Distributed Chasing of Network Intruders

Lélia Blin*
IBISC
University of Evry
91000 Evry, France
lelia.blin@lami.univ-evry.fr

Pierre Fraigniaud†
LIAFA
CNRS & University of Paris Diderot
75205 Paris, France
pierre@liafa.jussieu.fr

Nicolas Nisse†
DIM
Universidad of Chile
Santiago, Chile
nisse@lri.fr

Sandrine Vial*
PRiSM
University of Versailles-Saint Quentin
78035 Versailles, France
sandrine.vial@prism.uvsq.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.

†Part of this work has been done while these authors were at LRI, University Paris Sud, France. 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 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, $\mathtt{s}(G)$, 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, pw, of the graph, and therefore the search number of an n-node graph can be approximated in polynomial time, up to multiplicative factor $O(\log n\sqrt{\log tw})$ where tw denotes the treewidth of the graph (see [14], and use the fact that $pw/tw \leq O(\log n)$).

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 $\deg(u)$ edges incident to any node u are labeled from 1 to $\deg(u)$, 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 models 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 $\mathtt{mcs}(G, u_0)$.

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 $\mathtt{mcs}(G, u_0) + 1$. Since it is known [21] that, for any graph G and for any $u_0 \in V(G)$, we have $\mathtt{mcs}(G, u_0) \leq \mathtt{s}(G)\lceil \log n \rceil$, we get that our protocol uses at most $O(\log n)$ times the optimal number of searchers. In fact, it is conjectured that $\mathtt{mcs}(G) \leq 2\mathtt{s}(G)$ for all graph G (cf. [3]). If this holds, then our protocol uses at most twice the optimal number of searchers.

Note that the searchers clear the graph in a connected way, but not necesseraly in a monotone way. Moreover, at the end of the clearing of the graph, the description of a monotone connected search strategy S is written in a distributed way on the node' whiteboards. We also prove that S uses at most $\mathtt{mcs}(G, u_0) + 1$ searchers, that is, a number of searchers very close to the optimal in a centralized context. It as been proved [16] that, in an asynchronous distributed environment, we cannot do better, even if the searchers know the topology of the network in which they are launched.

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 of the access to 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, $\mathtt{mcs}(T) \leq 2\ \mathtt{s}(T) - 2$ 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*.

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 $SL = L$ [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 leaving 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 Δ is the maximum degree of the network.

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

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 $\deg(u)$ edges incident to any node u are labeled by distinct integers from 1 to $\deg(u)$. 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 for 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 ith 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 without 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 $\mathtt{s}(G)$ 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]). Hence, in the following, all our strategies are supposed to be 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 $\texttt{mcs}(G, u_0)$.

2.2 Summary of 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 $k = \texttt{mcs}(G, u_0) + 1$ searchers if $\texttt{mcs}(G, u_0) > 1$, and $k = 1$ searcher if $\texttt{mcs}(G, u_0) = 1$;*
- *Every searcher involved in the search strategy computed by* `dist_search` *uses $O(\log k)$ bits of memory;*
- *During the execution of* `dist_search`*, at most $O(m \log n)$ bits of information are stored at every whiteboard.*

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 whiteboards of the nodes of G. S is a monotone connected search strategy for G, starting from u_0, and using at most $\texttt{mcs}(G, u_0) + 1$ 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 graph G and $u_0 \in V(G)$ such that, $k = \texttt{mcs}(G, u_0)$ and for any distributed protocol $\mathcal{P}$ for capturing a fugitive in a monotone connected way, starting from u_0, and in asynchronous anonymous settings, $\mathcal{P}$ 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 $E(G) \setminus X$. 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. $V(\mathcal{C}_k)$ is the set of all possible k-configurations. There is an arc from X to X' in $\mathcal{C}_k$ if the configuration X' 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 $\mathtt{mcs}(G, u_0)$, 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 *An* 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.

We say that a searcher is *available* whenever it does not preserve the clear part of the graph from recontamination. That is, a searcher that stands at a vertex whose all incident edges are clear is available. If a vertex of the border of the clear part (a vertex incident to a contaminated edge and incident to a clear edge) is occupied by several searchers, all of these searchers but one become available. 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 $M_1|\dots|M_r$. Pushing an extended move M in Q results in a new state, denoted by $S|M$. We say that a extended move M is *valid according to* Q if $S' = S|M$ 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 $\delta(X_S)$ 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 $M = (a_1, a_1, 1)$, or $k > 1$ and $M = (a_2, a_1, 1)$.
- If $S \neq \emptyset$, $M = (a_i, a_j, p)$ is valid according to Q if and only if either $i = j$, a_i stands at a node $u \in \delta(X_S)$, and p is the only contaminated port of node u, or $i \neq j$, a_i is available, a_j stands at a node $u \in \delta(X_S)$, 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 $M = (a_i, a_j, p)$ and $M' = (a_{i'}, a_{j'}, p')$ be two extended moves. We define $M \prec M'$ if and only if either $(i < i')$, or $(i = i'$, and $j < j')$, or $(i = i'$, $j = j'$, and $p < p')$.*

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_{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_{last}. If there is an extend move $M \succ M_{last}$ 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. There are two cases depending on whether a push or a pop is performed. These two cases do not depend on the value of *state*. 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.

