance $u_{0}$ of the network is a critical node, it has to be permanently protected from the intruder in the sense that the intruder must never be able to access it.

Among all search strategies, monotone ones play an important role. A monotone strategy insures that, once an edge has been cleared, it will always remain clear. Monotone strategies guaranty a polynomial number of moves: exactly one move for clearing every edge, plus few moves required by the searchers to set up their positions before clearing the next edge. In the connected setting (i.e., the cleared part of the network is always connected), the corresponding graph searching parameter is called monotone connected search number starting at $u_{0}$ (cf., $[2,3,16,15,21])$, and is denoted by $\operatorname{mcs}\left(G, u_{0}\right)$.

### 1.1 Our results

We describe a distributed protocol, called dist_search, that enables the searchers to clear any asynchronous network in a fully decentralized manner, i.e., the search strategy is computed online by the searchers themselves, after being launched in the network without any information about its topology. This is the first distributed protocol that addresses the graph searching
problem in its whole generality, i.e., for arbitrary network topologies.
The distributed search strategy performed by the searchers in an asynchronous environment uses a number of searchers that is optimal up to a logarithmic factor. Indeed, we prove that the number of searchers involved in the strategy computed by our protocol in a network $G$ is equal to 1 plus the minimum number of searchers required to clear $G$ by a monotone connected search strategy starting at the homebase $u_{0} \in V(G)$, i.e., is equal to $\operatorname{mcs}\left(G, u_{0}\right)+1$. Since it is known [21] that, for any graph $G$ and for any $u_{0} \in V(G)$, we have $\operatorname{mcs}\left(G, u_{0}\right) \leq \mathrm{s}(G)\lceil\log n\rceil$, we get that our protocol uses at most $O(\log n)$ times the optimal number of searchers. In fact, it is conjectured that $\operatorname{mcs}(G) \leq 2 \mathrm{~s}(G)$ for all graph $G$ (cf. [3]). If this holds, then our protocol uses at most twice the optimal number of searchers.

Our protocol is space-efficient from many respects. First, it requires only $O(\log k)$ bits of memory for each of the $k$ searchers involved in the search. In particular, this amount of memory is independent from the size $n$ of the network. Second, the amount of information stored at every whiteboard never exceeds $O(m \log n)$ bits, where $m$ is the number of edges of the network.

To obtain our results, we had to address several problems.

- First, since the network is a priori unknown to the searchers, they have to explore it. However, this exploration cannot be achieved easily because of the potential corruption of the whiteboards by the intruder. Our protocol insures that exploration and searching are performed somehow simultaneously, and that the whiteboards of cleared nodes remain permanently protected unless there is no need to protect the stored information anymore.
- Second, as the searchers asynchronously spread out in the network, they become rapidly unaware of their relative positions. Our protocol synchronizes the searchers in a non trivial manner so that an action by a searcher is not ruined by the action of another searcher.
- Finally, to obtain space-efficient solutions, our protocol takes advantage from the accesses to the whiteboards, to store and read information useful to the searchers: it maintains a stack at every whiteboard, and every searcher at a node has access only to the top of a stack stored locally on the current node's whiteboard, and to few other variables also stored on the whiteboard.


### 1.2 Related Works

Graph searching, originated by Parson in [27], has been extensively studied in the literature (see [6] for a survey). Variants of the problem have been defined by Kirousis and Papadimitriou in $[22,23]$, and by Bienstock and Seymour in [7]. The notion of crusade allowed Bienstock and Seymour to simplify the proof of LaPaugh [24] about monotone graph searching: for any graph, there exists a minimal search strategy that is monotone (i.e., recontamination does not help). The notion of connected search strategy has been introduced by Barrière et al. [2, 3]. [2] describes a linear-time algorithm that computes minimal monotone connected search strategy for trees. [3] proves that, for any tree $T, \operatorname{mcs}(T) \leq 2 \mathrm{~s}(T)-2$ and this bound is tight. [31] shows that there exist graphs for which no minimal connected search strategies are monotone. On the other hand, [2] proves that recontamination does not help for connected search in trees.

Several protocols for clearing some specific networks in distributed setting have been proposed in the literature. Flocchini et al. have proposed protocols that address the graph searching problem in trees [2], hypercubes [16], tori and chordal rings [15]. For each of these classes of graphs, the authors have designed a protocol using $\operatorname{mcs}\left(G, u_{0}\right)+1$ searchers with $O(\log n)$ bits of memory and whiteboards of size $O(\log n)$ bits, that monotonously clears the graph in
polynomial time and polynomial number of moves. [16] proved that any distributed protocol clearing an asynchronous network in a monotone connected way requires $\operatorname{mcs}\left(G, u_{0}\right)+1$ searchers. Moreover, this remains true even if the topology of the network is known in advance.

Our problem is also very much related to graph exploration and mapping. In absence of whiteboards, it is known that network exploration is impossible using a finite team of finite automata [20, 29]. In fact, it is known that no finite team of finite automata is able to explore all graphs, even if these automata are given powerful communication facilities (cf., e.g., [10]). However, exploring trees is relatively easy [11], and a pre-computed labeling of the nodes with only three different labels enables just one finite automaton to explore all graphs [9]. In the recent paper of Reingold proving that $S L=L$ [28], a log-space constructible universal exploration sequence exploring all $d$-regular $n$-node graphs is described. Finally, $[4,5,19]$ investigated exploration of directed graphs.

In $[12,13]$, the objective of the authors is to determine the position of a blackhole in a network. A blackhole is an harmful node that destroys any agent visiting that node without letting any trace. On the other hand, the blackhole cannot move. [12, 13] have proved that $\Delta+1$ agents are necessary and sufficient to find a blackhole in any network, where $\Delta$ is the maximum degree of the network.

## 2 Model, Formal Statement, and Main Result

In this section, we specify our problem, and we state formally our main result.

### 2.1 Our problem

We summarize our problem setting. A network is an anonymous edge-labeled graph $G$. The $\operatorname{deg}(u)$ edges incident to any node $u$ are labeled by distinct integers from 1 to $\operatorname{deg}(u)$. These labels are called port numbers. A searcher is a mobile computing entity that can move along the edges of the network. At every node of the network, there is a whiteboard accessible to the searchers currently occupying this node. A whiteboard is a zone of the node's memory reserved to the searchers to read, write, and erase information. The access to every whiteboard is assumed to be performed under the control of a fair mutual exclusion mechanism. The decision taken by a searcher at a node depends on its internal state, the content of the local whiteboard, and the incoming port number. A decision results in either leaving the node through some port $p$, or waiting at the node until it has (again) access to the whiteboard. The searchers are generated by a unique node $u_{0} \in V$, called the homebase. The homebase is a source of searchers, in the following sense. New searchers can be generated at the homebase. For a new searcher to be generated, at least one searcher must be occupying the homebase, and calling for a new searcher. The $i$ th searcher generated at the homebase is given label $i$. The searchers are asynchronous in the sense that every action of a searcher takes a finite but unpredictable amount of time. When they are launched in a network, they ignore its topology, and have no information about it (they even ignore its size). The goal of the searchers is to capture an "intruder".

The intruder is a malicious mobile computing entity that can move along the edges of the network. The intruder is arbitrarily fast, and is assumed to be permanently aware of the positions of the searchers. It is invisible in the sense that the searchers are unaware of the position of the intruder. On the other hand, the intruder knows the topology of the network and is assumed to be permanently aware of the positions of the searchers. The intruder is caught if it meets a searcher at a node or along an edge. The intruder has the ability to corrupt the nodes, including the content of their whiteboards.

A distributed search protocol is a distributed program executed by the searchers for catching the intruder. Initially, one searcher is spontaneously generated at the homebase $u_{0}$, and the intruder can be placed at any node or edge of the network. The searcher can start moving in the network or calling for a new searcher. The execution of the search protocol results in a team of searchers moving in the network, looking for the intruder. A search protocol must perform independently from any a priori knowledge about the network. Hence, initially, the searchers ignore in which network they are running. On the other hand, the intruder is given a precise map of the edge-labeled network in which it has been placed, and it knows where in the network it has been placed. Again, all previous works [2, 16, 15] compute the search strategy from the entire knowledge of the network, and the strategy performs in synchronous steps. In our setting, the search strategy is computed by the searchers applying the search protocol, in absence of any a priori knowledge about the network, and in an asynchronous environment.

Clearly, $n+1$ searchers can easily capture the intruder in any $n$-node network (the team of searchers expand from the homebase until they occupy all the nodes, while one extra searcher "clear" all the edges). A search protocol is minimal if, for any network $G$, and for any node $u_{0} \in V(G)$, the number of searchers required by the protocol to capture the intruder in $G$ starting from the homebase $u_{0}$ is the smallest for this setting. This paper addresses the problem of designing a minimal distributed search protocol. This problem has been widely investigated in the literature in the framework of search games. In the general setting of search games, a search strategy for a network $G$ is an ordered sequence of search steps resulting in the intruder being caught, where each step is of one of the following three types:

1. place a searcher at a vertex $v \in V(G)$;
2. remove a searcher from a vertex $v \in V(G)$;
3. move a searcher along an edge $e \in E(G)$.

A $k$-search strategy is a search strategy in which at most $k$ searchers are present in the network at every step. The search number $\mathbf{s}(G)$ of a network $G$ is the smallest $k$ for which there exists a $k$-search strategy for $G$. Several search games as been defined in the literature [2, 3, 7, 22, 23, 31]. We consider the most realistic one as far as network security is concerned.

- A search strategy is internal if it does not contain any removal step. Internal search strategies are desired in communication networks since an agent cannot easily be placed at or removed from any node.
- A search strategy is monotone if it performs so that the intruder never occupies a node or an edge that has been previously visited by a searcher. Monotone search strategies are desired for they insure that the number of searcher moves is polynomial in the size of the network.
- A search strategy is connected if, at any step, the "cleared zone" of the network (i.e., the set of nodes and edges that has been cleared so far, and protected from recontamination by the intruder) is connected. Connected search strategies are desired because they insure that communications between the searchers can be performed without risk of corruption by the intruder. A connected strategy is obviously internal.
If there exists a monotone connected $k$-search strategy for the network $G$, then there exists such a strategy in which the $k$ searchers are initially placed at a same node, and all steps consist in moving searchers along the edges of the network (cf., e.g., [3]). In the following, all our strategies are connected. Given a network $G$, and a node $u_{0} \in V(G)$, the smallest $k$ for which there exists a monotone connected $k$-search strategy for $G$ where all searchers are initially placed at $u_{0}$ is denoted by $\operatorname{mcs}\left(G, u_{0}\right)$.


### 2.2 Main Results

Our main result is the design of a provably distributed search protocol, dist_search, for a team of searchers as defined in Section 2.1. The performances of our protocol are compared to the ones of monotone connected search strategies. The following theorem summarizes the main characteristics of dist_search.

Theorem 1 For any connected, asynchronous, and anonymous network $G$, and any $u_{0} \in V(G)$, dist_search enables capturing an intruder in $G$ using searchers, in a connected way, starting from the homebase $u_{0}$, and initially unaware of $G$. The main characteristics of dist_search are the following:

- dist_search uses at most $k=\operatorname{mcs}\left(G, u_{0}\right)+1$ searchers if $\operatorname{mcs}\left(G, u_{0}\right)>1$, and $k=1$ searcher if $\operatorname{mcs}\left(G, u_{0}\right)=1$;
- Every searcher involved in the search strategy computed by dist_search uses $O(\log k)$ bits of memory, where $k$ is the number of searchers;
- During the execution of dist_search, at most $O(m(\log k+\log \Delta))$ bits of information are stored at every whiteboard, where $\Delta$ and $m$ are respectively the mawimum degree and the number of edges of $G$, and $k$ is the number of searchers.


## Remarks.

- Note that the theorem above implies that, for networks searchable by a monotone connected search strategy using a constant number of searchers, the protocol dist_search can be implemented using finite state automata.
- The strategy performed by the searchers is connected but not necesseraly monotone. However, it is easy to check that, once the whole graph has been cleared by searchers applying dist_search, the description of a search strategy $S$ is stored in a distributed way on the nodes' whiteboards of $G . S$ is a monotone connected search strategy for $G$, starting from $u_{0}$, and using at most $\operatorname{mcs}\left(G, u_{0}\right)+1$ searchers. Moreover, an automaton with at most $O(\log n)$ bits of memory can collect $S$, assuming that no intruders can corrupt the information on the graph while $S$ is collected.
- Note also that the search strategy $S$ computed by protocol dist_search is optimal in the following sense. For any $k \geq 1$, there exists a network $G$ and $u_{0} \in V(G)$ such that, $k=\operatorname{mcs}\left(G, u_{0}\right)$ and for any distributed protocol $\mathcal{P}$ designed for capturing a fugitive in a monotone connected way, starting from $u_{0}, \mathcal{P}$ requires $k+1$ searchers [16].


### 2.3 Sketch of Protocol dist_search and of its proof

Given a connected network $G$, and $X \subseteq E(G)$, we denote by $\delta(X)$ the nodes in $V(G)$ that are incident to an edge in $X$ and an edge in $E(G) \backslash X$. Given $k \geq 1$, we call $k$-configuration any set $X \subseteq E(G)$ such that $|\delta(X)| \leq k$. The $k$-configuration digraph $\mathcal{C}_{k}$ of $G$ is defined as follows. $V\left(\mathcal{C}_{k}\right)$ is the set of all possible $k$-configurations. There is an arc from $X$ to $X^{\prime}$ in $\mathcal{C}_{k}$ if the configuration $X^{\prime}$ can be reached from $X$ by one step (i.e., place, move or remove a searcher) of a monotone connected search strategy using at most $k$ searchers. The objective of Protocol dist_search is essentially to try, for successive $k=1,2, \ldots$, whether the configuration graph $\mathcal{C}_{k}$ can be traversed from $\emptyset$ to $E(G)$ under the constraint that the searchers starts at $u_{0}$. If yes, then dist_search completes after having captured the intruder using at most $k$ searchers. Otherwise, dist_search tries with $k+1$ searchers.

Remark. This approach is similar to the (centralized) parametrized algorithms of the literature (cf., e.g., $[1,17,18]$ ). However, the difficulty of our approach is to discover whether the configuration digraph $\mathcal{C}_{k}$ can be traversed from $\emptyset$ to $E(G)$ in a decentralized manner.

For a fixed $k$, the objective of dist_search is to organize the movements of the searchers so that they perform a DFS of $\mathcal{C}_{k}$ (again, ignoring the topology of $G$, and in an asynchronous environment). This objective is achieved according to an order specified by a virtual stack in which are stored information related to the moves of the searchers. Roughly, Protocol dist_search constructs all possible states for the virtual stack, according to a lexicographic order on the states of the stack. The difficulty of the protocol is to distribute the virtual stack on the whiteboards so that when a searcher visits a node, it finds on the whiteboard enough information for computing the next step of the search strategy that it should perform. Since the intruder can corrupt the whiteboards, withdrawals from previously visited nodes must be scheduled so that to make sure that no information will be lost. Note here that, albeit the search strategy eventually computed by the searchers is monotone (in the sense that the contents of all the whiteboards describe a monotone search strategy when the protocol completes), failing search strategies investigated before (according to the lexicographic order on the states of the virtual stack) lead to withdrawals, and therefore to recontamination. If all strategies with $k$ searchers have failed, then the searchers terminate at the homebase, call a new searcher, and restart searching the network with $k+1$ searchers.

The additional searcher used by dist_search, compared to $\operatorname{mcs}\left(G, u_{0}\right)$, is used for avoiding deadlocks such as the one described in [16]. It is also used to schedule the moves of the other searchers and to transmit information between the searchers. It could be replaced by simple communication facilities. For instance, if the searchers would have the ability to send to and read from a mailbox available at the homebase, this additional searcher could be avoided. In particular, in the Internet, each searcher would just have to keep in its memory the IP address of the homebase.

The proof of correctness of Protocol dist_search is twofold. First, we prove the correctness of an algorithm, denoted by $\mathcal{A}$, that uses a centralized stack for traversing the configuration digraph $\mathcal{C}_{k}$. The second part of the proof consists in proving a one-to-one correspondence between every execution of dist_search using a virtual (i.e., decentralized) stack, and every execution of $\mathcal{A}$ using a centralized stack.

## 3 Search strategy using a centralized stack

In this section, we describe the algorithm $\mathcal{A}$ enabling a team of searchers launched in an unknown network to capture an intruder hidden in this network. Algorithm $\mathcal{A}$ is not fully distributed because it uses a centralized stack whose top is accessible from every node by every searchers.

### 3.1 Algorithm $\mathcal{A}$

Algorithm $\mathcal{A}$ uses the notion of extended moves, that are triples ( $a_{i}, a_{j}, p$ ) where $a_{i}$ and $a_{j}$ denote searchers, and $p$ is a port number.

Definition $1 A n$ extended move ( $a_{i}, a_{j}, p$ ) corresponds to the following: (1) searcher $a_{i}$ joins searcher $a_{j}$, and (2) the searcher with the smallest ID among $a_{i}$ and $a_{j}$ leaves the node now occupied by the two searchers via port $p$. (Note that $i=j$ is allowed, in which case $a_{i}$ leaves the node it occupies by port p).

The central stack stores extended moves and thus describes a sequence of operations performed by the searchers. More precisely, reading the stack bottom-up defines a sequence of operations that describes a partial execution of a search strategy.

Definition 2 For a fix parameter $k \geq 1$, a state of the virtual stack is valid if there exists a monotone connected search strategy using at most $k$ searchers whose partial execution is described by this state.

By some abuse of terminology, we sometime say that a stack $Q$ is valid, meaning that the current state $S$ of the stack $Q$ is valid. Given a valid state $S$ of a stack $Q$, we denote by $X_{S}$ the configuration induced by $S$, that is $X_{S}$ is the set of clear edges after the execution of the extended moves in $S$.

The principle of Algorithm $\mathcal{A}$ is the same as the one described in Section 2.3. That is, it tries, for each $k=1,2, \ldots$, every possible monotone connected search strategy using $k$ searchers, until one reaches a situation in which either the whole network is clear, or all search strategies have been exhausted. In the latter case, Algorithm $\mathcal{A}$ proceeds with $k+1$ searchers by calling for a new searcher at the homebase $u_{0}$. From now on, we assume that $k$ is fixed. The $k$ searchers are denoted by $a_{1}, \ldots, a_{k}$, where the ID of $a_{i}$ is simply its index $i$. Algorithm $\mathcal{A}$ is described in Figure 1. We detail its structure. Algorithm $\mathcal{A}$ returns a boolean possible. If possible is true then clearing the network with $k$ searchers is possible, in which case the stack $Q$ returned by Algorithm $\mathcal{A}$ is valid, and contains a monotone connected search strategy clearing $G$ with $k$ searchers.

In Algorithm $\mathcal{A}$, the stack $Q$ is initially empty, and only $a_{1}$ is placed at $u_{0}$. the other searchers $a_{2}, \ldots, a_{k}$ are available. In addition to the centralized stack $Q$, Algorithm $\mathcal{A}$ uses a global variable state that takes two possible values CLEAR or BACKTRACK whose meaning will appear clear later on. Finally, Algorithm $\mathcal{A}$ uses a boolean variable decided that is false until either a monotone connected search strategy using $k$ searchers clearing the network is discovered, or all possible monotone connected search strategies using $k$ searchers have been considered. Hence the main while-loop of Algorithm $\mathcal{A}$ is based on the value of decided (cf. Figure 1). This main while-loop mainly contains two blocks of instructions. These blocks are executed depending on the value of state (CLEAR or BACKTRACK).

The algorithm enters one of these two blocks unless all searchers are available, in which case a search strategy has been found. Initially, $a_{1}$ is placed at $u_{0}$ and is thus not available. Case clear corresponds to a situation in which Algorithm $\mathcal{A}$ has just cleared an edge, i.e., the last execution of the main while-loop has resulted in pushing some extended move in $Q$. Case BACKTRACK corresponds to a situation when the last execution of main while-loop has resulted in popping the stack $Q$, i.e., in recontaminating an edge.

Let us focus on the case state $=$ clear. Algorithm $\mathcal{A}$ focuses on specific extended moves, only those that do not imply recontamination (this is because $\mathcal{A}$ eventually computes a monotone strategy). More formally, let us consider a valid state $S$ of the stack $Q$, i.e., $S$ is a sequence of extended moves denoted by $M_{1}|\ldots| M_{r}$. Pushing an extended move $M$ in $Q$ results in a new state, denoted by $S \mid M$. We say that a extended move $M$ is valid according to $Q$ if $S^{\prime}=S \mid M$ is a valid state. Note that $\mathcal{A}$ does not maintain the set $X$ of clear edges and the set of available searchers. Indeed, given a valid state $S$ of the stack $Q$, one can easily construct $X_{S}$ by executing the partial search strategy described by $S$. A searcher is then available if either it stands at a node not in $\delta\left(X_{S}\right)$ or it stands at a node occupied by another searcher, of lower index. There is therefore a simple characterization of a valid extended move $M$ according to a valid state $S$ of $Q$ :

- If $S=\emptyset$, then $M$ is valid according to $Q$ if and only if either $u_{0}$ is a 1 -degree node and $M=\left(a_{1}, a_{1}, 1\right)$, or $k>1$ and $M=\left(a_{2}, a_{1}, 1\right)$.
- If $S \neq \emptyset, M=\left(a_{i}, a_{j}, p\right)$ is valid according to $Q$ if and only if either $i=j, a_{i}$ stands at a node $u \in \delta\left(X_{S}\right)$, and $p$ is the only contaminated port of node $u$, or $i \neq j, a_{i}$ is available, $a_{j}$ stands at a node $u \in \delta\left(X_{S}\right)$, and $p$ is a contaminated port of node $u$.

The first instruction of the case state = CLEAR consists in checking whether there exists a valid extended move according to $Q$. The key issue is to choose which extended move to apply, among all possible valid extended moves. For this choice, the extended moves are ordered in lexicographic order.

Definition 3 Let $M=\left(a_{i}, a_{j}, p\right)$ and $M^{\prime}=\left(a_{i^{\prime}}, a_{j^{\prime}}, p^{\prime}\right)$ be two extended moves. We define $M \prec M^{\prime}$ if and only if either $\left(i<i^{\prime}\right)$, or ( $i=i^{\prime}$, and $\left.j<j^{\prime}\right)$, or $\left(i=i^{\prime}, j=j^{\prime}\right.$, and $\left.p<p^{\prime}\right)$.

If there is an extended move that is valid according to $Q$ then Algorithm $\mathcal{A}$ chooses the one that has minimum lexicographic order among all extended moves that are valid according to $Q$. If there is no extended moves that are valid according to $Q$, then $\mathcal{A}$ switches to the state backtrack. For this purpose, the last move in $Q$ is popped out, and stored in the global variable $M_{\text {last }}$. If fact, if $Q=\emptyset$, then backtracking is not possible, and $\mathcal{A}$ decides that $k$ searchers are not sufficient to clear the network.

Let us now focus on the case state $=$ backtrack. $\mathcal{A}$ considers the move $M_{\text {last }}$. If there is an extend move $M \succ M_{\text {last }}$ that is valid according to the stack, then $\mathcal{A}$ performs the smallest such move by pushing $M$ in the stack, and going back to state clear. Otherwise $\mathcal{A}$ carries on backtracking by popping out the last extended move from the stack.

### 3.2 Proof of correctness of Algorithm $\mathcal{A}$

Lemma 1 After any execution of the while-loop in Algorithm $\mathcal{A}$, the state of the stack is valid.
Proof. Initially, the stack is empty, corresponding to the strategy in which $a_{1}$ is occupying node $u_{0}$, and hence is valid. Assume that the state of $Q$ before executing the while-loop is valid, and consider the state of $Q$ after the loop. Independently from whether state = CLEAR or state $=$ BACKTRACK, there are two cases depending on a push or a pop is performed. The result of the push is a valid state because only extended moves that are valid according to $Q$ are pushed in $Q$. The result of the pop is also valid state since it corresponds to the partial search strategy described by $Q$ before the loop, in which the last extended move is removed.

The next lemma requires to order the states of the stack, the same way, we ordered extended moves.

Definition 4 Given two states of the stack $Q, S=M_{1}|\cdots| M_{r}$ and $S^{\prime}=M_{1}^{\prime}|\cdots| M_{r^{\prime}}^{\prime}$, we define $S \prec S^{\prime}$ if and only if there exists $i \leq \min \left\{r, r^{\prime}\right\}$ such that $M_{i} \prec M_{i}^{\prime}$ and, for any $j<i, M_{j}=M_{j}^{\prime}$.

The order on the stacks defined above is a total order. Since the extended move pushed in the stack in the case clear of Algorithm $\mathcal{A}$ is the minimum extended move according to the current state of the stack, we get that the sequence of stacks constructed by Algorithm $\mathcal{A}$ respects this total order. Precisely, we have:

Lemma 2 All valid states constructed by Algorithm $\mathcal{A}$ are compatible with the total order of Definition 4, in the sense that if $r$ is the first execution of the while loop at which some state $S$ appears, then all valid states $S^{\prime} \prec S$ appeared before, and no valid state $S^{\prime \prime} \succ S$ appeared before.

We say that a valid sequence of extended moves is complete if the corresponding search strategy clears the whole network. The following is a direct consequence of Lemma 2

## - The Algorithm $\mathcal{A}$ -

Input: $k \geq 1$ searchers $a_{1}, a_{2}, \cdots, a_{k}$ and a node $u_{0}$ of a graph $G$.
$a_{1}$ is placed at $u_{0} ; a_{2}, \cdots, a_{k}$ are available.
Output: a boolean possible, and a stack $Q$ of extended moves.
begin
$Q \leftarrow \emptyset ;$
state $\leftarrow$ CLEAR;
decided $\leftarrow$ false;
while not decided do
if all searchers are available then
decided $\leftarrow$ true;
possible $\leftarrow$ true;
else
/* case state $=$ CLEAR */
if state $=$ CLEAR then
if there exists a valid extended move according to $Q$ then
$\left(a_{i}, a_{j}, p\right) \leftarrow$ minimum valid extended move according to $Q$;
push $\left(a_{i}, a_{j}, p\right)$;
else
if $Q \neq \emptyset$ then
$M_{\text {last }} \leftarrow \operatorname{pop}()$;
state $\leftarrow$ BACKTRACK;
else
decided $\leftarrow$ true;
possible $\leftarrow$ false;
endif
endif
/* case state $=$ BACKTRACK */
else
Let $M_{\text {last }}=\left(a_{i}, a_{j}, p\right)$;
if there exists a valid extended move according to $Q$ larger than $\left(a_{i}, a_{j}, p\right)$ then
$\left(a_{i}^{\prime}, a_{j}^{\prime}, p^{\prime}\right) \leftarrow$ minimum valid extended move according to $Q$, and larger than $\left(a_{i}, a_{j}, p\right)$;
push $\left(a_{i}^{\prime}, a_{j}^{\prime}, p^{\prime}\right)$;
state $\leftarrow$ CLEAR;
else
if $Q \neq \emptyset$ then $M_{\text {last }} \leftarrow \operatorname{pop}()$;
else
decided $\leftarrow$ true;
possible $\leftarrow$ false;
endif
endif
endif
endif
endwhile
return(possible, $Q$ );
end.

Figure 1: The Algorithm $\mathcal{A}$

Lemma 3 Let $S=M_{1}|\ldots| M_{r}$ be a sequence of extended moves corresponding to a partial execution of a search strategy using at most $k$ searchers. Either there exists a complete sequence $S^{\prime}$ of extended moves with $S^{\prime} \prec S$, or Algorithm $\mathcal{A}$ eventually computes state $S$ of the stack.

Lemma 4 If $\operatorname{mcs}\left(G, u_{0}\right)>k$ then Algorithm $\mathcal{A}$ returns (false, $\emptyset$ ) for $k$.
Proof. Let $S$ be the maximum valid sequence of extended moves according to the order of Definition 4. Since the graph cannot be cleared using $k$ searchers, starting from $u_{0}$, we get that, for any valid sequence $S^{\prime} \preceq S, S^{\prime}$ is not complete. By Lemma 3, Algorithm $\mathcal{A}$ eventually computes the state $S$ of the stack. After that, the algorithm always remains in the case backtrack and it successively pops all extended moves out of the stack. Thus, we reach the situation where $Q=\emptyset$ and there is no more valid extended move. Thus, Algorithm $\mathcal{A}$ returns (false, $\emptyset$ ).

Lemma 5 Assume $\operatorname{mcs}\left(G, u_{0}\right)=k$. Let $S$ be the smallest complete sequence of valid extended moves corresponding to a monotone connected search strategy starting from $u_{0}$. Algorithm $\mathcal{A}$ returns (true, $Q$ ) for $k$, where $Q$ is in state $S$.

Proof. By Lemma 3, since $S$ is the smallest complete sequence of valid extended moves, Algorithm $\mathcal{A}$ computes $S$. At this step of Algorithm $\mathcal{A}$, all nodes of the graph are clean. Thus, all the searchers are available, and therefore Algorithm $\mathcal{A}$ returns (true, $Q$ ).

As a direct consequence of the previous lemmas, we get:
Theorem 2 Algorithm $\mathcal{A}$ completes for $k=\operatorname{mcs}\left(G, u_{0}\right)$, and then the stack $Q$ describes a monotone connected search strategy starting at $u_{0}$ and using $k$ searchers.

## 4 Fully Distributed Search Strategy

In this section, we describe the main features of protocol dist_search. In this description, we assume that searchers are able to communicate by exchanging messages of size $O(\log k)$ bits where $k$ is the number of searchers currently involved in the search. With this facility, we will show that dist_search captures the intruder with $\operatorname{mcs}\left(G, u_{0}\right)$ searchers. Using an additional searcher for implementing the communications between the $\operatorname{mcs}\left(G, u_{0}\right)$ other searchers, dist_search captures the intruder with $\operatorname{mcs}\left(G, u_{0}\right)+1$ searchers. Assuming that the searchers can communicate by exchanging messages is only for the purpose of simplifying the presentation. The fact that an additionnal searcher can implement the communications between searchers will appear clear while describing the protocol dist_search. The main reasons for which this can be done is that finding its way in the clear part of the network is easy thanks to the information stored on the whiteboards. The sender of a message is always the searcher that has performed the last action, and an action is always the result of a reception of a message.

Moreover, for the sake of simplicity, we assume that two searchers on the same node can "see" each other and exchange their states. This is not a restrictive assumption since this can be implemented with the whiteboards, but it would unnecessarily complicate the presentation.

First, we describe the data structure used by dist_search.