— 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 ← ∅;
    state ← CLEAR;
    decided ← false;
    while not decided do
        if all searchers are available then
            decided ← true;
            possible ← true;
        else
            /* case state = CLEAR */
            if state = CLEAR then
                if there exists a valid extended move according to Q then
                    (a_i, a_j, p) ← minimum valid extended move according to Q;
                    push(a_i, a_j, p);
                else
                    if Q ≠ ∅ then
                        M_last ← pop();
                        state ← BACKTRACK;
                    else
                        decided ← true;
                        possible ← false;
                    endif
                endif
            /* case state = BACKTRACK */
            else
                Let M_last = (a_i, a_j, p);
                if there exists a valid extended move according to Q larger than (a_i, a_j, p) then
                    (a'_i, a'_j, p') ← minimum valid extended move according to Q, and larger than (a_i, a_j, p);
                    push(a'_i, a'_j, p');
                    state ← CLEAR;
                else
                    if Q ≠ ∅ then M_last ← pop();
                    else
                        decided ← true;
                        possible ← false;
                    endif
                endif
            endif
        endif
    endwhile
    return(possible, Q);
end.
```

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

Definition 4 *Given two states of the stack Q $S = M_1|\cdots|M_r$ and $S' = M'_1|\cdots|M'_{r'}$, we define $S \prec S'$ if and only if there exists $i \leq \min\{r, r'\}$ such that $M_i \prec M'_i$ and, for any $j < i$, $M_j = M'_j$.*

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 bt Algorithm $\mathcal{A}$ is 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 $S' \prec S$ appeared before, and no valid state $S'' \succ S$ 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

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' of extended moves with $S' \prec S$, or Algorithm $\mathcal{A}$ eventually computes state S of the stack.*

Lemma 4 *If $\mathtt{mcs}(G, u_0) > k$ 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, for any valid sequence $S' \preceq S$, S' 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 $\mathtt{mcs}(G, u_0) = k$. 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 $k = \mathtt{mcs}(G, u_0)$, 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 $\mathtt{mcs}(G, u_0)$ searchers. Using an additional searcher for implementing the communications between the $\mathtt{mcs}(G, u_0)$ other searchers, `dist_search` captures the intruder with $\mathtt{mcs}(G, u_0) + 1$ 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 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

- 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 it 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).

Searcher a_1 has an extra boolean variable *terminated* that is used to indicate whether the graph is clear. This occurs when searcher a_1 has tried helping all others searchers and none of them stand at a vertex incident to a contaminated edge.
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 $1 \le p \le \deg(v)$, `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. 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;
- (CJ, 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;
- (JJ, i, j) means that searcher a_i decided to join (J) searcher a_j;
- (RT, i, j) means that searcher a_i received (R) a message $(\texttt{move}, j)$ or $(\texttt{help}, j)$ from searcher a_j;
- (ST, i, j) means that searcher a_i decided to send (S) a message to searcher a_j;
- (AC, i, p) means that searcher a_i arrived (A) at v by port p after clearing (C) the corresponding edge;
- (AH, i, p) means that searcher a_i arrived (A) at v by port p in order to join another (H) searcher.

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 describe the protocol `dist_search` as it appears on Figure 2

If searcher a_i receives a message $(\texttt{move}, j)$, 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 (RT, 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 $(\texttt{help}, j)$ then it means that a_j has just arrived at the same node as a_i to help it. Thus, a_i pushes (RT, 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

Program of searcher i at node v.
 	
begin
/* Searcher i receives a message */
`Case:`
    `message` $=$ `start`
        $\mathit{decide}()$;
    `message` $= ($`move`$, j)$
        $\mathit{push}(RT, i, j)$;
        $\mathit{decide}()$;
    `message` $= ($`help`$, j)$
        $\mathit{push}(RT, i, j)$;
        $p \leftarrow$ smallest contaminated port;
        $\mathit{clear\_edge}(CJ, i, p)$
    `message` $= ($`sorry`$, j)$
        $\mathit{back}()$;
	/* Searcher i arrives at node v by port p */
	`Case:`
	    `state` $=$ CLEAR
	        if no other searcher is at v **then**
	            erase whiteboard;
	        `direction`$[0] \leftarrow p$;
	        `cleared_port`$[p] \leftarrow 1$;
	        $\mathit{push}(AC, i, p)$;
	        if $i \neq 1$ **then**
	            $\mathit{push}(ST, i, 1)$;
	            send message $($`move`$, i)$ to 1;
	        else $\mathit{decide}()$;
	    `state` $= ($HELP$, j)$
	        $\mathit{push}(AH, i, p)$;
	        $\mathit{join}(j)$;
	    `state` $=$ BACKTRACK
	        $\mathit{back}()$;
	end

Figure 2: Skeleton of Protocol `dist_search`

calling procedure $\mathit{clear\_edge}(CJ, i, p)$. 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 $\mathit{back}()$ 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 it clearing the network. Hence a_i pushes (AH, 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 $\mathit{join}()$ to figure out what to do next in order to join a_j. In state BACKTRACK, a_i simply calls procedure $\mathit{back}()$ 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 reset 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. a_i then pushes (AC, 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 $\mathit{decide}()$ 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 $(ST, i, 1)$ 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 $\mathit{decide}()$ and $\mathit{back}()$, protocol `dist_search` always tries to perform the smallest action. This is in particular the role of procedure $\mathit{next\_searcher}(i)$ described on the right hand side of Figure 3.

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 $(\texttt{move}, i)$. As always, a trace of this action is kept at the current node by pushing (ST, 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 *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).

```
clear_edge(action X, searcher_ID i, port p)
/* X ∈ {CC; CJ} */
begin
    push(X, i, p);
    cleared_port[p] ← 1;
    state ← CLEAR;
    move(p, i);
end

move(port_number p, searcher_ID i)
begin
    direction[i] ← p;
    leave the current vertex by port number p;
end
```

```
next_searcher(searcher_ID i)
begin
    j ← i + 1;
    if i is not the smallest searcher at node v then
      while (j is at node v) and (j ≤ k) do
          j ← j + 1;
    if j ≤ k then
       push(ST, i, j);
       send (move, i) to j;
    else
       back()
end
```

Figure 3: Procedures `clear_edge`, `next_searcher` and `move`.

Procedure *clear_edge*()**:**

The searcher 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 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 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_ℓ, $\ell < i$, 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

```
back()
begin
    state ← BACKTRACK;
    msg ← pop();
    case:
        msg = (RT, i, j)
            send (sorry, i) to j;
        msg = (JJ, i, j)
            if (i = k and j = i − 1) then back();
            else if (i ≠ k and j = k) then
                    next_searcher(i);
                else
                    if j + 1 ≠ i then ℓ ← j + 1;
                        else ℓ ← j + 2;
                    push(JJ, i, ℓ);
                    join(ℓ);
        msg = (CC, i, p)
            cleared_port[p] ← 0;
            if i = k then back()
            else next_searcher(i);
        msg = (CJ, i, p)
            cleared_port[p] ← 0;
            if ∃q the smallest contaminated port
            with q > p then clean_edge(CJ, i, q);
            else
                msg2 ← pop();
                if msg2 = (AH, i, p) then move(p, i);
                else msg2 = (RT, i, j) then
                    send (sorry, i) to j;
        msg = (AC, i, p)
            cleared_port[p] ← 0;
            move(p, i);
        msg = (AH, i, p)
            move(p, i);
        msg = (ST, i, j)
            back();
        msg = ∅
            k ← k + 1;
            initialisation(k);
end
```