### 4.1 Data Structure of dist_search

Every searcher has a local state variable that can take $k+2$ different values where $k$ is the current number of searchers. These $k+2$ states are: Clear, backtrack, and (help, $j$ ), for $j=1, \ldots, k$. Initially, all searchers are in state clear. During the execution of the protocol, we have:

- a searcher is in state clear if it has just cleared an edge;
- a searcher is in state BACKTRACK if it has just backtracked through an edge that it has previously cleared;
- a searcher is in state (HELP, $j$ ) if it is aiming at joining searcher $j$ to help him clearing the network (i.e., one of the two searchers will guard a node, while the other will clear an edge incident to this node).

The messages that searchers can exchange are of four types: start, move, help and sorry.

- start is an initialization message, that is only used to start Protocol dist_search (only searcher $a_{1}$ receives this message, at the very beginning of the protocole execution).
- If a searcher $i$ receives a message (move, $j$ ) from searcher $a_{j}$, then it is the turn of searcher $a_{i}$ to proceed. (As it should appear clear later, the searchers schedule themselves so that exactly one searcher performs an action at a time).
- If a searcher $a_{i}$ receives a message (help,$j$ ) from searcher $a_{j}$, then $a_{j}$ is currently just arriving at the same node as $a_{i}$ to help $a_{i}$. (Note that $a_{i}$ and $a_{j}$ could use the whiteboard to communicate, and this type of messages is just used for a purpose of unification with the other message types).
- If a searcher $a_{i}$ had received a message (move, $j$ ) or (help, $j$ ) from searcher $a_{j}$ and, after having possibly performed several actions, it turns out that these actions are useless, then $a_{i}$ sends a message (sorry, $i$ ) back to searcher $a_{j}$.

The whiteboard of every node contains a local stack, and two vectors direction[] and cleared_port[]. The protocol insures that, after the node has been visited by a searcher, direction $[0]$ indicates the port number to take for reaching the homebase, and, for $i>0$, direction $[i]$ is the port number of the edge that searcher $a_{i}$ has used to leave the current node the last time it was at this node. At node $v$, for any $1 \leq p \leq \operatorname{deg}(v)$, cleared_port $[p]=1$ if and only if the edge corresponding to the port number $p$ is clear, otherwise cleared_port $[p]=0$.

When a searcher at a node $v$ decides to perform any action, it saves a trace of this action in the local stack of the node. A trace is a triple ( $X, a, x$ ) where $X$ is a symbol, $a$ is a searcher's ID, and $x$ is either a port number, or a searcher's ID, depending on symbol $X$. More precisely:

- (CC, i,p) means that $p$ is the only contaminated (C) port, and searcher $a_{i}$ decided to clear (C) the edge that corresponds to $p$;
- ( $C J, i, p$ ) means that some searcher joined (J) $a_{i}$ at this node, and $a_{i}$ decided to clear (C) the edge that corresponds to $p$;
- ( $J J, i, j$ ) means that searcher $a_{i}$ decided to join (J) searcher $a_{j}$;
- $(R T, i, j)$ means that searcher $a_{i}$ received (R) a message from searcher $a_{j}$;
- $(S T, i, j)$ means that searcher $a_{i}$ decided to send (S) a message to searcher $a_{j}$;
- ( $A C, i, p$ means that searcher $a_{i}$ arrived (A) at $v$ by port $p$ after clearing (C) the corresponding edge;
- $(A H, i, p)$ means that searcher $a_{i}$ arrived (A) at $v$ by port $p$ in order to join another (H) searcher.

```
Program of searcher \(i\) at node \(v\).
begin
/* Searcher \(i\) receives a message */
Case:
    message \(=\) start
        decide();
    message \(=(\) move,\(j)\)
        push \((R T, i, j)\);
        decide();
    message \(=(\) help,\(j)\)
        push( \(R T, i, j)\);
        \(p \leftarrow\) smallest contaminated port;
        clear_edge(CJ, i, p)
    message \(=(\) sorry,\(j)\)
        back();
```

```
/* Searcher i arrives at node v by port p*/
```

/* Searcher i arrives at node v by port p*/
Case:
Case:
state = CLEAR
state = CLEAR
if no other searcher is at v then
if no other searcher is at v then
erase whiteboard;
erase whiteboard;
direction[0] \leftarrowp;
direction[0] \leftarrowp;
cleared_port [p]}\leftarrow1\mathrm{ ;
cleared_port [p]}\leftarrow1\mathrm{ ;
push(AC,i,p);
push(AC,i,p);
if i\not=1 then
if i\not=1 then
push(ST,i,1);
push(ST,i,1);
send message (move,i) to 1;
send message (move,i) to 1;
else decide();
else decide();
state = (HELP, j)
state = (HELP, j)
push(AH,i,p);
push(AH,i,p);
join(j);
join(j);
state = BACKTRACK
state = BACKTRACK
back();
back();
end

```
end
```

Figure 2: Skeleton of Protocol dist_search

### 4.2 The algorithm dist_search

The protocol dist_search organizes the movements of the searchers, and the messages exchanged between them, in a specific order. Based on a lexicographic order of the searchers' actions, dist_search orders them to always execute the smallest action that can be performed. As for Algorithm $\mathcal{A}$, the principle of dist_search is to try every possible monotone connected search strategy using $k$ searchers, until either the whole graph is clear, or no searcher can move without implying recontamination. In the latter case, the searcher that made the last move backtracks, and dist_search tries the next action according to the lexicographic order on the actions.

The termination of dist_search is insured as follows. The graph is cleared at time $t$ if and only if all searchers are occupying clear nodes at this time, i.e., nodes whose all incident edges are clear. This configuration is identified by the searchers because searcher $a_{1}$ tries to help all the other searchers, from $a_{2}$ to $a_{k}$, but none of them need help. Conversely, the searchers identify that $k$ searchers are not sufficient to clear the graph when they are all occupying the homebase, and try to pop the local stack that is empty. In this case, $a_{1}$ calls for a new searcher, and the $k+1$ searchers are ready to try again capturing the intruder from the homebase.

A skeleton of the protocol dist_search is given in Figures 2-4. More precisely, Figure 2 describe the global behavior of a searchers, using subroutines described in Figures 3-4. A searcher reacts to either the reception of a message (cf. left part of Figure 2), or to its arrival at a node (cf. right part of Figure 2). The message type start is uniquely for the purpose of the initialization: initially, searcher $a_{1}$ receives a message start (and hence calls procedure decide()).

We now examine the overall description od dist_search as it appears on Figure 2.
If searcher $a_{i}$ receives a message (move, $j$ ), then, by definition of such a message, it simply means that it is the turn of $a_{i}$ to proceed. Therefore, $a_{i}$ writes on the whiteboard of the node where it is currently standing that it received a message from searcher $a_{j}$ giving it turn to proceed. For this purpose, $a_{i}$ pushes ( $R T, i, j$ ) in the local stack. The nature of the next actions of $a_{i}$ depends on the result of procedure decide(). Before describing this latter procedure, let us
list all other cases depending on the message received by $a_{i}$. If $a_{i}$ receives a message (help, $j$ ) then it means that $a_{j}$ has just arrived at the same node as $a_{i}$ to help him. Thus, $a_{i}$ pushes $(R T, i, j)$ in the local stack, and clears the edge with the smallest port number $p$ among all contaminated edges incident to the node where $a_{i}$ is standing. This action is performed by calling procedure clear_edge $(C J, i, p)$. Finally, if $a_{i}$ receives a message (sorry, $j$ ), then it means that $a_{i}$ had sent a message (move, $i$ ) or a message (help, $i$ ) to $a_{j}$ but $a_{j}$ could not do anything, or all actions $a_{j}$ attempted lead to backtracking. Therefore, $a_{i}$ calls procedure back() to figure out which searcher it can help next.

The action of searcher $a_{i}$ arriving at some node $v$ by port $p$ depends on its local state. In state (HELP, $j$ ), $a_{i}$ aims at joining $a_{j}$ to help him clearing the network. Hence $a_{i}$ pushes $(A H, i, p)$ in the local stack to indicate that it arrived here by port $p$ in order to join another searcher, and then calls procedure join() to figure out what to do next in order to join $a_{j}$. In state BACKTRACK, $a_{i}$ simply calls procedure back() to carry on its backtracking. The case where $a_{i}$ arrives at a node $v$ in state CLEAR is more evolved. If there is no other searcher at $v$ then $a_{i}$ erases the whiteboard since it was accessible to the intruder, and thus its content is meaningless (when a searchers erases a whiteboard, it resets all local variables to 0 , and the local stack to $\emptyset$ ). Then $a_{i}$ sets direction $[0]$ to $p$ to indicate that it arrived here via port $p$, and sets cleared_port $[p]$ to 1 to indicate that the edge of port $p$ is clear. It then pushes $(A C, i, p)$ in the local stack at $v$ to indicate that indeed $a_{i}$ arrived at $v$ by port $p$ after clearing the corresponding edge. At this point, the behavior of $a_{i}$ depends on whether $i=1$ or not. While $a_{1}$ simply calls decide() to figure out what to do next, $a_{i}$, for $i>1$, proposes to $a_{1}$ to proceed next. For this purpose, $a_{i}$ sends a message (move, $i$ ) to $a_{1}$. Of course, to keep trace of this action, $a_{i}$ pushes $(S T, i, 1)$ in the local stack.

Remark. Before entering into the details of the procedures mentioned above, note that the actions are ordered. For instance, if several incident edges can be cleared then the cleared one is with the smallest port number. Similarly, after clearing an edge, $a_{i}$ proposes to the smallest searcher $a_{1}$ to proceed next. As we will see in the details of the procedures decide() and back(), protocol dist_search always tries to perform the smallest action. This is in particular the role of procedure next_searcher $(i)$ described on the right hand side of Figure 3.

We now describe the procedures of Figures 3-4. Unless specified otherwise, searcher $a_{i}$ is the one executing the procedure.

## Procedure next_searcher()

Procedure next_searcher () aims at determining which searcher $a_{j}$ proceeds next. In the case where $a_{i}$ is the searcher with smallest index occupying the node, $j=i+1$. Otherwise, i.e., $a_{i}$ is not the searcher with smallest index occupying the node, $j$ is the smallest index $>i$ such that $a_{j}$ is not occupying the same node as $a_{i}$. Once $j$ is found, $a_{i}$ offers to $a_{j}$ to proceed next, by sending it a message (move, $i$ ). As always, a trace of this action is kept at the current node by pushing $(S T, i, j)$ in the local stack. If there is no $a_{j}$ with $j>i$ not occupying the same node as $a_{i}$, then $a_{i}$ calls $\operatorname{back}()$ for the purpose of backtracking.

The procedures clear_edge() and move() described in the left side of Figure 3 execute clearing an edge, and traversing an edge, respectively. (Of course, clearing an edge requires traversing it).

## Procedure clear_edge():

The searcher $a_{i}$ executing Procedure clear_edge $(X, i, p)$ first pushes the trace $(X, i, p)$ on the local stack, sets cleared_port $[p]$ to 1 for specifying that the edge of port $p$ is clear, resets its

```
clear_edge(action X , ID i, port p )
\(/^{*} X \in\{C C ; C J\}^{*} /\)
begin
    push \((X, i, p)\);
    cleared_port \([p] \leftarrow 1\);
    state \(\leftarrow\) CLEAR;
    move \((p, i)\);
end
move(port_number p, ID i)
begin
    direction \([i] \leftarrow p\);
    leave the current vertex by port number \(p\);
next_searcher (searcher_ID i)
begin
    \(j \leftarrow i+1 ;\)
    if \(i\) is not the smallest searcher at node \(v\) then
        while ( \(j\) is at node \(v\) ) and ( \(j \leq k\) ) do
            \(j \leftarrow j+1 ;\)
    if \(j \leq k\) then
        push \((S T, i, j)\);
        send (move, \(i\) ) to \(j\);
end
```

end

Figure 3: Procedures clear_edge, next_searcher and move.
local state to CLEAR, and finally leaves the node through port $p$ to clear the corresponding edge.

## Procedure move():

The searcher $a_{i}$ executing Procedure move $(p, i)$ simply leaves the current node via port $p$. But before doing so, it sets direction $[i]=p$ to specify that, in order to reach $a_{i}$ from that node, one should take port $p$.

We now describe procedures decide(), back(), and join() detailed in Figure 4.

## Procedure decide()

Procedure decide() is called at a node when the concerned searcher $a_{i}$ aims at deciding what search action it has to perform. Let $v$ be the node where searcher $a_{i}$ applies decide().

If node $v$ is clear, or at least another searcher $a_{\ell}, \ell<i$, stands at $v$, then $a_{i}$ is not required to guard node $v$. Thus $a_{i}$ tries to help another searcher. According to the order mentioned above, $a_{i}$ tries to help the searcher with the smallest ID. Hence, $a_{i}$ applies join(2) if $i=1$, and join(1) otherwise. (The internal boolean variable terminated of $a_{1}$ is set to true if $i=1$; recall that this variable is used to insure termination of Protocol dist_search).

If there is a single contaminated edge incident to $v$, then searcher $a_{i}$ clears it by applying procedure clear_edge.

Otherwise (i.e., $a_{i}$ is the searcher with smallest ID currently standing at vertex $v$, and $v$ has more than one incident contaminated edge), $a_{i}$ cannot move since the protocol insures that it is the searcher with smallest ID at a node that preserves it from recontamination. Therefore, if $i=k$ (i.e., all searchers have tried to progress, but none of them can) then searcher $a_{i}$ applies $\operatorname{back}()$ in order to backtrack. On the other hand, if $i<k$ then $a_{i}$ applies next_searcher $(i)$ to let another searcher the chance to progress.