```
decide()
begin
    if node v is clear
    or there is another searcher ℓ < i at v then
        if i = 1 then
            j ← 2;
            terminated ← true;
        else j ← 1;
        push(JJ, i, j);
        join(j);
    else if ∃ unique contaminated port p then
            clear_edge(CC, i, p);
        else if i ≠ k then
                next_searcher(i);
            else back();
end

join(searcher_ID j)
begin
    state ← (HELP, j);
    if j is present at v then
        if v is clean then
            if i = 1 and terminated and j = k
                then "The graph is clear";
            else back();
        else
            if i = 1 then terminated ← false;
            Let q be the smallest contaminated port;
            if j < i then
                push(ST, i, j);
                send (help, i) to j;
            else
                clear_edge(CJ, i, q);
    else
        p ← direction[j];
        if p = 0 do p ← direction[0];
        move(p, i);
end
```

Figure 4: Procedures `back`, `decide`, and `join`.

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 *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 behaviour of a_i then depends on msg, 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 $msg = (RT, i, j)$: it means that searcher a_j had sent a message to a_i to let it 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 $(\texttt{sorry}, i)$ to a_j in order to decline, and to let a_i the possibility to do something else.

- Case $msg = (JJ, i, j)$: 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_ℓ), by applying $join(\ell)$.

  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 $msg = (CC, i, p)$: 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 $msg = (CJ, i, p)$: 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 tried to clear yet (i.e., a contaminated edge with port number $q > p$), then searcher a_i applies Procedure $clear\_edge(CJ, i, q)$ to clear this edge (CJ 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 $msg2$. If $msg2 = (AH, i, q)$, 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 $msg2 \neq (AH, i, q)$, then the only possible case is $msg2 = (RT, i, j)$, 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 $(\texttt{help}, j)$ 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 $(\texttt{sorry}, i)$ to a_j.

- Case $msg = (AC, i, p)$: 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 *back*(), a_i backtracks, i.e., goes back through the same edge letting this edge be recontaminated.
- Case $msg = (AH, i, p)$: 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 *back*(), a_i backtracks its move by going back through the same edge it came from.
- Case $msg = (ST, i, j)$: 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 $(\texttt{sorry}, j)$, meaning that a_j could not do anything more. Thus, a_i applies *back*() in order to backtrack again.
- Case $msg = \emptyset$: 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 *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 $clear\_edge(CJ, i, q)$ to clear the edge (CJ meaning that this cleaning can be done thanks to the presence of another searcher). If $i > j$, then searcher a_i sends $(\texttt{help}, i)$ 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 going to the homebase, which is possible thanks to the local vector `direction`. Procedure *join*() 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*, $Q_{virtual}$, 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 $(a_i \to a_j, p)$, to be interpreted as follows.

- If $i \neq j$, then $(a_i \to a_j, p)$ means that a_i leaves its current node by port p with the objective of joining a_j;
- The move $(a_i \to a_i, p)$ 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 $(a_i \to a_i, p)$, and if $i \neq j$ then the extended move (a_i, a_j, p) is equivalent to the sequence of moves

$$(a_i \to a_j, p_1), (a_i \to a_j, p_2), \ldots, (a_i \to a_j, p_\ell), (\min\{a_i, a_j\} \to \min\{a_i, a_j\}, p)$$

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.

$Q_{virtual}$ is updated in the following way. At every execution of the Procedure *move*(), we push or pop a move in $Q_{virtual}$ as follows. If a_i applies $move(p, i)$ during the execution of Procedure $clear\_edge(X, i, p)$, then the move $(a_i \to a_i, p)$ is pushed in $Q_{virtual}$. If a_i applies $move(p, i)$ during the execution of Procedure $join(j)$, then the move $(a_i \to a_j, p)$ is pushed in $Q_{virtual}$, where p is the port number set during the execution of *join*(), before the call of procedure *move*(). Finally, if a searcher applies $move(p, i)$ during the execution of Procedure *back*(), then $Q_{virtual}$ is popped.

With this definition of $Q_{virtual}$, we show that the stack Q of the centralized algorithm $\mathcal{A}$, and the virtual stack $Q_{virtual}$ 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:

- $Q_{virtual}$ is *strongly equivalent* to Q if $Q_{virtual} = S_1 | \cdots | S_r$, such that, for any $1 \leq j \leq r$, S_j is a sequence of moves equivalent to M_j.
- $Q_{virtual}$ is *weakly equivalent* to Q if $Q_{virtual} = S_1 | \cdots | S_r | S_{r+1}$, such that, for any $1 \leq j \leq r$, S_j is a sequence of moves equivalent to M_j, and $S_{r+1} = (a_i \to a_{i'}, p_1), (a_i \to a_{i'}, p_2), \ldots, (a_i \to a_{i'}, p_\ell)$ where $p_1, \cdots, p_\ell$ is a sequence of port numbers corresponding to a path between a searcher a_i and a searcher $a_{i'}$, 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 $Q_{virtual}$ are only weakly equivalent, then the strategy associated to $Q_{virtual}$ 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 latter version of equivalence is important in our proof. The two stacks $Q_{virtual}$ 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 $(a_i \to a_i, p)$ is pushed in or popped out $Q_{virtual}$.

Formally, we prove that, for any $t \geq 0$, the virtual stack $Q_{virtual}$ 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 of steps. We assume that the centralized stack Q and the virtual stack $Q_{virtual}$ 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 $Q_{virtual}$ just cleared an edge e. The first case corresponds to the graph being entirely clear. Otherwise there are 3 cases: (1) a searcher can clear a new 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 other 3 cases depending on the way e has been cleared. Thus Group A yields 7 cases in total.

- Group B: Q and $Q_{virtual}$ 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', has to be canceled). In the former case, there are 3 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 4 subcases depending on the way e' had been cleared. Thus Group B yields 7 other cases.

The proof consists in a careful analysis of each of these 14 cases. Before analysing these 14 cases, we first prove that the stacks computed at the first step of both algorithms are equivalent. Initially, both Q and $Q_{virtual}$ are empty. In `dist_search`, a_1 executes the *decide* function.

- If $deg(u_0) = 1$, then Algorithm `dist_search` pushes $(CC, 1, 1)$ and $(AC, 1, p)$ on the distributed whiteboards, while $(a_1 \to a_1, 1)$ is pushed in $Q_{virtual}$. During the first execution of the while loop in Algorithm $\mathcal{A}$, since $\deg(u_0) = 1$, $Q = ((a_1, a_1, 1))$. Moreover, in both cases, the cleared subgraph is one edge (u_0, w) incident to u_0 with a_1 at node w, and all the others at node u_0.

- If $deg(u_0) > 1$ and $k = 1$, then both algorithms state that another searcher is needed. The two stacks remain empty and only u_0 is clear.

- If $deg(u_0) > 1$ and $k > 1$, then Algorithm `dist_search` pushes $(ST, 1, 2)$, $(RT, 2, 1)$, $(JJ, 2, 1)$, $(ST, 2, 1)$, $(RT, 1, 2)$, $(CC, 1, 1)$ and $(AC, 1, p)$ on the distributed whiteboards, while $(a_1 \to a_1, 1)$ is pushed in $Q_{virtual}$. Algorithm $\mathcal{A}$ pushes $(a_2, a_1, 1)$ 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 $Q_{virtual}$ are equivalent. We prove, for the 14 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 groups A and B.

5.1 Group A

Group A assumes that Q and $Q_{virtual}$ have been reached by clearing an edge. Let S and $S_{virtual}$ be the states of Q and $Q_{virtual}$ at this step of both algorithms. Since, Q and $Q_{virtual}$ has been reached by clearing an edge, they are strongly equivalent. Thus, there exist a sequence S' of valid extended moves, and a sequence $S'_{virtual}$ of moves, with S' and $S'_{virtual}$ strongly equivalent, and there exist an extended move M, and a sequence M' of moves, with M' equivalent to M, such that $S = S'|M$ and $S_{virtual} = S'_{virtual}|M'$.

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 $(\texttt{move}, j)$ to a_1, who received $(\texttt{move}, j)$ 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 $(JJ, 1, 2)$ and applies Procedure *join*(2) after having set *terminated* to *true*. Applying Procedure *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 $(1 \rightarrow 2, p_1)$ in $Q_{virtual}$. 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 $(AH, 1, q_1)$ on the whiteboard, and applies again the *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 $\{v_i, v_{i+1}\}$ at v_i (resp., v_{i+1}). At every node v_i, $i \geq 2$, searcher a_1 writes $(AH, 1, q_{i-1})$ during the execution of *join*(). In $Q_{virtual}$, we push $(a_1 \rightarrow a_2, p_i)$ for $i = 1, \ldots, t-1$. Since v_t is clear, searcher a_2 does not need help, and thus a_1 applies Procedure *back*(). Therefore, it pops $(AH, 1, q_{t-1})$ 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 $(AH, 1, q_{i-1})$. 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 $Q_{virtual}$. Eventually, a_1 is back at v_1 is state BACKTRACK. At v_1, searcher a_1 applies Procedure *back*(), and thus pops $(JJ, 1, 2)$ 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 *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 $Q_{virtual} = S_{virtual}|(a_1 \rightarrow a_k, r_1)|\cdots|(a_1 \rightarrow a_k, r_\ell)$ 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 $Q_{virtual}$ were equivalent before these sequence of moves, the new state S of stack Q, and the new state $S_{virtual}|(a_1 \rightarrow a_k, p_1)|\cdots|(a_1 \rightarrow a_k, p_\ell)$ of stack $Q_{virtual}$, 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 $S|M$). 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' of moves that is equivalent to M, and such that, after the next step, the state of $Q_{virtual}$ becomes $S_{virtual}|M'$.

- Case A.2.1: M is of type (a_i, a_i, p).

  We consider only the case $i > 1$, the case $i = 1$ follows easily this study. In this case, 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 single contaminated edge. Executing Algorithm `dist_search`, a_1 applies the *decide*() procedure. Applying this procedure, searcher a_1 writes $(ST, 1, 2)$ and sends $(\texttt{move}, 1)$ to a_2. For any $2 \leq j \leq i$, searcher a_j receives $(\texttt{move}, j-1)$ from a_{j-1} and writes $(RT, j, j-1)$. Applying Procedure *decide*(), searcher a_j writes $(ST, j, j+1)$ and sends $(\texttt{move}, j)$ to a_{j+1}. When a_i receives the message $(\texttt{move}, i-1)$ from a_{i-1}, it applies the *decide*() procedure that calls Procedure $clear\_edge(CC, i, p)$. Thus, a_i writes (CC, 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 (AC, i, q) and $(ST, i, 1)$ on the whiteboard of v. We push the move $(a_i \rightarrow a_i, p)$ in $Q_{virtual}$. Thus, the state of $Q_{virtual}$ becomes $S_{virtual}|(a_i \rightarrow a_i, p)$ which is strongly equivalent to the state $S|(a_i, a_i, p)$ of Q.

- Case A.2.2: M is of type (a_1, a_j, p) 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 $(JJ, 1, 2)$ and applies Procedure $join(2)$ after having set *terminate* to *true*. Similarly 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 *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 reaches a_j at v_j, the state variable *terminated* of a_1 is set to $false$. The virtual stack satisfies $Q_{virtual} = S_{virtual}|(a_1 \rightarrow a_j, r_1)|\cdots|(a_1 \rightarrow a_j, r_\ell)$ 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 $clear\_edge(CJ, 1, p)$, that is, he writes $(CJ, 1, p)$ and clears the corresponding edge. The move $(a_1 \rightarrow a_1, p)$ is pushed in $Q_{virtual}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{virtual}$ is $S_{virtual}|(a_1 \rightarrow a_j, p_1)|\cdots|(a_1 \rightarrow a_j, p_{|P|})|(a_1 \rightarrow a_1, p)$, which is strongly equivalent to the state $S|(a_1, a_j, p)$ of Q.

- Case A.2.3: M is of type (a_i, a_1, p) with $i > 1$.

  In this case, for any $\ell < i$, searcher a_ℓ is alone at a vertex v_ℓ 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_ℓ, with $\ell < i$. In the distributed algorithm `dist_search`, a_1 applies Procedure *decide*(). Applying this procedure, searcher a_1 writes $(ST, 1, 2)$ and sends $(\texttt{move}, 1)$ to a_2. For any $2 \leq j \leq i$, searcher a_j receives $(\texttt{move}, j-1)$ from a_{j-1} and writes $(RT, j, j-1)$. Applying Procedure *decide*(), searcher a_j writes $(ST, j, j+1)$ and sends $(\texttt{move}, j)$ to a_{j+1}. When a_i receives the message $(\texttt{move}, i-1)$ from a_{i-1}, it applies the *decide*() procedure 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 $\{w_j, w_{j+1}\}$ at w_j (resp., w_{j+1}). At every node w_j, $j \geq 2$, searcher a_i writes (AH, i, q_{j-1}) during the execution of $join()$. In $Q_{virtual}$, we push $(a_i \to a_1, p_j)$ for $j = 1, \ldots, r-1$. At w_r, searcher a_i writes $(ST, i, 1)$ and sends the message $(\texttt{help}, i)$ to searcher a_1. Then, a_1 writes $(RT, 1, i)$ and $(CJ, 1, p)$, and clears the corresponding edge. The move $(a_1 \to a_1, p)$ is pushed in $Q_{virtual}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{virtual}$ is $S_{virtual}|(a_i \to a_1, p_1)|\cdots|(a_i \to a_1, p_r)|(a_1 \to a_1, p)$, which is strongly equivalent to the state $S|(a_i, a_1, p)$ of Q.