## Procedure back()

Procedure back () is called for the purpose of backtracking, yielding recontamination in some cases. Let $v$ be a vertex where searcher $a_{i}$ applies Procedure back(). Searcher $a_{i}$ first updates its state to BACKTRACK, and pops the top of the local stack, stored in the local variable msg . The

```
back()
begin
    state \(\leftarrow\) BACKTRACK;
    \(m s g \leftarrow \operatorname{pop}() ;\)
    case:
        \(m s g=(R T, i, j)\)
            send (sorry, \(i\) ) to \(j\);
        \(m s g=(J J, i, j)\)
            if \((i=k\) and \(j=i-1)\) then \(\operatorname{back}()\);
            else if \((i \neq k\) and \(j=k)\) then
                next_searcher(i);
                    else
                    if \(j+1 \neq i\) then \(\ell \leftarrow j+1\);
                    else \(\ell \leftarrow j+2\);
                    push(JJ,i, \(\ell)\);
                    join( \(\ell\) );
        \(m s g=(C C, i, p)\)
            cleared_port \([p] \leftarrow 0\);
            if \(i=k\) then back()
            else next_searcher \((i)\);
        \(m s g=(C J, i, p)\)
            cleared_port \([p] \leftarrow 0\);
            if \(\exists q\) the smallest contaminated port
            with \(q>p\) then clean_edge \((C J, i, q)\);
            else
                    \(m s g 2 \leftarrow \operatorname{pop}() ;\)
                    if \(m s g 2=(A H, i, p)\) then \(\operatorname{move}(p, i)\);
                    else \(m s g 2=(R T, i, j)\) then
                            send (sorry, i) to \(j\);
        \(m s g=(A C, i, p)\)
            cleared_port \([p] \leftarrow 0\);
            move \((p, i)\);
        \(m s g=(A H, i, p)\)
            move \((p, i)\);
        \(m s g=(S T, i, j)\)
            back();
        \(m s g=\emptyset\)
            \(k \leftarrow k+1 ;\)
            initialisation( \(k\) );
end
```

```
decide()
```

decide()
begin
begin
if node v is clear
if node v is clear
or there is another searcher \ell<i at v then
or there is another searcher \ell<i at v then
if }i=1\mathrm{ then
if }i=1\mathrm{ then
j\leftarrow2;
j\leftarrow2;
terminated }\leftarrow\mathrm{ true;
terminated }\leftarrow\mathrm{ true;
else j}\leftarrow1
else j}\leftarrow1
push(JJ,i,j);
push(JJ,i,j);
join(j);
join(j);
else if \exists unique contaminated port p then
else if \exists unique contaminated port p then
clear_edge(CC,i,p);
clear_edge(CC,i,p);
else if i\not=k then
else if i\not=k then
next_searcher(i);
next_searcher(i);
else back();
else back();
end
end
join(j)
join(j)
begin
begin
state \leftarrow ( (HELP, }j)\mathrm{ ;
state \leftarrow ( (HELP, }j)\mathrm{ ;
if j is present at v}\mathrm{ then
if j is present at v}\mathrm{ then
if v}\mathrm{ is clean then
if v}\mathrm{ is clean then
if i=1 and terminated and j=k
if i=1 and terminated and j=k
then "The graph is clear";
then "The graph is clear";
else back();
else back();
else
else
if i=1 then terminated }\leftarrow\mathrm{ false;
if i=1 then terminated }\leftarrow\mathrm{ false;
Let q}\mathrm{ be the smallest contaminated port;
Let q}\mathrm{ be the smallest contaminated port;
if j<i then
if j<i then
push(ST,i,j);
push(ST,i,j);
send (help,i) to }j
send (help,i) to }j
else
else
clear_edge(CJ,i,q);
clear_edge(CJ,i,q);
else
else
p\leftarrowdirection[j];
p\leftarrowdirection[j];
if }p=0\mathrm{ do }p\leftarrow\mathrm{ direction[0];
if }p=0\mathrm{ do }p\leftarrow\mathrm{ direction[0];
move(p,i);
move(p,i);
end

```
end
```

Figure 4: Procedures back, decide, and join, executed by searcher $a_{i}$.
behaviour of $a_{i}$ then depends on $m s g$, and leads to eight cases. These eight cases correspond to the as many different types of traces let at the top of the stack.

- Case $m s g=(R T, i, j)$ : it means that searcher $a_{j}$ had sent a message to $a_{i}$ to let him a chance to progress. Since $a_{i}$ applies back(), it means that $a_{i}$ actually cannot do anything now (note however that $a_{i}$ might have done something before, and later backtracked). Thus, $a_{i}$ sends message (sorry, $i$ ) to $a_{j}$ in order to decline, and to let $a_{j}$ the possibility to do something else.
- Case $m s g=(J J, i, j):$ it means that, at some previous step of the strategy, searcher $a_{i}$,
standing at vertex $v$, had decided to help searcher $a_{j}$. Since $a_{i}$ applies back (), it means that its attempt to help searcher $a_{j}$ did not succeded. Several situations must then be considered:
If there is another searcher that $a_{i}$ has not tried to help yet (i.e., $j<k$ and $i \neq k$, or, $i=k$ and $j<k-1$ ), then $a_{i}$ tries to help among those the searcher that has smallest ID (denoted by $a_{\ell}$ ), by applying $j \operatorname{join}(\ell)$.
Otherwise, if $i=k$ (i.e., all searchers have tried to progress, but none of them could) then searcher $a_{i}$ applies back () again in order to backtrack again. But if $i<k$ then searcher $a_{i}$ applies next_searcher $(i)$ to let another searcher the chance to progress.
- Case $m s g=(C C, i, p)$ : it means that $a_{i}$ is the searcher with smallest ID at vertex $v$, and $v$ has a single incident contaminated edge, with port $p$. Since $a_{i}$ applies back(), it means that $a_{i}$ just backtracked from clearing this edge, letting it be recontaminated. Hence, $a_{i}$ cannot do anything else. Thus, either $i=k$ (i.e., all searchers have tried to progress, but none of them could) and then searcher $a_{i}$ applies back() again in order to backtrack again, or, $i<k$ and then searcher $a_{i}$ applies next_searcher $(i)$ to let another searcher the chance to progress.
- Case $m s g=(C J, i, p)$ : it means that $a_{i}$ just backtracked from clearing the edge corresponding to port number $p$, letting it be recontaminated. Moreover, this clearing involved another searcher $a_{j}$ (with $j>i$ ). Two cases are then possible depending on whether searcher $a_{i}$ had come at $v$ to help searcher $a_{j}$ or the other way around. The former case is called Case 1, and the latter Case 2.
If there is an edge that $a_{i}$ has not try to clear yet (i.e., a contaminated edge with port number $q>p$ ), then searcher $a_{i}$ applies Procedure clear_edge $(C J, i, q)$ to clear this edge ( $C J$ indicates that such a move is possible because of the presence of another searcher at $v)$.
Otherwise, $p$ is the largest port number associated to a contaminated edge. Therefore, in Case 1, searcher $a_{i}$ had tried to help $a_{j}$ (resp., in Case 2, $a_{j}$ had tried to help $a_{i}$ ) without success. In both cases, $a_{i}$ has to backtrack again, and thus, it pops the top of the local stack in a local variable called $m s g 2$. If $m s g 2=(A H, i, q)$, then we are in Case 1 , and thus searcher $a_{i}$ goes back through the edge from which it had come (i.e., the edge with port number $q$ ). If $m s g 2 \neq(A H, i, q)$, then the only possible case is $m s g 2=(R T, i, j)$, which corresponds to Case 2 . That is, searcher $a_{j}$ had come at $v$ to help searcher $a_{i}$, and, since $i<j$, searcher $a_{j}$ had sent the message (help,j) to $a_{i}$ (cf. Procedure join()). In this latter case, searcher $a_{i}$ informs searcher $a_{j}$ that its help has been unsuccessfull, by sending message (sorry, $i$ ) to $a_{j}$.
- Case $m s g=(A C, i, p)$ : it means that searcher $a_{i}$ had come to this vertex by the edge with port number $p$, after clearing this edge. Since searcher $a_{i}$ is applying $\operatorname{back}(), a_{i}$ backtracks, i.e., goes back through the same edge letting this edge be recontaminated.
- Case $m s g=(A H, i, p)$ : it means that searcher $a_{i}$ had come to this vertex by the edge with port number $p$, in order to help a searcher (i.e., this edge was already clear). Since searcher $a_{i}$ is applying $\operatorname{back}(), a_{i}$ backtracks its move by going back through the same edge it came from.
- Case $m s g=(S T, i, j)$ : it means that searcher $a_{i}$ had send a message to searcher $a_{j}$, and that $a_{j}$ just had sent to $a_{i}$ the message (sorry, $j$ ), meaning that $a_{j}$ could not do anything more. Thus, $a_{i}$ applies back() in order to backtrack again.
- Case $m s g=\emptyset:$ it means that all actions that searchers might have done before have been backtracked. Note that only searcher $a_{1}$ can be in such a situation. Since it is in state backtrack, it means that all strategies using $k$ searchers have been tried without success. Thus, the protocol carries on with one more searcher.


## Procedure join()

Let $v$ be a vertex where searcher $a_{i}$ applies join $(j)$. Applying this procedure means that searcher $a_{i}$ has decided to help searcher $a_{j}$. First, $a_{i}$ updates its state to (HELP, $j$ ).

If $a_{j}$ is standing at $v$ then the behaviour of searcher $a_{i}$ depends on whether $v$ is clear or not. If $v$ is clear, $i=1$, terminated is true, and $j=k$, then searcher $a_{1}$ has tried to help all the searchers but none of them need its help. Thus, the whole graph is clear. Else, but still under the assumption that $v$ is clear, searcher $a_{i}$ backtracks its attempt of helping $a_{j}$ by applying $\operatorname{back}()$, since $a_{j}$ does not need any help. The last subcase is when $a_{j}$ is standing at a node $v$ that is not clear. In this case, the searcher of smallest ID between $a_{i}$ and $a_{j}$ has to clear the contaminated edge with smallest port number (say $q$ ) incident to $v$. If $i<j$, then searcher $a_{i}$ applies clear_edge $(C J, i, q)$ to clear the edge ( $C J$ meaning that this cleaning can be done thanks to the presence of another searcher). If $i>j$, then searcher $a_{i}$ sends (help,i) to $a_{j}$, in order to let searcher $a_{j}$ know that it can clear some edge thanks to the presence of $a_{i}$.

If $a_{j}$ is not standing at $v$, then $a_{i}$ tries to join searcher $a_{j}$ by following it (if $a_{j}$ has already visited node $v$ ), or by returning to the homebase. Precisely, $a_{i}$ uses indications on whiteboards. Recall that if $a_{j}$ was at a node, the whiteboard contains in direction $[j]$ the port number through which $a_{j}$ left that node. Agent $a_{i}$ returns to the homebase using direction[0] until it passes through a node where direction $[j]$ is set, in which case $a_{i}$ starts following this direction to eventually find $a_{j}$.

## 5 Proof of Correctness of dist_search

At any step of dist_search, there is only one operation performed, on only one of the stacks distributed over all nodes of the network. Indeed, only the searcher who has just received a message can perform an action, and, in particular, modify a stack. Thus we can define a centralized virtual stack, $Q_{\text {virtual }}$, where we push or pop all the moves performed by the searchers, at the same time they are pushed or popped in and out of the distributed stacks.

Precisely, a move is a pair $\left(a_{i} \rightarrow a_{j}, p\right)$, to be interpreted as follows.

- If $i \neq j$, then $\left(a_{i} \rightarrow a_{j}, p\right)$ means that $a_{i}$ leaves its current node by port $p$ with the objective of joining $a_{j}$;
- The move $\left(a_{i} \rightarrow a_{i}, p\right)$ means that $a_{i}$ leaves its current node by port $p$, clearing the corresponding edge.

An extended move corresponds to a sequence of moves. From the interpretation above, the extended move ( $a_{i}, a_{i}, p$ ) is equivalent to the move ( $a_{i} \rightarrow a_{i}, p$ ), and if $i \neq j$ then the extended move ( $a_{i}, a_{j}, p$ ) is equivalent to the sequence of moves

$$
\left(a_{i} \rightarrow a_{j}, p_{1}\right),\left(a_{i} \rightarrow a_{j}, p_{2}\right), \ldots,\left(a_{i} \rightarrow a_{j}, p_{\ell}\right),\left(\min \left\{a_{i}, a_{j}\right\} \rightarrow \min \left\{a_{i}, a_{j}\right\}, p\right)
$$

where $p_{1}, \ldots, p_{\ell}$ is a sequence of port numbers corresponding to a path (in the cleared part of the graph) between the node occupied by $a_{i}$ and the node occupied by $a_{j}$ when the extended move ( $a_{i}, a_{j}, p$ ) is considered.
$Q_{\text {virtual }}$ is updated in the following way. At every execution of the Procedure move(), we push or pop a move in $Q_{v i r t u a l}$ as follows. If $a_{i}$ applies move $(p, i)$ during the execution of Procedure clear_edge $(X, i, p)$, then the move $\left(a_{i} \rightarrow a_{i}, p\right)$ is pushed in $Q_{v i r t u a l}$. If $a_{i}$ applies move $(p, i)$ during the execution of Procedure $\operatorname{join}(j)$, then the move $\left(a_{i} \rightarrow a_{j}, p\right)$ is pushed in $Q_{\text {virtual }}$, where $p$ is the port number set during the execution of $\operatorname{join}()$, before the call of procedure move(). Finally, if a searcher applies $\operatorname{move}(p, i)$ during the execution of Procedure back () , then $Q_{\text {virtual }}$ is popped.

With this definition of $Q_{\text {virtual }}$, we show that the stack $Q$ of the centralized algorithm $\mathcal{A}$, and the virtual stack $Q_{v i r t u a l}$ are equivalent in the following way. Let $Q=M_{1}|\cdots| M_{r}$ be a valid sequence of extended moves (possibly empty). We define the following notions:

- $Q_{\text {virtual }}$ is strongly equivalent to $Q$ if, for any $1 \leq j \leq r$, there exists a sequence $S_{j}$ of moves that is equivalent to $M_{j}$, such that $Q_{v i r t u a l}=S_{1}|\cdots| S_{r}$.
- $Q_{\text {virtual }}$ is weakly equivalent to $Q$ if, for any $1 \leq j \leq r$, there exists a sequence $S_{j}$ of moves that is equivalent to $M_{j}$, such that $Q_{\text {virtual }}=S_{1}|\cdots| S_{r} \mid S_{r+1}$ where $S_{r+1}=\left(a_{i} \rightarrow\right.$ $\left.a_{i^{\prime}}, p_{1}\right),\left(a_{i} \rightarrow a_{i^{\prime}}, p_{2}\right), \ldots,\left(a_{i} \rightarrow a_{i^{\prime}}, p_{\ell}\right)$ where $p_{1}, \cdots, p_{\ell}$ is a sequence of port numbers corresponding to a path between a searcher $a_{i}$ and a searcher $a_{i^{\prime}}$, in the cleared part of the graph (in the configuration associated to $Q$ in state $M_{1}|\cdots| M_{r}$ ).

It is easy to check that two strongly equivalent stacks correspond to exactly the same strategy (i.e., at the end of both strategies, the set of cleared edges, and the positions of the searchers are the same). If $Q$ and $Q_{v i r t u a l}$ are only weakly equivalent, then the strategy associated to $Q_{\text {virtual }}$ consists in performing the strategy associated to $Q$ and then to move some searcher to the node occupied by some other searcher (in the cleared part of the graph, and without recontamination). We will see later why this weaker version of equivalence is important in our proof.

The two stacks $Q_{\text {virtual }}$ and $Q$ are said equivalent if they are either strongly equivalent or weakly equivalent.

The proof of dist_search proceeds by considering the algorithm step by step, where a step is a stage of the execution where an edge is either cleared or recontaminated. That is, a step of dist_search denotes a step of its execution when a move of type $\left(a_{i} \rightarrow a_{i}, p\right)$ is pushed in or popped out $Q_{\text {virtual }}$.

Formally, we prove that, for any $t \geq 0$, the virtual stack $Q_{v i r t u a l}$ after step $t$ of dist_search is equivalent to the stack $Q$ constructed by $\mathcal{A}$. In other words, we prove that, at any step $t \geq 0$, both algorithms construct the same partial strategy. That is, at any step, the cleared subgraph and the positions of the searchers that guard the border of this cleared subgraph are the same for both strategies. Simultaneously, we prove that for any step, when an extended move is popped out in $\mathcal{A}$, all the traces of the equivalent sequence of moves in dist_search are removed from the distributed whiteboards

Our proof is by induction on the number $t$ of steps. We assume that the centralized stack $Q$ and the virtual stack $Q_{\text {virtual }}$ are equivalent up to step $t$, and we consider the next step for proving that they are again equivalent. The difficulty of the proof is due to the number of different cases to consider. There are actually exactly fourteen cases to consider, grouped in two groups:

- Group A: $Q$ and $Q_{\text {virtual }}$ just cleared an edge $e$. One case corresponds to the graph being entirely clear. Otherwise there are three cases: (1) a searcher can clear a new edge edge, or (2) a searcher can join another searcher and one of them can clear a new edge, or (3) no other edge can be cleared and the clearing of $e$ has to be canceled. These cases have
to be combined with three other cases depending on the way $e$ has been cleared. Thus Group A yields seven cases in total.
- Group B: $Q$ and $Q_{\text {virtual }}$ just cancelled the clearing of an edge. Then, either another edge $e$ can be cleared, or no other edge can be cleared (and the last cleared edge, say $e^{\prime}$, has to be canceled). In the former case, there are three subcases depending on the type of move that has been popped out the stack (canceling corresponding to popping out the stack). In the latter case, there are four subcases depending on the way $e^{\prime}$ had been cleared. Thus Group B yields seven other cases.