5.1.3 Case A.3

Case A.3 assumes that there does not exist a valid extended move according to the current state of the stack Q. Therefore, Algorithm $\mathcal{A}$ pops the last executed extended move M from $S = S'|M$. 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 $S_{virtual} = S'_{virtual}|M'$ with M' a sequence of moves equivalent to M. There are three cases two be considered:

- **Case A.3.1**: $M = (a_i, a_i, p)$. In this case, M' is the 1-element sequence $(a_i \to a_i, p)$.
- **Case A.3.2**: There is $k \geq s > i$ such that $M = (a_i, a_s, p)$. In this case, searcher a_i leaves a vertex, say v_i to join searcher a_s at vertex 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 $\{w_j, w_{j+1}\}$ at w_j (resp., w_{j+1}). By the induction hypothesis, $M' = (a_i \to a_s, p_1)|\cdots|(a_i \to a_s, p_r)|(a_i \to a_i, p)$
- **Case A.3.3**: There is $k \geq s > i$ such that $M = (a_s, a_i, p)$. In this case, searcher a_s leaves a vertex, say v_s to join searcher a_i at vertex 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 $\{w_j, w_{j+1}\}$ at w_j (resp., w_{j+1}). By the induction hypothesis, $M' = (a_s \to a_i, p_1)|\cdots|(a_s \to a_i, p_r)|(a_i \to a_i, p)$

Searcher a_i has arrived at the node w by port number, say q, and a_i has pushed (AC, i, q). If $i > 1$, a_i has also pushed $(ST, i, 1)$ and sent $(\texttt{move}, i)$ to a_1, who has pushed $(RT, 1, i)$ 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 $(ST, j, j+1)$ and sends $(\texttt{move}, j)$ to a_{j+1} by applying Procedure $next\_searcher(j)$ in Procedure *decide*(). Then searcher a_{j+1} pushes $(RT, j+1, j)$ at its current vertex before applying Procedure *decide*() too. When a_k receives the message $(\texttt{move}, k-1)$ from a_{k-1}, it applies Procedure *decide*() that calls Procedure *back*(). Then, a_k pops $(RT, k, k-1)$ and sends $(\texttt{sorry}, k)$ to searcher a_{k-1}. For any $j > 1$, a_j receives $(\texttt{sorry}, j+1)$ from a_{j+1}. Then searcher a_j applies Procedure *back*() that pops $(ST, j, j+1)$, then pops $(RT, j, j-1)$, and sends $(\texttt{sorry}, j)$ to a_{j-1}. When a_1 receives $(\texttt{sorry}, 2)$, a_1 applies Procedure *back*() that pops $(ST, 1, 2)$, then pops $(RT, 1, i)$, and sends $(\texttt{sorry}, 1)$ to a_i. By applying Procedure *back*(), a_i pops $(ST, i, 1)$, then (AC, 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 $(a_i \to a_i, p)$ is popped from $Q_{virtual}$. 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 $(a_i \to a_i, p)$. Thus, the new state of $Q_{virtual}$ depends on the case:

- Case A.3.1: $S'_{virtual}$
- Case A.3.2: $S'_{virtual}|(a_i \to a_s, p_1)|\cdots|(a_i \to a_s, p_\ell)$
- Case A.3.3: $S'_{virtual}|(a_s \to a_i, p_1)|\cdots|(a_s \to a_i, p_\ell)$

Therefore, $S'_{virtual}$ is equivalent to the state S' of Q (strongly equivalent in case A.3.1, and weakly equivalent in both other cases).

5.2 Group B

Cases in Group B assumes that Q and $Q_{virtual}$ 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 $S_{virtual}$ be the states of Q and $Q_{virtual}$ at this step of both algorithms. 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 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*().

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 $M' \succ M$ in Q. In the following, M' can be of three different types defined bellow. Let us prove that Algorithm `dist_search` executes a sequence of moves equivalent to M'. There are 3 cases depending on the type of the extended move M.

- **Case B.1.1**: $M = (a_i, a_i, p)$. This case occurs after the removal operation as in case $A.3.1$. Thus S and $S_{virtual}$ are actually strongly equivalent. In this case, there exist $i < j \leq \ell \leq k$ and $0 \leq q \leq n$ such that there exists an extended move $M' = (a_j, a_\ell, q)$ 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 $(ST, i, i+1)$ at v, and sends $(\texttt{move}, i)$ to a_{i+1}. In the same way as for Case A.2, the message $(\texttt{move}, j-1)$ is received by a_j which can perform a valid extended move. As for Case A.2, searcher a_j performs this move and we push in $Q_{virtual}$ a sequence of moves equivalent to M'. Thus, both stacks remains strongly equivalent.

- **Case B.1.2**: $M = (a_i, a_j, p)$ with $i < j$.

  This case occurs after the removal operation as in case $A.3.2$. Thus S and $S_{virtual}$ are weakly equivalent. More precisely, there exit a state $S'_{virtual}$ that is strongly equivalent to S and a sequence $(p_1, \ldots, p_{t-1})$ of port numbers, such that $S_{virtual} = S'_{virtual}|(a_i \to a_j, p_1)|\cdots|(a_i \to a_j, p_{t-1})$. Let v_i (resp., v_j) be the vertex where searcher a_i (resp., a_j) stands in the configuration associated to $S'_{virtual}$. Note that $v_j = v$. $(p_1, \ldots, p_{t-1})$ 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 $S_{virtual}$ is got from the configuration associated to $S'_{virtual}$ (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 $\{w_{t-1}, v_j\}$. Recall that, when it had joined a_j at

v_j, searcher a_i had written (AH, i, q). Then, since $i < j$, a_i had written (CJ, i, p) and had cleared the edge.

To prove that both stacks remain equivalent, we consider the type of the extended move M'. There are three cases:

- Case B.1.2.a: there is a port number $r \leq n$ of v_j, larger than p such that the corresponding edge is contaminated. In this case, $M' = (a_i, a_j, r)$.
- Case B.1.2.b: there is a searcher with ID $\ell \leq k$, larger than j, at vertex v_ℓ, and a port number $r \leq n$ of v_ℓ such that the corresponding edge is contaminated. In this case, $M' = (a_i, a_\ell, r)$.
- Case B.1.2.c: there is a searcher with ID $\ell \leq k$, larger than i, at vertex v_ℓ, that can preform a valid extended move. That is, there exist $\ell < u \leq k$ and $r \leq n$ such that $M' = (a_\ell, a_u, r)$.

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 *back*(), a_i pops (CJ, 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.1.2.a:

- Algorithm $\mathcal{A}$ pushes $M' = (a_i, a_j, r)$ in Q. Searcher a_i pushes (CJ, i, r) at v_j and clear the corresponding edge, arriving at a new node by port, say o. Searcher a_i pushes (AC, i, o) and $(ST, i, 1)$ at the new node, and then send message $(\texttt{move}, i)$ to a_1. We push $(a_i \rightarrow a_i, r)$ in $Q_{virtual}$. Thus, the state of Q is $S|M'_1$ and the state of $Q_{virtual}$ is $S'_{virtual}|(a_i \rightarrow a_j, p_1)|\cdots|(a_i \rightarrow a_j, p_{t-1})|(a_i \rightarrow a_i, r)$. 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 (AH, 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 BACKTRACK, and thus pops the local stack, that contains (AH, i, q_f) 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 $(a_i \rightarrow a_j, p_f)$ that we had previously pushed in $Q_{virtual}$. Eventually, a_i is back at v_i in state BACKTRACK. At this stage of the execution of `dist_search`, the current state of $Q_{virtual}$ is $S'_{virtual}$ that is strongly equivalent to S. Then, by applying Procedure *back*(), a_i pops (JJ, i, j). Then, Algorithm `dist_search` checks whether searcher a_i can help a searcher with ID larger than j. By applying Procedure *back*(), a_i pushes $(JJ, i, j+1)$ and applies Procedure $join(j+1)$. Similarily to the Case A.1, this procedure asks a_i to try helping every possible searcher a_t, for $j+1 \leq k$. Let us assume that there is a searcher with ID $\ell \leq k$, larger than j, at vertex v_ℓ, and a port number $r \leq n$ of v_ℓ such that the corresponding edge is contaminated. This corresponds to the Case B.1.2.b:

- Algorithm $\mathcal{A}$ pushes $M' = (a_i, a_\ell, r)$ in Q. For any $f = j+1, \ldots, \ell-1$, since searcher a_f does not need help, searcher a_i applies Procedure *back*() after having reached a_f. Then a_i goes back to v_i and applies$join(f+1)$ (cf., Case A.1). When a_i eventually reachs a_ℓ at v_ℓ, the virtual stack satisfies $Q_{virtual} = S_{virtual}|(a_i \rightarrow a_\ell, p_1)|\cdots|(a_i \rightarrow a_\ell, p_t)$ where $p_1, \ldots, p_t$ is the sequence of port numbers from v_i to v_ℓ. Then, searcher

a_i applies Procedure *clear_edge*(CJ, i, r), that is, he writes (CJ, i, r) and clears the corresponding edge. The move $(a_i \to a_i, r)$ is pushed in $Q_{virtual}$. Thus, the smallest valid extended move is performed in both algorithms. Moreover, after this step, the state of $Q_{virtual}$ is $S_{virtual}|(a_i \to a_\ell, p_1)| \cdots |(a_i \to a_\ell, p_t)|(a_i \to a_i, r)$, which is strongly equivalent to the state $S|(a_i, a_\ell, r)$ of Q.

Now, we consider the case where there is no $\ell > j$ such that searcher a_ℓ stands at a vertex v_ℓ incident to a contaminated edge. Thus, a_i reachs back v_i after having tried to help any searcher a_ℓ, for $j < \ell \leq k$ (by iteratively applying Procedure *join*() as for the previous case). At this stage of the execution `dist_search`, the current state of $Q_{virtual}$ is $S'_{virtual}$ that is strongly equivalent to S (the current state of Q). When a_i reachs back v_i, it pops (JJ, i, k). Thus, Procedure *back*() calls Procedure *next_searcher*(i). Therefore, a_i pushes $(ST, i, i+1)$ and sends $(\texttt{move}, i)$ to searcher a_{i+1}. Let us assume that there is a searcher with ID $\ell \leq k$, larger than i, at vertex v_ℓ, 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 $M' = (a_\ell, a_u, r)$ in Q. As for the case $A.2.3$, for any $i \leq f \leq \ell$, searcher a_f receives $(\texttt{move}, f-1)$ from a_{f-1} and writes $(RT, f, f-1)$. Applying Procedure *decide*(), searcher a_f writes $(ST, f, f+1)$ and sends $(\texttt{move}, f)$ to a_{f+1}. When a_ℓ receives the message $(\texttt{move}, \ell-1)$ from $a_{\ell-1}$, it applies the *decide*() procedure. Then the move m' is performed as for the case A.2. Thus, both stacks become strongly equivalent.

- **Case B.1.3**: $M = (a_j, a_i, p)$ with $i < j$.

This case occurs after the removal operation as in case $A.3.3$. Thus S and $S_{virtual}$ are weakly equivalent. More precisely, there exit a state $S'_{virtual}$ that is strongly equivalent to S and a sequence $(p_1, \ldots, p_{t-1})$ of port numbers, such that $S_{virtual} = S'_{virtual}|(a_j \to a_i, p_1)| \cdots |(a_j \to a_i, p_{t-1})$. Let v_i (resp., v_j) be the vertex where searcher a_i (resp., a_j) stands in the configuration associated to $S'_{virtual}$. Note that $v_i = v$. $(p_1, \ldots, p_{t-1})$ is exactly the sequence of port numbers that searcher a_j has followed along a path from v_j to v_i. Let $P = w_1, \cdots, w_t$ be this path, with $w_1 = v_j$ and $w_t = v_i$. More precisely, the configuration associated to $S_{virtual}$ is got from the configuration associated to $S'_{virtual}$ (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 $(p_1, \ldots, p_{t-1})$ of port numbers. Let q be the port number of v_i corresponding to the edge $\{w_{t-1}, v_i\}$. Recall that, when it had joined a_i at v_i, searcher a_j had written (AH, j, q). Then, since $i < j$, a_j had pushed (ST, j, i) at v_i and sent $(\texttt{help}, j)$ to searcher a_i. Then, searcher a_i has pushed (RT, i, j) and (CJ, i, p) at v_i, and had cleared the edge.

To prove that both stacks remain equivalent, we consider the type of the extended move M'. There are three cases:

- Case B.1.3.a: there is a port number $r \leq n$ of v_i, larger than p such that the corresponding edge is contaminated. In this case, $M' = (a_j, a_i, r)$.
- Case B.1.3.b: there is a searcher with ID $\ell \leq k$, larger than i, at vertex v_ℓ, and a port number $r \leq n$ of v_ℓ such that the corresponding edge is contaminated. In this case, $M' = (a_j, a_\ell, r)$.
- Case B.1.3.c: there is a searcher with ID $\ell \leq k$, larger than j, at vertex v_ℓ, that can preform a valid extended move. That is, there exist $\ell < u \leq k$ and $r \leq n$ such that $M' = (a_\ell, a_u, r)$.

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 *back*(), a_i pops (CJ, 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*(). a_i pops (CJ, i, p), then (RT, i, j), and sends $(messsorry, i)$ to searcher a_j. Then, searcher a_j applies Procedure *back*(). Therefore, it pops (ST, j, i) and (AH, 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 (AH, j, q_f) 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 $(a_j \to a_i, p_f)$ that we had previously pushed in $Q_{virtual}$. Eventually, a_j is back at v_j in state BACKTRACK. At this stage of the execution of `dist_search`, the current state of $Q_{virtual}$ is $S'_{virtual}$ that is strongly equivalent to S. Then, by applying Procedure *back*(), a_j pops (JJ, 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' and a sequence of valid extended moves S' such that $S = S'|M'$. In the former case, Algorithm $\mathcal{A}$ claims that another searcher is required. In the latter case, Algorithm $\mathcal{A}$ pops M' from Q. Let us prove it is also the case for Algorithm `dist_search`. There are four cases according to whether $S = \emptyset$ or what is the type of M'.