The proof consists in a careful analysis of each of these fourteen cases. Before analysing these cases, we first prove that the stacks computed at the first step of both algorithms are equivalent. Initially, both $Q$ and $Q_{\text {virtual }}$ are empty. In dist_search, $a_{1}$ executes the decide function.

- If $\operatorname{deg}\left(u_{0}\right)=1$, then Algorithm dist_search pushes $(C C, 1,1)$ and $(A C, 1, p)$ on the distributed whiteboards, while $\left(a_{1} \rightarrow a_{1}, 1\right)$ is pushed in $Q_{v i r t u a l}$. During the first execution of the while loop in Algorithm $\mathcal{A}$, since $\operatorname{deg}\left(u_{0}\right)=1$, we get $Q=\left(\left(a_{1}, a_{1}, 1\right)\right)$. Moreover, in both cases, the cleared subgraph is one edge $\left(u_{0}, w\right)$ incident to $u_{0}$ with $a_{1}$ at node $w$, and all the others at node $u_{0}$.
- If $\operatorname{deg}\left(u_{0}\right)>1$ and $k=1$, then both algorithms state that another searcher is needed. The two stacks remain empty and only $u_{0}$ is clear.
- If $\operatorname{deg}\left(u_{0}\right)>1$ and $k>1$, then Algorithm dist_search pushes $(S T, 1,2),(R T, 2,1)$, $(J J, 2,1),(S T, 2,1),(R T, 1,2),(C C, 1,1)$ and $(A C, 1, p)$ on the distributed whiteboards, while $\left(a_{1} \rightarrow a_{1}, 1\right)$ is pushed in $Q_{\text {virtual }}$. Algorithm $\mathcal{A}$ pushes $\left(a_{2}, a_{1}, 1\right)$ in $Q$. Thus, both stacks are strongly equivalent. Indeed, $a_{2}$ and $a_{1}$ were already at the same node (the homebase) and thus there is no move associated to the fact that $a_{2}$ joins $a_{1}$. Then, in both stacks, $a_{1}$ clears the edge with port number 1 at $u_{0}$.

Let us assume that after step $t$ of both algorithms, the two stacks $Q$ and $Q_{\text {virtual }}$ are equivalent. We prove, for the fourteen cases previously enumerated, that after the next step $t+1$ of both algorithms, the search strategy will remain the same for both algorithms, i.e., both stacks remain equivalent, and the same configurations are achieved by both algorithms. The next two subsections consider separately the cases in group A and in group B.

### 5.1 Group A

Group A assumes that $Q$ and $Q_{\text {virtual }}$ have been reached by clearing an edge. Let $S$ and $S_{\text {virtual }}$ be the states of $Q$ and $Q_{v i r t u a l}$ at this step of both algorithms. Since, $Q$ and $Q_{v i r t u a l}$ has been reached by clearing an edge, they are strongly equivalent. Thus, there exist a sequence $S^{\prime}$ of valid extended moves, and a sequence $S_{\text {virtual }}^{\prime}$ of moves, with $S^{\prime}$ and $S_{\text {virtual }}^{\prime}$ strongly equivalent, and there exist an extended move $M$, and a sequence $M^{\prime}$ of moves, with $M^{\prime}$ equivalent to $M$, such that $S=S^{\prime} \mid M$ and $S_{\text {virtual }}=S_{\text {virtual }}^{\prime} \mid M^{\prime}$.

We first prove that the next step of the execution of Algorithm dist_search starts with $a_{1}$ applying Procedure decide(). Let $1 \leq j \leq k$ be the ID of the searcher that has just cleared the last contaminated edge. Searcher $a_{j}$ arrived at a node in state clear. Either $j=1$, and $a_{j}$ applied Procedure decide(), or $a_{j}$ sent (move, $j$ ) to $a_{1}$, who received (move, $j$ ) from $j$. In both cases, $a_{1}$ applies Procedure decide().

Now, we consider the subcases of Group A.

### 5.1.1 Case A. 1

In Case A.1, the whole graph is assumed to be cleared. In this case, by Lemma 5, Algorithm $\mathcal{A}$ terminates. Let us prove it is also the case for Algorithm dist_search. Searcher $a_{1}$ applies Procedure decide(). Since the graph is clear, the vertex $v_{1}$ where $a_{1}$ stands, is clear. Thus, $a_{1}$ pushes ( $J J, 1,2$ ) and applies Procedure join(2) after having set terminated to true. Applying Procedure $\operatorname{join}(), a_{1}$ computes a port number $p_{1}$ that is either direction[2] if $a_{2}$ has already been at vertex $v_{1}$, and direction $[0]$ otherwise (recall that direction $[0]$ is the direction of the homebase). The former case is identified by the fact that direction $[2] \neq 0$. We push $\left(1 \rightarrow 2, p_{1}\right)$ in $Q_{\text {virtual }}$. Then, $a_{1}$ takes the edge corresponding to port $p_{1}$ at $v_{1}$, and arrives at a new node $v_{2}$ by port $q_{1}$, in state (HELP, 2). At $v_{2}$, searcher $a_{1}$ writes $\left(A H, 1, q_{1}\right)$ on the whiteboard, and applies again the $j \operatorname{join}()$ procedure. This is repeated until $a_{1}$ eventually joins $a_{2}$, at a node $v_{t}$. Let $P=v_{1}, v_{2}, \ldots, v_{t}$ be the path followed by $a_{1}$ from $v_{1}$ until it reaches $a_{2}$ at $v_{t}$. Let $p_{i}$ (resp., $q_{i}$ ) be the port number of the edge $\left\{v_{i}, v_{i+1}\right\}$ at $v_{i}$ (resp., $v_{i+1}$ ). At every node $v_{i}, i \geq 2$, searcher $a_{1}$ writes $\left(A H, 1, q_{i-1}\right)$ during the execution of join(). In $Q_{v i r t u a l}$, we push $\left(a_{1} \rightarrow a_{2}, p_{i}\right)$ for $i=1, \ldots, t-1$. Since $v_{t}$ is clear, searcher $a_{2}$ does not need help, and thus $a_{1}$ applies Procedure $\operatorname{back}()$. Therefore, it pops $\left(A H, 1, q_{t-1}\right)$ from the whiteboard of $v_{t}$, and returns to $v_{t-1}$. At every node $v_{i}$, for $i=t-1, \ldots, 2$, searcher $a_{1}$ arrives in state backtrack, and thus pops the local stack, that contains $\left(A H, 1, q_{i-1}\right)$. As a result, it goes to $v_{i-1}$ using port $q_{i-1}$. Simultaneously, we pop ( $a_{1} \rightarrow a_{2}, p_{i}$ ) that we had previously pushed in $Q_{\text {virtual }}$. Eventually, $a_{1}$ is back at $v_{1}$ is state backtrack. At $v_{1}$, searcher $a_{1}$ applies Procedure back (), and thus pops ( $J J, 1,2$ ) from the local stack. This procedure asks $a_{1}$ to try helping every possible searcher $a_{i}$, for $3 \leq i \leq k$. For this purpose, $a_{1}$ successively applies Procedure $\operatorname{join}(i)$ for $i=3, \ldots, k$. Since the whole graph is clear, no searcher needs help, and therefore the same situation as for $a_{2}$ occurs for $i=3, \ldots, k-1$, i.e., $a_{1}$ joins $a_{i}$, and goes back to $v_{1}$ since $a_{i}$ does not need help. The sequence of pushes and pops is the same for $a_{i}$ as for $a_{2}$. When $a_{1}$ eventually reachs $a_{k}$, the state variable terminated of $a_{1}$ is still equal to true, and thus Algorithm dist_search terminates. The virtual stack satisfies $Q_{\text {virtual }}=S_{\text {virtual }}\left|\left(a_{1} \rightarrow a_{k}, r_{1}\right)\right| \cdots \mid\left(a_{1} \rightarrow a_{k}, r_{\ell}\right)$ where $r_{1}, \ldots, r_{\ell}$ is the sequence of port numbers from $v_{1}$ to the node where $a_{1}$ meets $a_{k}$. The stack $Q$ is again in state $S$ because no extended moves have been pushed in it. Since, by the induction hypothesis, both stacks $Q$ and $Q_{\text {virtual }}$ were equivalent before these sequence of moves, the new state $S$ of stack $Q$, and the new state $S_{\text {virtual }}\left|\left(a_{1} \rightarrow a_{k}, p_{1}\right)\right| \cdots \mid\left(a_{1} \rightarrow a_{k}, p_{\ell}\right)$ of stack $Q_{v i r t u a l}$, are weakly equivalent.

### 5.1.2 Case A. 2

Case A. 2 assumes that a valid extended move can be performed in the current configuration of the search strategy. In this case, Algorithm $\mathcal{A}$ pushes in $Q$ the smallest valid extended move $M$ (thus, the state of $Q$ becomes $S \mid M$ ). Let us prove it is also the case for Algorithm dist_search, independentely from the type of $M$. We prove that there exists a sequence $M^{\prime}$ of moves that is equivalent to $M$, and such that, after the next step, the state of $Q_{v i r t u a l}$ becomes $S_{\text {virtual }} \mid M^{\prime}$.

- Case A.2.1: $M$ is of type $\left(a_{i}, a_{i}, p\right)$.

We consider only the case $i>1$, as the case $i=1$ easily follows. For any $1 \leq j<i, a_{j}$ is guarding some node $v_{j}$ that has more than one incident contaminated edge, and $a_{j}$ is the searcher with the smallest ID at $v_{j}$. Moreover, the node $v_{i}$ where searcher $a_{i}$ stands has only one incident contaminated edge. Let $p$ be the port number corresponding to this unique contaminated edge. In Algorithm dist_search, $a_{1}$ applies decide(). Applying this procedure, searcher $a_{1}$ writes $(S T, 1,2)$ and sends (move, 1) to $a_{2}$. For any $2 \leq$ $j \leq i$, searcher $a_{j}$ receives (move, $j-1$ ) from $a_{j-1}$ and writes ( $R T, j, j-1$ ). Applying Procedure decide(), searcher $a_{j}$ writes $(S T, j, j+1)$ and sends (move, $j$ ) to $a_{j+1}$. When
$a_{i}$ receives the message (move, $i-1$ ) from $a_{i-1}$, it applies decide () that calls Procedure clear_edge ( $C C, i, p$ ). Thus, $a_{i}$ writes ( $C C, i, p$ ) on the whiteboard of $v_{i}$ and takes the edge corresponding to port $p$ of $v_{i}$, clearing this edge. Then, searcher $a_{i}$ arrives at a new node $v$. Finally, $a_{i}$ writes $(A C, i, q)$ and $(S T, i, 1)$ on the whiteboard of $v$. We push the move $\left(a_{i} \rightarrow a_{i}, p\right)$ in $Q_{\text {virtual }}$. Thus, the state of $Q_{\text {virtual }}$ becomes $S_{\text {virtual }} \mid\left(a_{i} \rightarrow a_{i}, p\right)$ which is strongly equivalent to the state $S \mid\left(a_{i}, a_{i}, p\right)$ of $Q$.

- Case A.2.2: $M$ is of type $\left(a_{1}, a_{j}, p\right)$ with $j>1$.

In this case, searcher $a_{j}$ is the searcher with the smallest ID, that stands at a contaminated vertex, say $v_{j}$. In particular, searcher $a_{1}$ stands at a clear vertex, say $v_{1}$, and is aiming at helping searcher $a_{j} . a_{1}$ applies Procedure decide(). Since the vertex $v_{1}$ is clear, $a_{1}$ pushes $(J J, 1,2)$ and applies Procedure join(2) after having set terminate to true. Similarily to the Case A.1, this procedure asks $a_{1}$ to try helping every possible searcher $a_{i}$, for $3 \leq i \leq j$. For any $i=3, \ldots, j-1$, since searcher $a_{i}$ does not need help, searcher $a_{1}$ applies Procedure $\operatorname{back}()$ after having reached $a_{i}$. Then $a_{1}$ goes back to $v_{1}$ and applies join $(i+1)$ (cf., Case A.1). When $a_{1}$ eventually reachs $a_{j}$ at $v_{j}$, the state variable terminated of $a_{1}$ is set to false. The virtual stack satisfies $Q_{\text {virtual }}=S_{\text {virtual }}\left|\left(a_{1} \rightarrow a_{j}, r_{1}\right)\right| \cdots \mid\left(a_{1} \rightarrow a_{j}, r_{\ell}\right)$ where $r_{1}, \ldots, r_{\ell}$ is the sequence of port numbers from $v_{1}$ to the node where $a_{1}$ meets $a_{j}$. Let $p$ be the smallest port number of a contaminated edge incident to $v_{j}$. Then, searcher $a_{1}$ applies Procedure clear_edge $(C J, 1, p)$, that is, it writes $(C J, 1, p)$ and clears the corresponding edge. The move ( $a_{1} \rightarrow a_{1}, p$ ) is pushed in $Q_{\text {virtual }}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}\left|\left(a_{1} \rightarrow a_{j}, r_{1}\right)\right| \cdots\left|\left(a_{1} \rightarrow a_{j}, r_{\ell}\right)\right|\left(a_{1} \rightarrow a_{1}, p\right)$, which is strongly equivalent to the state $S \mid\left(a_{1}, a_{j}, p\right)$ of $Q$.

- Case A.2.3: $M$ is of type $\left(a_{i}, a_{1}, p\right)$ with $i>1$.

In this case, for any $\ell<i$, searcher $a_{\ell}$ is alone at a vertex $v_{\ell}$ with more than one contaminated incident edge. Moreover, searcher $a_{i}$ stands at $v_{i}$, a clear vertex or a vertex occupied by a searcher $a_{\ell}$, with $\ell<i$. In the distributed algorithm dist_search, $a_{1}$ applies Procedure decide () . Applying this procedure, searcher $a_{1}$ writes $(S T, 1,2)$ and sends (move, 1) to $a_{2}$. For any $2 \leq j \leq i$, searcher $a_{j}$ receives (move, $j-1$ ) from $a_{j-1}$ and writes ( $R T, j, j-1$ ). Applying Procedure decide (), searcher $a_{j}$ writes ( $S T, j, j+1$ ) and sends (move, $j$ ) to $a_{j+1}$. When $a_{i}$ receives the message (move, $i-1$ ) from $a_{i-1}$, it applies decide () that calls Procedure join(1). This procedure is called until $a_{i}$ eventually joins $a_{1}$. Let $P=w_{1}, w_{2}, \ldots, w_{r}$ be the path followed by $a_{i}$ from $w_{1}=v_{i}$ until $w_{r}=v_{1}$. Let $p_{j}$ (resp., $q_{j}$ ) be the port number of the edge $\left\{w_{j}, w_{j+1}\right\}$ at $w_{j}$ (resp., $w_{j+1}$ ). At every node $w_{j}, j \geq 2$, searcher $a_{i}$ writes ( $A H, i, q_{j-1}$ ) during the execution of join(). In $Q_{v i r t u a l}$, we push $\left(a_{i} \rightarrow a_{1}, p_{j}\right)$ for $j=1, \ldots, r-1$. At $w_{r}$, searcher $a_{i}$ writes $(S T, i, 1)$ and sends the message (help, i) to searcher $a_{1}$. Then, $a_{1}$ writes ( $R T, 1, i$ ) and ( $C J, 1, p$ ), and clears the corresponding edge. The move ( $a_{1} \rightarrow a_{1}, p$ ) is pushed in $Q_{\text {virtual }}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}\left|\left(a_{i} \rightarrow a_{1}, p_{1}\right)\right| \cdots\left|\left(a_{i} \rightarrow a_{1}, p_{r}\right)\right|\left(a_{1} \rightarrow a_{1}, p\right)$, which is strongly equivalent to the state $S \mid\left(a_{i}, a_{1}, p\right)$ of $Q$.