- **Case B.2.1** If $S = \emptyset$, there are two case. 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 (a_k, a_{k-1}, p) 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 $((ST, 1, 2), (RT, 2, 1), \cdots, (ST, i, i+1), (RT, i+1, i), \cdots, (ST, k-1, k), (RT, k, k-1), (JJ, k, k-1), (ST, k, k-1), (RT, k-1, k), (CJ, k-1, p))$. Thus, $S_{virtual} = \emptyset$. Thus, Q and $Q_{virtual}$ are strongly equivalent. Moreover, it is easy to check that Procedure *back*() asks for a $(k+1)^{th}$ 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 $f = (v', w')$ be the edge cleared by the move M'. Let $s \leq k$ be the Id of the searcher that has cleared f, arriving by port r' of w'. Let r be the port number of v' corresponding to f. Let us consider the three possible types for the move M':

- **Case B.2.2** $M' = (a_s, a_s, r)$. In this case, there is a sequence of valid moves $S'_{virtual}$ strongly equivalent to S', and a sequence of valid moves $M_{virtual}$ equivalent to M such that $S_{virtual} = S'_{virtual}|(a_s \to a_s, r)|M_{virtual}$.

- **Case B.2.3** There is $s' < s$ such that $M' = (a_s, a_{s'}, r)$. In this case, there is a sequence of valid moves $S'_{virtual}$ strongly equivalent to S', a sequence of valid moves $M_{virtual}$ equivalent to M and a sequence $(p_1, \ldots, p_{t-1})$ of port numbers, such that $S_{virtual} = S'_{virtual}|(a_s \to a_{s'}, p_1)| \cdots |(a_s \to a_{s'}, p_{t-1})|(a_s \to a_s, r)|M_{virtual}$.

- **Case B.2.4** There is $s' < s$ such that $M' = (a_{s'}, a_s, r)$. In this case, there is a sequence of valid moves $S'_{virtual}$ strongly equivalent to S', a sequence of valid moves $M_{virtual}$ equivalent to M and a sequence $(p_1, \ldots, p_{t-1})$ of port numbers, such that $S_{virtual} = S'_{virtual}|(a_{s'} \to a_s, p_1)| \cdots |(a_{s'} \to a_s, p_{t-1})|(a_s \to a_s, r)|M_{virtual}$.

After having cleared f, searcher a_s has pushed (AH, s, r'), then $(ST, s, 1)$, and sent $(\texttt{move}, s)$ to searcher a_1. Then a_1 applies the *decide*() procedure. Applying this procedure, searcher a_1 writes $(ST, 1, 2)$ and sends $(\texttt{move}, 1)$ to a_2. For any $2 \leq j \leq i-1$, searcher a_j receives $(\texttt{move}, j-1)$ from a_{j-1} and writes $(RT, j, j-1)$. Applying Procedure *decide*(), searcher a_j writes $(ST, j, j+1)$ and sends $(\texttt{move}, j)$ to a_{j+1}. When a_i receives the message $(\texttt{move}, i-1)$ from a_{i-1}, it pushes $(RT, i, i-1)$ at its current vertex v_i, and applies the *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 $p \leq n$ be the port number of v corresponding to e. Since there are no valid extended move larger than M, only three cases are possible:

- $M = (a_i, a_i, p)$ 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 $Q_{virtual}$ 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 Procedure *back*(), a_i pops (CC, i, p), pushes $(ST, i, i+1)$ at v_i, and sends $\texttt{move}, i)$ to searcher a_{i+1}. For any $i+1 \leq j \leq k$, searcher a_j receives $(\texttt{move}, j-1)$ from a_{j-1} and writes $(RT, j, j-1)$. Applying Procedure *decide*(), searcher a_j writes $(ST, j, j+1)$ and sends $(\texttt{move}, j)$ to a_{j+1}. When a_k receives the message $(\texttt{move}, k-1)$ from a_{k-1}, searcher a_k applies Procedure *decide*(), then Procedure *back*(). Searcher a_k pops $(ST, k, k-1)$ and sends $(\texttt{sorry}, k)$ to a_{k-1}. For any $k > j > i$, searcher a_j receives $(\texttt{sorry}, j+1)$ from a_{j+1} and pops $(ST, j, j+1)$. Applying Procedure *back*(), searcher a_j pops $(RT, j-1, j)$ and sends $(\texttt{sorry}, j)$ to a_{j-1}. When searcher a_i receives $(\texttt{sorry}, i+1)$, it pops $(ST, i, i+1)$, and then pops $(RT, i, i-1)$ from the local stack of v_i.

- $i < k$, $M = (a_i, a_k, p)$ and for any $i < j \leq k$, searcher a_k stands alone at a vertex, say v_j. In this case, there exit a state $S'_{virtual}$ that is strongly equivalent to S and a sequence $(p_1, \ldots, p_{t-1})$ of port numbers, such that $S_{virtual} = S'_{virtual}|(a_i \to a_k, p_1)| \cdots |(a_i \to a_k, p_{t-1})$. Note that in the configuration associated to $S'_{virtual}$ (resp., to $S_{virtual}$), searcher a_i stands at v_i (resp., v). Searcher a_k stands at v in both configurations. $(p_1, \ldots, p_{t-1})$ is exactly the sequence of port numbers that searcher a_i has followed along a path from v_i to v. 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 $\{v_{f-1}, w_f\}$. Recall that, a_i had followed the path P to join a_k. Then, searcher a_1 had written (AH, i, q_t). Then, since $i < j$, a_i had written (CJ, 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 (AH, i, q_t) 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 (AH, i, q_f). As a result, it goes to w_{f-1} using port q_{f-1}. Simultaneously, we pop $(a_i \to a_k, p_f)$ that we had previously pushed in $Q_{virtual}$. Eventually, a_i is back at v_i in state BACKTRACK. At this stage of the execution of `dist_search`, the current state of $Q_{virtual}$ is $S'_{virtual}$ that is strongly equivalent to S. Then, by applying Procedure *back*(), a_i pops (JJ, i, k), pushes $(ST, i, i+1)$ 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 $(RT, j, j-1)$. Applying Procedure *decide*(), searcher a_j writes $(ST, j, j+1)$ 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 Procedure *decide*(), then Procedure *back*(). Searcher a_k pops $(ST, k, k-1)$ 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 $(ST, j, j+1)$. Applying Procedure *back*(), searcher a_j pops $(RT, j-1, j)$ and sends $($`sorry`$, j)$ to a_{j-1}. When searcher a_i receives $($`sorry`$, i+1)$, it pops $(ST, i, i+1)$, and then pops $(RT, i, i-1)$ from the local stack of v_i.

- $i = k$ and $M' = (a_i, a_{k-1}, p)$. Similarily to the previous case, we can prove that there is a round of the execution of `dist_search` when a_i pops $(RT, i, i-1)$ 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 $(RT, i, i-1)$ from the local stack of v_i. Moreover, at this round, Q and $Q_{virtual}$ are strongly equivalent.

If $i = k$, searcher a_k applies Procedure *back*(). Otherwise, a_i calls Procedure *next_searcher*(i), pushes $(ST, i, i+1)$ and sends $($`move`$, i)$ to a_{i+1}. Then, for $i < j < k$, a_j pushes $(RT, j, j-1)$ and $(ST, j, j+1)$ at its current node, and sends $($`move`$, j)$ to a_{j+1}. When a_k receives message $($`move`$, k-1)$, it applies the *back*() procedure. Searcher a_k sends $($`sorry`$, k)$ to searcher a_{k-1}. Then, for $k \geq j > i$, a_j pops $(ST, j, j+1)$ and $(RT, j, j-1)$, and sends $($`sorry`$, j)$ to a_{j-1}. Finally, a_i receives $($`sorry`$, i+1)$ from searcher a_{i+1}, and applies Procedure *back*(). For any $i \geq j > s$, a_j pops $(ST, j, j+1)$ and $(RT, j, j-1)$, and sends $($`sorry`$, j)$ to a_{j-1}. Finally, a_s receives $($`sorry`$, s+1)$ from searcher a_{s+1} and applies Procedure *back*(). Then, searcher a_s pops (AC, s, r') from the local stack of w'. Then, it goes back to v' in state BACKTRACK, letting recontaminated the edge f. We pop $(a_s \to a_s, r)$ from $Q_{virtual}$. Thus, in Case B.2.2 (resp., B.2.3 and B.2.4), Q and $Q_{virtual}$ becomes strongly equivalent (resp., weakly equivalent).