### 5.1.3 Case A. 3

Case A. 3 assumes that there does not exist any valid extended move according to the current state of the stack $Q$. Therefore, Algorithm $\mathcal{A}$ pops the last executed extended move $M$ from $S=$ $S^{\prime} \mid M$. Let us prove that Algorithm dist_search does the same. Let us assume that $a_{i}, i \geq 1$,
has cleared the last edge $e=(v, w)$ by taking the port $p$ of $v$. Recall that $S_{v i r t u a l}=S_{v i r t u a l}^{\prime} \mid M^{\prime}$ with $M^{\prime}$ a sequence of moves equivalent to $M$. There are three cases to be considered:

- Case A.3.1: $M=\left(a_{i}, a_{i}, p\right)$. In this case, $M^{\prime}$ is the 1-element sequence ( $a_{i} \rightarrow a_{i}, p$ ).
- Case A.3.2: There exists $s, i<s \leq k$, such that $M=\left(a_{i}, a_{s}, p\right)$. In this case, searcher $a_{i}$ leaves a node, say $v_{i}$ to join searcher $a_{s}$ at node $v$. Then searcher $a_{i}$ clears the edge corresponding to the port $p$ of $v$. Let $P=w_{1}, w_{2}, \ldots, w_{r}$ be the path followed by $a_{i}$ from $w_{1}=v_{i}$ until $w_{r}=v$. Let $p_{j}$ (resp., $q_{j}$ ) be the port number of the edge $\left\{w_{j}, w_{j+1}\right\}$ at $w_{j}$ (resp., $w_{j+1}$ ). By the induction hypothesis, $M^{\prime}=\left(a_{i} \rightarrow a_{s}, p_{1}\right)|\cdots|\left(a_{i} \rightarrow a_{s}, p_{r}\right) \mid\left(a_{i} \rightarrow\right.$ $\left.a_{i}, p\right)$
- Case A.3.3: There exists $s, i<s \leq k$, such that $M=\left(a_{s}, a_{i}, p\right)$. In this case, searcher $a_{s}$ leaves a node, say $v_{s}$ to join searcher $a_{i}$ at node $v$. Then searcher $a_{i}$ clears the edge corresponding to the port $p$ of $v$. Let $P=w_{1}, w_{2}, \ldots, w_{r}$ be the path followed by $a_{s}$ from $w_{1}=v_{s}$ until $w_{r}=v$. Let $p_{j}$ (resp., $q_{j}$ ) be the port number of the edge $\left\{w_{j}, w_{j+1}\right\}$ at $w_{j}$ (resp., $w_{j+1}$ ). By the induction hypothesis, $M^{\prime}=\left(a_{s} \rightarrow a_{i}, p_{1}\right)|\cdots|\left(a_{s} \rightarrow a_{i}, p_{r}\right) \mid\left(a_{i} \rightarrow\right.$ $a_{i}, p$ )
Searcher $a_{i}$ has arrived at the node $w$ by port number, say $q$, and $a_{i}$ has pushed ( $\left.A C, i, q\right)$. If $i>1, a_{i}$ has also pushed $(S T, i, 1)$ and sent (move, $i$ ) to $a_{1}$, who has pushed ( $R T, 1, i$ ) at its current vertex. Then, searcher $a_{1}$ applied Procedure decide(). Since there does not exist any valid extended move, it means that, for any $1 \leq j \leq k$, searcher $a_{j}$ is at a vertex $v_{j}$ which has more than one incident contaminated edge, and for any $1 \leq j<\ell \leq k, v_{j} \neq v_{\ell}$. For any $j<k$, $a_{j}$ writes $(S T, j, j+1)$ and sends (move, $j$ ) to $a_{j+1}$ by applying Procedure next_searcher $(j)$ in Procedure decide(). Then searcher $a_{j+1}$ pushes ( $R T, j+1, j$ ) at its current vertex before applying Procedure decide () too. When $a_{k}$ receives the message (move, $k-1$ ) from $a_{k-1}$, it applies Procedure decide() that calls Procedure back(). Then, $a_{k}$ pops ( $R T, k, k-1$ ) and sends (sorry, $k$ ) to searcher $a_{k-1}$. For any $j>1, a_{j}$ receives (sorry, $j+1$ ) from $a_{j+1}$. Then searcher $a_{j}$ applies Procedure $b a c k()$ that pops $(S T, j, j+1)$, then pops ( $R T, j, j-1$ ), and sends (sorry, $j$ ) to $a_{j-1}$. When $a_{1}$ receives (sorry, 2), $a_{1}$ applies Procedure back() that pops (ST, 1, 2), then pops ( $R T, 1, i$ ), and sends (sorry, 1) to $a_{i}$. By applying Procedure back (), $a_{i}$ pops ( $S T, i, 1$ ), then $(A C, i, q)$. Finally, $a_{i}$ puts cleared_port $[q]$ to false and goes back to $v$ (letting the edge $e$ be recontaminated). Searcher $a_{i}$ arrives in state backtrack by port number $p$. THus, the move $\left(a_{i} \rightarrow a_{i}, p\right)$ is popped from $Q_{v i r t u a l}$. Then $a_{i}$ puts cleared_port $[p]$ to false. Thus, the edge $e$ is known to have been recontaminated and $a_{i}$ has returned to his previous position. Thus, both algorithms have backtracked the clearing of the last cleared edge. Note that in the three subcases, we only popped the move $\left(a_{i} \rightarrow a_{i}, p\right)$. Thus, the new state of $Q_{v i r t u a l}$ depends on the case:
- Case A.3.1: $S_{\text {virtual }}^{\prime}$
- Case A.3.2: $S_{\text {virtual }}^{\prime}\left|\left(a_{i} \rightarrow a_{s}, p_{1}\right)\right| \cdots \mid\left(a_{i} \rightarrow a_{s}, p_{\ell}\right)$
- Case A.3.3: $S_{\text {virtual }}^{\prime}\left|\left(a_{s} \rightarrow a_{i}, p_{1}\right)\right| \cdots \mid\left(a_{s} \rightarrow a_{i}, p_{\ell}\right)$

Therefore, $S_{v i r t u a l}^{\prime}$ is equivalent to the state $S^{\prime}$ of $Q$ (strongly equivalent in case A.3.1, and weakly equivalent in the two other cases).

### 5.2 Group B

Cases in Group B assumes that $Q$ and $Q_{\text {virtual }}$ have been achieved by backtracking the clearing of an edge. Let $M$ be the extended move popped by Algorithm $\mathcal{A}$ during the previous step. Let $S$
and $S_{\text {virtual }}$ be the states of $Q$ and $Q_{v i r t u a l}$ at this step in the two algorithms respectively. Thus, there exist $i \geq 1$, a vertex $v$, a port $p$ of $v$ corresponding to an edge $e$, such that searcher $a_{i}$ has just arrived in state BACKTRACK, at vertex $v$, by port $p$, letting the edge $e$ be recontaminated. Thus, in these cases, the next step of the execution of Algorithm dist_search starts with $a_{i}$ applying Procedure $\operatorname{back}()$.

### 5.2.1 Case B. 1

Case B. 1 assumes that there exists a valid extended move larger than $M$. In this case, Algorithm $\mathcal{A}$ pushes the smallest valid extended move $M^{\prime} \succ M$ in $Q$. In the following, $M^{\prime}$ can be of three different types defined bellow. Let us prove that Algorithm dist_search executes a sequence of moves equivalent to $M^{\prime}$. There are three cases depending on the type of the extended move $M$.

- Case B.1.1: $M=\left(a_{i}, a_{i}, p\right)$. This case occurs after the removal operation as in case A.3.1. Thus $S$ and $S_{\text {virtual }}$ are actually strongly equivalent. In this case, there exist $i<j \leq \ell \leq k$ and $0 \leq q \leq n$ such that the extended move $M^{\prime}=\left(a_{j}, a_{\ell}, q\right)$ is larger than $M$. That is, $j$ is the smallest ID larger than $i$ such that $a_{j}$ can perform a valid extended move. By applying Procedure back (), $a_{i}$ pops (CC,i,p) at $v$. Thus, $a_{i}$ calls procedure next_searcher $(i)$, then pushes $(S T, i, i+1)$ at $v$, and sends (move, $i$ ) to $a_{i+1}$. In the same way as for Case A.2, the message (move, $j-1$ ) is received by $a_{j}$ which can perform a valid extended move. As in Case A. 2 again, searcher $a_{j}$ performs this move and we push in $Q_{v i r t u a l}$ a sequence of moves equivalent to $M^{\prime}$. Thus, both stacks remains strongly equivalent.
- Case B.1.2: $M=\left(a_{i}, a_{j}, p\right)$ with $i<j$.

This case occurs after the removal operation as in case $A .3 .2$. Thus $S$ and $S_{\text {virtual }}$ are weakly equivalent. More precisely, there exit a state $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$, and a sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers, such that $S_{\text {virtual }}=S_{\text {virtual }}^{\prime} \mid\left(a_{i} \rightarrow\right.$ $\left.a_{j}, p_{1}\right)|\cdots|\left(a_{i} \rightarrow a_{j}, p_{t-1}\right)$. Let $v_{i}$ (resp., $\left.v_{j}\right)$ be the vertex where searcher $a_{i}$ (resp., $a_{j}$ ) stands in the configuration associated to $S_{v i r t u a l}^{\prime}$. Note that $v_{j}=v$. The sequence ( $p_{1}, \ldots, p_{t-1}$ ) is exactly the sequence of port numbers that searcher $a_{i}$ has followed along a path from $v_{i}$ to $v_{j}$. Let $P=w_{1}, \cdots, w_{t}$ be this path, with $w_{1}=v_{i}$ and $w_{t}=v_{j}$. More precisely, the configuration associated to $S_{\text {virtual }}$ is got from the configuration associated to $S_{\text {virtual }}^{\prime}$ (which is also the configuration associated to $S$ ) by moving searcher $a_{i}$ along the path from $v_{i}$ to $v_{j}$ by following the sequence ( $p_{1}, \ldots, p_{t-1}$ ) of port numbers. Let $q$ be the port number of $v_{j}$ corresponding to the edge $\left\{w_{t-1}, v_{j}\right\}$. Recall that, when it had joined $a_{j}$ at $v_{j}$, searcher $a_{i}$ had written $(A H, i, q)$. Then, since $i<j, a_{i}$ had written ( $C J, i, p$ ) and had cleared the edge.

To prove that both stacks remain equivalent, we consider the type of the extended move $M^{\prime}$. There are three cases:

- Case B.1.2.a: there is a port number $r$ of $v_{j}$, larger than $p$ such that the corresponding edge is contaminated. In this case, $M^{\prime}=\left(a_{i}, a_{j}, r\right)$.
- Case B.1.2.b: there is a searcher with ID $\ell \leq k$, larger than $j$, at vertex $v_{\ell}$, and a port number $r$ of $v_{\ell}$ such that the corresponding edge is contaminated. In this case, $M^{\prime}=\left(a_{i}, a_{\ell}, r\right)$.
- Case B.1.2.c: there is a searcher with ID $\ell \leq k$, larger than $i$, at vertex $v_{\ell}$, that can preform a valid extended move. That is, there exist $\ell<u \leq k$ and $r$ such that $M^{\prime}=\left(a_{\ell}, a_{u}, r\right)$.

Note that the extended move in Case B.1.2.a is smaller than the extended move in Case B.1.2.b that is smaller than the extended move in Case B.1.2.c.

Now, let us consider what is the execution of Protocol dist_search after having backtracked the clearing of $e$. By applying Procedure $\operatorname{back}(), a_{i}$ pops ( $\left.C J, i, p\right)$. Then, Algorithm dist_search first checks whether there exists a port number $r>p$ of a contaminated edge incident to $v_{j}$. Let us assume that such a port number exists. This corresponds to the Case B.1.2.a:

- Algorithm $\mathcal{A}$ pushes $M^{\prime}=\left(a_{i}, a_{j}, r\right)$ in $Q$. Searcher $a_{i}$ pushes $(C J, i, r)$ at $v_{j}$ and clear the corresponding edge, arriving at a new node by port, say $o$. Searcher $a_{i}$ pushes ( $A C, i, o$ ) and $(S T, i, 1)$ at the new node, and then send message (move, $i$ ) to $a_{1}$. We push ( $\left.a_{i} \rightarrow a_{i}, r\right)$ in $Q_{v i r t u a l}$. Thus, the state of $Q$ is $S \mid M_{1}^{\prime}$ and the state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}^{\prime}\left|\left(a_{i} \rightarrow a_{j}, p_{1}\right)\right| \cdots\left|\left(a_{i} \rightarrow a_{j}, p_{t-1}\right)\right|\left(a_{i} \rightarrow a_{i}, r\right)$. Therefore, both stacks are strongly equivalent.

Now, let us assume that there does not exist a port number of $v_{j}$, larger than $p$, corresponding to a contaminated edge. In this case, $a_{i}$ applies Procedure back(). Therefore, it pops $(A H, i, q)$ from the whiteboard of $w_{t}=v_{j}$, and returns to $w_{t-1}$. At every node $w_{f}$, for $f=t-1, \ldots, 2$, searcher $a_{i}$ arrives in state BACKтRACK, and thus pops the local stack, that contains $\left(A H, i, q_{f}\right)$ where $q_{f}$ is the port number leading to $w_{f-1}$. As a result, it goes to $w_{f-1}$ using port $q_{f-1}$. Simultaneously, we pop $\left(a_{i} \rightarrow a_{j}, p_{f}\right)$ that we had previously pushed in $Q_{\text {virtual }}$. Eventually, $a_{i}$ is back at $v_{i}$ in state backtrack. At this stage of the execution of dist_search, the current state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$. Then, by applying Procedure back(), $a_{i}$ pops ( $J J, i, j$ ). Then, Algorithm dist_search checks whether searcher $a_{i}$ can help a searcher with ID larger than $j$. By applying Procedure $\operatorname{back}(), a_{i}$ pushes $(J J, i, j+1)$ and applies Procedure $j o i n(j+1)$. Similarily to the Case A.1, this procedure asks $a_{i}$ to try helping every possible searcher $a_{t}$, for $j+1 \leq t \leq k$. Let us assume that there is a searcher with ID $\ell \leq k$, larger than $j$, at vertex $v_{\ell}$, and a port number $r$ of $v_{\ell}$ such that the corresponding edge is contaminated. This corresponds to the Case B.1.2.b:

- Algorithm $\mathcal{A}$ pushes $M^{\prime}=\left(a_{i}, a_{\ell}, r\right)$ in $Q$. For any $f=j+1, \ldots, \ell-1$, since searcher $a_{f}$ does not need help, searcher $a_{i}$ applies Procedure $\operatorname{back}()$ after having reached $a_{f}$. Then $a_{i}$ goes back to $v_{i}$ and applies $j o i n(f+1)$ (cf., Case A.1). When $a_{i}$ eventually reachs $a_{\ell}$ at $v_{\ell}$, the virtual stack satisfies $Q_{\text {virtual }}=S_{\text {virtual }}\left|\left(a_{i} \rightarrow a_{\ell}, p_{1}\right)\right| \cdots \mid\left(a_{i} \rightarrow\right.$ $a_{\ell}, p_{t}$ ) where $p_{1}, \ldots, p_{t}$ is the sequence of port numbers from $v_{i}$ to $v_{\ell}$. Then, searcher $a_{i}$ applies Procedure clear_edge $(C J, i, r)$, that is, it writes $(C J, i, r)$ and clears the corresponding edge. The move $\left(a_{i} \rightarrow a_{i}, r\right)$ is pushed in $Q_{v i r t u a l}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}\left(a_{i} \rightarrow a_{\ell}, p_{1}\right)|\cdots|\left(a_{i} \rightarrow a_{\ell}, p_{t}\right) \mid\left(a_{i} \rightarrow a_{i}, r\right)$, which is strongly equivalent to the state $S \mid\left(a_{i}, a_{\ell}, r\right)$ of $Q$.

Now, we consider the case where there is no $\ell>j$ such that searcher $a_{\ell}$ stands at a vertex $v_{\ell}$ incident to a contaminated edge. Thus, $a_{i}$ reachs back $v_{i}$ after having tried to help all searchers $a_{\ell}$, for $j<\ell \leq k$ (by iteratively applying Procedure join() as in the previous case). At this stage of the execution dist_search, the current state of $Q_{v i r t u a l}$ is $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$ (the current state of $Q$ ). When $a_{i}$ reachs back $v_{i}$, it pops $(J J, i, k)$. Thus, Procedure back() calls Procedure next_searcher $(i)$. Therefore, $a_{i}$ pushes $(S T, i, i+1)$ and sends (move, $i$ ) to searcher $a_{i+1}$. Let us assume that there is a searcher
with ID $\ell \leq k$, larger than $i$, at vertex $v_{\ell}$, that can preform a valid extended move. This corresponds to the Case B.1.2.c:

- In this case, there exist $\ell \leq u \leq k$ and $r \leq n$ such that Algorithm $\mathcal{A}$ pushes $M^{\prime}=\left(a_{\ell}, a_{u}, r\right)$ in $Q$. As for the case $A .2 .3$, for any $i \leq f \leq \ell$, searcher $a_{f}$ receives (move, $f-1$ ) from $a_{f-1}$ and writes ( $R T, f, f-1$ ). Applying Procedure decide(), searcher $a_{f}$ writes ( $S T, f, f+1$ ) and sends (move, f) to $a_{f+1}$. When $a_{\ell}$ receives the message (move, $\ell-1$ ) from $a_{\ell-1}$, it applies the decide() procedure. Then the move $m^{\prime}$ is performed as for the case A.2. Thus, both stacks become strongly equivalent.
- Case B.1.3: $M=\left(a_{j}, a_{i}, p\right)$ with $i<j$.

This case occurs after the removal operation as in case $A .3 .3$. Thus $S$ and $S_{\text {virtual }}$ are weakly equivalent. More precisely, there exit a state $S_{v i r t u a l}^{\prime}$ that is strongly equivalent to $S$, and a sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers, such that $S_{\text {virtual }}=S_{\text {virtual }}^{\prime} \mid\left(a_{j} \rightarrow\right.$ $\left.a_{i}, p_{1}\right)|\cdots|\left(a_{j} \rightarrow a_{i}, p_{t-1}\right)$. Let $v_{i}$ (resp., $\left.v_{j}\right)$ be the vertex where searcher $a_{i}$ (resp., $a_{j}$ ) stands in the configuration associated to $S_{v i r t u a l}^{\prime}$. Note that $v_{i}=v$. The sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ is exactly the sequence of port numbers that searcher $a_{j}$ has followed along a path from $v_{j}$ to $v_{i}$. Let $w_{1}, \cdots, w_{t}$ be this path, with $w_{1}=v_{j}$ and $w_{t}=v_{i}$. More precisely, the configuration associated to $S_{\text {virtual }}$ is got from the configuration associated to $S_{\text {virtual }}^{\prime}$ (which is also the configuration associated to $S$ ) by moving searcher $a_{j}$ along the path from $v_{j}$ to $v_{i}$ by following the sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers. Let $q$ be the port number of $v_{i}$ corresponding to the edge $\left\{w_{t-1}, v_{i}\right\}$. Recall that, when it had joined $a_{i}$ at $v_{i}$, searcher $a_{j}$ had written $(A H, j, q)$. Then, since $i<j, a_{j}$ had pushed $(S T, j, i)$ at $v_{i}$ and sent (help,$j$ ) to searcher $a_{i}$. Then, searcher $a_{i}$ has pushed $(R T, i, j)$ and ( $\left.C J, i, p\right)$ at $v_{i}$, and had cleared the edge.
To prove that both stacks remain equivalent, we consider the type of the extended move $M^{\prime}$. There are three cases:

- Case B.1.3.a: there is a port number $r$ of $v_{i}$, larger than $p$ such that the corresponding edge is contaminated. In this case, $M^{\prime}=\left(a_{j}, a_{i}, r\right)$.
- Case B.1.3.b: there is a searcher with ID $\ell \leq k$, larger than $i$, at vertex $v_{\ell}$, and a port number $r$ of $v_{\ell}$ such that the corresponding edge is contaminated. In this case, $M^{\prime}=\left(a_{j}, a_{\ell}, r\right)$.
- Case B.1.3.c: there is a searcher with ID $\ell \leq k$, larger than $j$, at vertex $v_{\ell}$, that can preform a valid extended move. That is, there exist $\ell<u \leq k$ and $r$ such that $M^{\prime}=\left(a_{\ell}, a_{u}, r\right)$.

Note that the extended move in Case B.1.3.a is smaller than the extended move in Case B.1.3.b that is smaller than the extended move in Case B.1.3.c.

Now, let us consider what is the execution of Protocol dist_search after having backtracked the clearing of $e$. By applying Procedure $\operatorname{back}(), a_{i}$ pops $(C J, i, p)$. Then, Algorithm dist_search first checks whether there exists a port number $r>p$ of a contaminated edge incident to $v_{j}$. Let us assume that such a port number exists. This corresponds to the Case B.1.3.a.

- As for the case B.1.2..a, searcher $a_{i}$ clears the edge corresponding to the port number $r$ and both stacks remain strongly equivalent.

If there does not exist a port number of $v_{j}$, larger than $p$, corresponding to a contaminated edge, $a_{i}$ applies Procedure back (). Searcher $a_{i}$ pops ( $C J, i, p$ ), then ( $R T, i, j$ ), and sends
(sorry, $i$ ) to searcher $a_{j}$. Then, searcher $a_{j}$ applies Procedure back (). Therefore, it pops $(S T, j, i)$ and $(A H, i, q)$ from the whiteboard of $w_{t}=v_{i}$, and returns to $w_{t-1}$. At every node $w_{f}$, for $f=t-1, \ldots, 2$, searcher $a_{j}$ arrives in state BACKTRACK, and thus pops the local stack, that contains $\left(A H, j, q_{f}\right)$ where $q_{f}$ is the port number of $w_{f}$ leading to $w_{f-1}$. As a result, it goes to $w_{f-1}$ using port $q_{f-1}$. Simultaneously, we pop ( $a_{j} \rightarrow a_{i}, p_{f}$ ) that we had previously pushed in $Q_{\text {virtual }}$. Eventually, $a_{j}$ is back at $v_{j}$ in state backtrack. At this stage of the execution of dist_search, the current state of $Q_{v i r t u a l}$ is $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$. Then, by applying Procedure back ()$, a_{j}$ pops ( $J J, j, i$ ). Then, Algorithm dist_search checks whether searcher $a_{j}$ can help a searcher with ID larger than $i$. Then, Case B.1.3.b is similar to Case B.1.2.b, and Case B.1.3.c is similar to Case B.1.2.c. Thus, both stacks become strongly equivalent.

### 5.2.2 Case B. 2

Case B. 2 assumes that there does not exist a valid extended move greater than $M$. In this case, either $S=\emptyset$ or there is a valid move $M^{\prime}$ and a sequence of valid extended moves $S^{\prime}$ such that $S=S^{\prime} \mid M^{\prime}$. In the former case, Algorithm $\mathcal{A}$ claims that another searcher is required. In the latter case, Algorithm $\mathcal{A}$ pops $M^{\prime}$ from $Q$. Let us prove it is also the case for Algorithm dist_search. There are four cases according to whether $S=\emptyset$ or not, and depending on the type of $M^{\prime}$.

- Case B.2.1 If $S=\emptyset$, there are two cases. Either $k=1$ and $u_{0}$ has more than one incident edge, or $k>1$. In the former case, searcher $a_{1}$ applies Porcedure decide(), then Procedure back () that asks for a second searcher. In the latter case, $M$ must be the extended move $\left(a_{k}, a_{k-1}, p\right)$ where $p$ is the greatest port number of $u_{0}$. Indeed, if $M$ is not this extended move, then an extended move greater than $M$ would be valid. In this case, searcher $a_{k-1}$ has just arrived in state BACKTRACK, at vertex $u_{0}$ by port $p$. Moreover, all searchers are standing at $u_{0}$. Beside, the whiteboard of $u_{0}$ contains exactly the sequence $((S T, 1,2),(R T, 2,1), \cdots,(S T, i, i+1),(R T, i+1, i), \cdots,(S T, k-1, k),(R T, k, k-$ $1),(J J, k, k-1),(S T, k, k-1),(R T, k-1, k),(C J, k-1, p))$. Thus, $S_{\text {virtual }}=\emptyset$. Thus, $Q$ and $Q_{\text {virtual }}$ are strongly equivalent. Moreover, it is easy to check that Procedure back() asks for a $(k+1)^{t h}$ searcher.

Let us assume that $S \neq \emptyset$. Recall that, in Case B., a searcher $a_{i}$ has just arrived back in state backtrack, at the vertex $v$, by port $p$, letting the edge $e$ be recontaminated. Thus, in these cases, the next step of the execution of Algorithm dist_search starts with the $a_{i}$ applying Procedure back().

Let $f=\left(v^{\prime}, w^{\prime}\right)$ be the edge cleared by the move $M^{\prime}$. Let $s \leq k$ be the ID of the searcher that has cleared $f$, arriving by port $r^{\prime}$ of $w^{\prime}$. Let $r$ be the port number of $v^{\prime}$ corresponding to $f$. Let us consider the three possible types for the move $M^{\prime}$ :

- Case B.2.2 $M^{\prime}=\left(a_{s}, a_{s}, r\right)$. In this case, there is a sequence of valid moves $S_{v i r t u a l}^{\prime}$ strongly equivalent to $S^{\prime}$, and a sequence of valid moves $M_{v i r t u a l}$ equivalent to $M$ such that $S_{\text {virtual }}=S_{\text {virtual }}^{\prime}\left|\left(a_{s} \rightarrow a_{s}, r\right)\right| M_{\text {virtual }}$.
- Case B.2.3 There exists $s^{\prime}<s$ such that $M^{\prime}=\left(a_{s}, a_{s^{\prime}}, r\right)$. In this case, there is a sequence of valid moves $S_{v i r t u a l}^{\prime}$ strongly equivalent to $S^{\prime}$, a sequence of valid moves $M_{\text {virtual }}$ equivalent to $M$, and a sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers, such that $S_{\text {virtual }}=$ $S_{\text {virtual }}^{\prime}\left|\left(a_{s} \rightarrow a_{s^{\prime}}, p_{1}\right)\right| \cdots\left|\left(a_{s} \rightarrow a_{s^{\prime}}, p_{t-1}\right)\right|\left(a_{s} \rightarrow a_{s}, r\right) \mid M_{\text {virtual }}$.
- Case B.2.4 There exists $s^{\prime}<s$ such that $M^{\prime}=\left(a_{s^{\prime}}, a_{s}, r\right)$. In this case, there is a sequence of valid moves $S_{\text {virtual }}^{\prime}$ strongly equivalent to $S^{\prime}$, a sequence of valid moves $M_{\text {virtual }}$
equivalent to $M$, and a sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers, such that $S_{\text {virtual }}=$ $S_{\text {virtual }}^{\prime}\left|\left(a_{s^{\prime}} \rightarrow a_{s}, p_{1}\right)\right| \cdots\left|\left(a_{s^{\prime}} \rightarrow a_{s}, p_{t-1}\right)\right|\left(a_{s} \rightarrow a_{s}, r\right) \mid M_{\text {virtual }}$.

After having cleared $f$, searcher $a_{s}$ has pushed ( $A H, s, r^{\prime}$ ), then ( $S T, s, 1$ ), and sent (move, $s$ ) to searcher $a_{1}$. Then $a_{1}$ applies decide (). Applying this procedure, searcher $a_{1}$ writes ( $S T, 1,2$ ) and sends (move, 1) to $a_{2}$. For all $2 \leq j \leq i-1$, searcher $a_{j}$ receives (move, $j-1$ ) from $a_{j-1}$ and writes $(R T, j, j-1)$. Applying Procedure decide(), searcher $a_{j}$ writes $(S T, j, j+1)$ and sends (move,$j$ ) to $a_{j+1}$. When $a_{i}$ receives the message (move,$i-1$ ) from $a_{i-1}$, it pushes ( $R T, i, i-1$ ) at its current vertex $v_{i}$, and applies decide(). Let $v_{i}$ be the vertex where $a_{i}$ is standing at this stage of the execution of Protocol dist_search.

Let us consider the type of the extended move $M$. Let $p$ be the port number of $v$ corresponding to $e$. Since there is no valid extended move larger than $M$, only three cases are possible:

- $M=\left(a_{i}, a_{i}, p\right)$ and for any $i<j \leq k$, searcher $a_{j}$ stands alone at a vertex, say $v_{j}$. By backtracking such a move, Protocol dist_search insures that $Q$ and $Q_{\text {virtual }}$ are strongly equivalent (cf., Case A.3.1). Thus, $v=v_{i}$. In this case, searcher $a_{i}$ arrives back at $v$ in state backtrack. Applying back ()$, a_{i}$ pops $(C C, i, p)$, pushes $(S T, i, i+1)$ at $v_{i}$, and sends (move,$i$ ) to searcher $a_{i+1}$. For any $i+1 \leq j \leq k$, searcher $a_{j}$ receives (move, $j-1$ ) from $a_{j-1}$ and writes ( $R T, j, j-1$ ). Applying decide (), searcher $a_{j}$ writes $(S T, j, j+1)$ and sends (move, $j$ ) to $a_{j+1}$. When $a_{k}$ receives the message (move, $k-1$ ) from $a_{k-1}$, searcher $a_{k}$ applies decide (), then back (). Searcher $a_{k}$ pops ( $S T, k, k-1$ ) and sends (sorry, $k$ ) to $a_{k-1}$. For any $k>j>i$, searcher $a_{j}$ receives (sorry, $j+1$ ) from $a_{j+1}$ and pops ( $S T, j, j+1$ ). Applying back (), searcher $a_{j}$ pops ( $R T, j-1, j$ ) and sends (sorry, $j$ ) to $a_{j-1}$. When searcher $a_{i}$ receives (sorry, $i+1$ ), it pops ( $S T, i, i+1$ ), and then pops ( $R T, i, i-1$ ) from the local stack of $v_{i}$.
- $i<k, M=\left(a_{i}, a_{k}, p\right)$ and for any $i<j \leq k$, searcher $a_{k}$ stands alone at a vertex, say $v_{j}$. In this case, there exists a state $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$ and a sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ of port numbers, such that $S_{\text {virtual }}=S_{\text {virtual }}^{\prime}\left|\left(a_{i} \rightarrow a_{k}, p_{1}\right)\right| \cdots \mid\left(a_{i} \rightarrow\right.$ $\left.a_{k}, p_{t-1}\right)$. Note that in the configuration associated to $S_{\text {virtual }}^{\prime}$ (resp., to $S_{\text {virtual }}$ ), searcher $a_{i}$ stands at $v_{i}$ (resp., $v$ ). Searcher $a_{k}$ stands at $v$ in both configurations. The sequence $\left(p_{1}, \ldots, p_{t-1}\right)$ is exactly the sequence of port numbers that searcher $a_{i}$ has followed along a path from $v_{i}$ to $v$. Let $P=w_{1}, \cdots, w_{t}$ be this path, with $w_{1}=v_{i}$ and $w_{t}=v$. For $2 \leq f \leq t$, let $q_{f}$ br the port number leading of $w_{f}$ corresponding to the edge $\left\{v_{f-1}, w_{f}\right\}$. Recall that, $a_{i}$ had followed the path $P$ to join $a_{k}$. Then, searcher $a_{1}$ had written $\left(A H, i, q_{t}\right)$. Then, since $i<j, a_{i}$ had written $(C J, i, p)$ and had cleared the edge.
Now, let us consider what is the execution of Protocol dist_search after having backtracked $M$. Arriving at $v$, by port $p$, in state backtrack, $a_{i}$ applies Procedure back(). Therefore, it pops $\left(A H, i, q_{t}\right)$ from the whiteboard of $v$, and returns to $w_{t-1}$. For $f=$ $t-1, \ldots, 2$, searcher $a_{i}$ arrives in state BACKTRACK at every node $w_{f}$. Then, it pops the local stack, that contains $\left(A H, i, q_{f}\right)$. As a result, it goes to $w_{f-1}$ using port $q_{f-1}$. Simultaneously, we pop ( $a_{i} \rightarrow a_{k}, p_{f}$ ) that we had previously pushed in $Q_{v i r t u a l}$. Eventually, $a_{i}$ is back at $v_{i}$ in state backtrack. At this stage of the execution of dist_search, the current state of $Q_{\text {virtual }}$ is $S_{\text {virtual }}^{\prime}$ that is strongly equivalent to $S$. Then, by applying $b a c k(), a_{i}$ pops ( $J J, i, k$ ), pushes ( $S T, i, i+1$ ) at $v_{i}$, and sends move, $i$ ) to searcher $a_{i+1}$. For any $i+1 \leq j \leq k$, searcher $a_{j}$ receives (move, $j-1$ ) from $a_{j-1}$ and writes ( $R T, j, j-1$ ). Applying decide $\left(\right.$ ), searcher $a_{j}$ writes $(S T, j, j+1)$ and sends (move,$\left.j\right)$ to $a_{j+1}$. When $a_{k}$ receives the message (move, $k-1$ ) from $a_{k-1}$, searcher $a_{k}$ applies decide(), then back () . Searcher $a_{k}$ pops ( $S T, k, k-1$ ) and sends (sorry, $k$ ) to $a_{k-1}$. For any $k>j>i$, searcher
$a_{j}$ receives (sorry, $j+1$ ) from $a_{j+1}$ and pops ( $S T, j, j+1$ ). Applying $\operatorname{back}()$, searcher $a_{j}$ pops ( $R T, j-1, j$ ) and sends (sorry, $j$ ) to $a_{j-1}$. When searcher $a_{i}$ receives (sorry, $i+1$ ), it pops $(S T, i, i+1)$, and then pops $(R T, i, i-1)$ from the local stack of $v_{i}$.
- $i=k$ and $M^{\prime}=\left(a_{i}, a_{k-1}, p\right)$. Similarily to the previous case, one can prove that there exists a round of the execution of dist_search when $a_{i}$ pops ( $R T, i, i-1$ ) from the local stack of $v_{i}$.

Thus, whatever be the type of $M$, there is a round of the execution of dist_search when $a_{i}$ pops $(R T, i, i-1)$ from the local stack of $v_{i}$. Moreover, at this round, $Q$ and $Q_{v i r t u a l}$ are strongly equivalent.

If $i=k$, searcher $a_{k}$ applies back (). Otherwise, $a_{i}$ calls next_searcher $(i)$, pushes $(S T, i, i+1)$ and sends (move, $i$ ) to $a_{i+1}$. Then, for $i<j<k, a_{j}$ pushes $(R T, j, j-1)$ and $(S T, j, j+1)$ at its current node, and sends (move, $j$ ) to $a_{j+1}$. When $a_{k}$ receives message (move, $k-1$ ), it applies $\operatorname{back}()$. Searcher $a_{k}$ sends (sorry, $k$ ) to searcher $a_{k-1}$. Then, for $k \geq j>i, a_{j} \operatorname{pops}(S T, j, j+1)$ and ( $R T, j, j-1$ ), and sends (sorry, $j$ ) to $a_{j-1}$. Finally, $a_{i}$ receives (sorry, $i+1$ ) from searcher $a_{i+1}$, and applies back (). For all $j, i \geq j>s, a_{j}$ pops $(S T, j, j+1)$ and $(R T, j, j-1)$, and sends (sorry, $j$ ) to $a_{j-1}$. Finally, $a_{s}$ receives (sorry, $s+1$ ) from searcher $a_{s+1}$ and applies back (). Then, searcher $a_{s}$ pops ( $A C, s, r^{\prime}$ ) from the local stack of $w^{\prime}$. Then, it goes back to $v^{\prime}$ in state bACKTRACK, letting recontaminated the edge $f$. We pop ( $a_{s} \rightarrow a_{s}, r$ ) from $Q_{\text {virtual }}$. Thus, in Case B.2.2 (resp., B.2.3 and B.2.4), $Q$ and $Q_{\text {virtual }}$ becomes strongly equivalent (resp., weakly equivalent).

We have proved, that in any case, both stacks remain equivalent after a step of the execution of Protocol dist_search (that is, they represent the same search strategy). Moreover, both algorithms terminate in the same state. Thus, the proof of Theorem 1 follows directly from Theorem 2.

### 5.3 Size of whiteboards

Lemma 6 Let $G$ be a connected $n$-node graph. Let $m \geq 0$ be the number of edges of $G$. During the execution of dist_search, starting from $u_{0} \in V(G)$, at most $O(m(\log \Delta+\log k))$ bits are stored in any node's whiteboard, where $\Delta$ is the maximum degree of $G$ and $k=\operatorname{mcs}\left(G, u_{0}\right)$.

Proof. First, note that during the execution of Protocol dist_search, when a move is backtracked, all its traces are erased from the whiteboards. We can only consider the traces of an extended move that has actually be performed. We study the case when a searcher $a_{i}$ has to join another searcher $a_{j}(j<i)$, and searcher $a_{j}$ clears an edge $e$. In this case, the number of traces is the largest possible. Let $f$ be the edge that has just been cleared by a searcher, say $a_{\ell}$. Let $v_{i}$ (resp., $v_{j}$ ) be the node where $a_{i}$ (resp., $a_{j}$ ) is standing after the clearing of $f$. After having cleared the edge $f, a_{\ell}$ sends the message move to $a_{1}$. Then, for any $1 \leq t \leq i$, the message move is transmitted from searcher $a_{t}$ to searcher $a_{t+1}$ until message move reachs $a_{i}$. By Procedure next_searcher (), if more than two searchers are on the same node, only the two smallest of them receive the message. It is unnecessary to send the message to the other ones. Indeed, if the two smallest cannot do anything, the others as well. Thus, between the clearing of the two edges $f$ and $e$, at most two traces of type ( $R T, \ell, s$ ) and two traces of type ( $S T, \ell, s$ ) are written on every whiteboard. When it receives move from searcher $a_{i-1}$, searcher $a_{i}$ decides to join $a_{j}$ and pushes $(J J, i, j)$ at $v_{i}$. By joining $a_{j}, a_{i}$ writes routing traces. That is, $a_{i}$ pushes one trace $(A H, i, j)$ on every whiteboard of the path between $v_{i}$ and $v_{j}$. Finally, $a_{i}$ sends message help to $a_{j}$ who clears the edge. That is, searcher $a_{i}$ pushes $(S T, i, j)$ at $v_{j}$. Then,searcher $a_{j}$ pushes $(R T, j, i)$ and $(C J, j, p)$ at $v_{j}$. Finally $a_{j}$ clears the edge $e$ and $(A C, j, q)$ and $(S T, j, 1)$
at the other end of $e$. To summarize, on every whiteboard, it has been written at most three traces of type $(R T, \ell, s)$, three of type $(S T, \ell, s)$, one for each of the types $(J J, i, j),(A H, i, j)$, $(C J, j, p)$ and $(A C, j, q)$. Thus, when an extended move is performed, at most $O(1)$ traces are written on every whiteboard. Since there are $m$ extended moves, at most $O(m)$ traces of size $(\log \Delta+\log k)$ bits are written on every whiteboard.

## 6 Conclusion

We have described a distributed search protocol that captures an intruder in any network, starting from any entry point in the network, and using a small number of searchers for this task. This result opens a wide field of investigations. For instance, is it possible to achieve the same performances as protocol dist_search using searchers modeled by finite automata? (In our case, the searchers have $O(\log k)$-bit memory when $k$ searchers are used). Also, is it possible to design a distributed protocol that performs in polynomial time, at the price of using a number of searchers bounded by a polylogarithmic function of $n$ times the search number of the network? Finally, is it possible to design a distributed protocol that computes a non-monotone connected search strategy? *

## References

[1] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a $k$-tree. SIAM J. Alg. Disc. Meth. 8(2):277-284, 1987.
[2] 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.
[3] L. Barrière, P. Fraigniaud, N. Santoro, and D. M. Thilikos. Searching is not jumping. In 29th Workshop on Graph Theoretic Concepts in Computer Science (WG), Springer-Verlag, LNCS 2880, pages 34-45, 2003.
[4] M. Bender, A. Fernandez, D. Ron, A. Sahai and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In 30th Ann. Symp. on Theory of Computing (STOC), pages 269-278, 1998.
[5] M. Bender and D. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In 35th Ann. Symp. on Foundations of Computer Science (FOCS), pages 75-85, 1994.
[6] D. Bienstock, Graph searching, path-width, tree-width and related problems (a survey), DIMACS Ser. in Discrete Mathematics and Theoretical Computer Science, 5 (1991), pp. 3349.
[7] D. Bienstock and P. Seymour. Monotonicity in graph searching. Journal of Algorithms 12:239-245, 1991.
[8] R. Breisch. An intuitive approach to speleotopology. Southwestern Cavers VI(5):72-78, 1967.

[^1][9] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg. Label-Guided Graph Exploration by a Finite Automaton. In 32nd Int. Colloquium on Automata, Languages and Programming (ICALP), 2005.
[10] S. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. SIAM J. on Computing 9(3):636-652, 1980.
[11] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree Exploration with Little Memory. In 13th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA), pages 588-597, 2002.
[12] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile Search for a Black Hole in an Anonymous Ring. In 15th Int. Symposium on Distributed Computing (DISC), 2001.
[13] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a Black Hole in Arbitrary Networks: Optimal Mobile Agent Protocols. In 21st ACM Symp. on Principles of Distributed Computing (PODC), 2002.
[14] U. Feige, M. Hajiaghayi, and J. Lee. Improved approximation algorithms for minimumweight vertex separators. In 37th ACM Symposium on Theory of Computing (STOC), 2005.
[15] P. Flocchini, F.L. Luccio, and L. Song. Decontamination of chordal rings and tori. Proc. of 8th Workshop on Advances in Parallel and Distributed Computational Models (APDCM), 2006.
[16] P. Flocchini, M. J. Huang, F.L. Luccio. Contiguous search in the hypercube for capturing an intruder. Proc. of 18th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS), 2005.
[17] F. Fomin, P. Fraigniaud and N. Nisse. Nondeterministic Graph Searching: From Pathwidth to Treewidth. In 30th International Symposium on Mathematical Foundations of Computer Science (MFCS), LNCS 3618, pages 364-375, Springer, 2005.
[18] F. V. Fomin, D. Kratsch, and I. Todinca. Exact algorithms for treewidth and minimum fill-in. In 31st Int. Colloquium on Automata, Languages and Programming (ICALP 2004), LNCS vol. 3142, Springer, pp. 568-580, 2004.
[19] P. Fraigniaud, and D. Ilcinkas. Digraphs Exploration with Little Memory. Proc. 21st Symposium on Theoretical Aspects of Computer Science (STACS), LNCS 2296, pages 246-257, 2004.
[20] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. Theoretical Computer Science 345(2-3): pages 331-344, 2005.
[21] P. Fraigniaud and N. Nisse. Connected Treewidth and Connected Graph Searching. In 7th Latin American Theoretical Informatics, LNCS 3887, pages 470-490, 2005.
[22] L. Kirousis, C. Papadimitriou. Interval graphs and searching. Discrete Math. 55, pages 181-184, 1985.
[23] L. Kirousis, C. Papadimitriou. Searching and Pebbling. Theoretical Computer Science 47, pages 205-218, 1986.
[24] A. Lapaugh. Recontamination does not help to search a graph. Journal of the ACM 40(2):224-245, 1993.
[25] F. S. Makedon and I. H. Sudborough, On minimizing width in linear layouts, Discrete Appl. Math., 23:243-265, 1989.
[26] N. Megiddo, S. Hakimi, M. Garey, D. Johnson and C. Papadimitriou. The complexity of searching a graph. Journal of the ACM 35(1):18-44, 1988.
[27] T. Parsons. Pursuit-evasion in a graph. Theory and Applications of Graphs, Lecture Notes in Mathematics, Springer-Verlag, pages 426-441, 1976.
[28] O. Reingold. Undirected ST-Connectivity in Log-Space. In 37th ACM Symp. on Theory of Computing (STOC), 2005.
[29] H.A. Rollik. Automaten in planaren Graphen. Acta Informatica 13:287-298, 1980 (also in LNCS 67, pages 266-275, 1979).
[30] K. Skodinis Computing optimal linear layout of trees in linear time. In 8th European Symp. on Algorithms (ESA), Springer, LNCS 1879, pages 403-414, 2000. (Also, to appear in SIAM Journal on Computing).
[31] B. Yang, D. Dyer, and B. Alspach. Sweeping Graphs with Large Clique Number. In 15th Annual International Symposium on Algorithms and Computation (ISAAC), pages 908-920, 2004.


[^0]:    ${ }^{\dagger}$ These authors received additional supports from the project "PairAPair" of the ACI Masses de Données, from the project "Fragile" of the ACI Sécurité Informatique, and from the project "Grand Large" of INRIA.
    *These authors received additional supports from the project "ALGOL" of the ACI Masses de Données, and from the project "ROM-EO" of the RNRT program.

[^1]:    * Although non-monotone connected search strategies may require less searchers than monotone ones [31], they are much more difficult to design, even in the centralized setting. For instance, it is even unknown whether a 1-certificate checkable in polynomial time exists for non-monotone connected graph searching.