We have proved, that in any case, both stack 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`*, at most $O(m \log n)$ bits are stored in any node's whiteboard.*

Proof. Recall that 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. Let t_1 and t_2 be two steps of the execution of protocol `dist_search` satisfying (1) an edge f is cleared during step t_1, (2) an edge e is cleared during step t_2, and (3) all edges that have been cleared between steps t_1 and t_2, have been recontaminated (i.e., the clearing of each of these edges has been backtracked).

Let $size(v,t)$ be the number of bits that are written on the whiteboard of vertex v at step t during the execution of protocol `dist_search`. We prove that, for any vertex $v \in V(G)$, $size(v,t_2) - size(v,t_1) = O(\log n)$.

Let us assume that the clearing of the edge e at step t_2 consists of the following sequence of moves. A searcher a_i, $1 \leq i \leq k$, joins another searcher a_j, $1 \leq j < i$, and searcher a_j clears edge e. This is the case where the number of traces is the greatest possible. Let v_i (resp., v_j) be the vertex where a_i (resp., a_j) is standing after the clearing of the edge f at step t_1. Note that if t_1 is the first step, then $v_i = v_j = v_0$. If t_1 is not the first step, then let a_ℓ, $\ell \geq 1$, be the searcher that has cleared f. After having cleared f, a_ℓ sends the message $(\texttt{move}, \ell)$ to a_1. Then, for any t, $1 \leq t \leq i-1$, the message $(\texttt{move}, t)$ is transmitted from searcher a_t to searcher a_{t+1} until message $(\texttt{move}, i-1)$ reachs a_i. By Procedure *next_searcher*(), if more than two searchers are on the same node, then only the two smallest ones receive the message. It is unnecessary to send the message to the other ones. Indeed, if the two smallest ones cannot do anything, then the others searchers cannot do neither. Thus, between the clearing of the two edges f and e, at most two traces of type (RT, ℓ, s) and two traces of type (ST, ℓ, s) are written on each whiteboard.

Let s_1 be the step during the execution of protocol `dist_search` when a_i receives $(\texttt{move}, i-1)$ from searcher a_{i-1}. Let s_2 be the step when a_i decides to try to help a_j. After the step s_1, searcher a_i first tries to help all searchers a_p, $p < j$. Since all these moves have been backtracked, all traces that have been written by searchers between steps s_1 and s_2 have been erased.

At step s_2, searcher a_i decides to join a_j and pushes (JJ, i, j) at v_i. By joining a_j, a_i pushes a trace (AH, i, j) on every whiteboard along the path between v_i and v_j. Finally, a_i sends message $(\texttt{help}, i)$ to a_j that clears the edge e. That is, searcher a_i pushes (ST, i, j) at v_j. Then, searcher a_j pushes (RT, j, i) and (CJ, j, p) at v_j. Finally a_j clears the edge e, and it pushes (AC, j, q) and $(ST, j, 1)$ at the other end of e.

To summarize, on every whiteboard, have been written at most three traces of type (RT, ℓ, s), three of type (ST, ℓ, s), and one for each of the types (JJ, i, j), (AH, i, j), (CJ, j, p) and (AC, j, q). Thus, when an extended move is performed, at most $O(1)$ traces are written on each whiteboard, i.e., at most $O(\log n)$ bits are written on each whiteboard.

Since there are m extended moves in total, at most $O(m \ \log n)$ bits are written on each 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 minimum number of searchers for this task. This result opens a wide field of investigations.

- First, it would be interesting to reduce the memory of the searchers, and the size of the whiteboards. More precisely, is it possible to achieve the same performances as protocol `dist_search` using searchers modeled as finite automata? (In our case, the searchers have $O(\log k)$-bit memory when k searchers are used). Also, what is the minimum size of the whiteboards that would enable achieving the same performances as protocol `dist_search`?

- Our distributed protocol eventually computes a monotone and connected search stratey using the optimal number of searchers. Nevertheless the strategy that is performed by our protocol is not monotone. In fact, the number of moves performed by our strategy may be exponential (recall that computing the monotone connected search number of a graph is NP-complete). Is it possible to design a distributed protocol that performs in polynomial time, at the price of relaxing some other constraint?

– For instance, it is known that the connected monotone search number of a graph is at most $\log n$ times its search number [21]. Thus, it makes sense to ask whether it is possible to design a distributed protocol that performs in polynomial time, using a number of searchers bounded by the search number of the network times a polylogarithmic function of n.
– Also, it would be interesting to analyse the impact of introducing some parallelism in the actions of the searchers, in order to reduce the time complexity of our protocol.
– Last but not least, what is the minimum quantity of information about the graph's topology that must be provided to the searchers, so that they can clear all graphs in a connected monotone way?

- Finally, is it possible to design a distributed protocol that eventually computes a non-monotone connected search strategy? Recall that 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.

References

[1] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. SIAM J. Alg. Disc. Meth. 8(2), pages 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, pages 33-49, 1991.

[7] D. Bienstock and P. Seymour. Monotonicity in graph searching. Journal of Algorithms 12, pages 239-245, 1991.

[8] R. Breisch. An intuitive approach to speleotopology. *Southwestern Cavers* VI(5), pages 72-78, 1967.

[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), LNCS vol. 3580, pages 335-346, 2005.

[10] S. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. SIAM J. on Computing 9(3), pages 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), LNCS 2180, pages 166-179, 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), pages 153-161, 2002.

[14] U. Feige, M. Hajiaghayi, and J. Lee. Improved approximation algorithms for minimum-weight vertex separators. In 37th ACM Symposium on Theory of Computing (STOC), pages 563-572, 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, 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), LNCS vol. 3142, Springer, pages 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 vol. 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 (LATIN), LNCS vol. 3887, pages 470-490, 2006.

[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): pages 224–245, 1993.

[25] F. S. Makedon and I. H. Sudborough, On minimizing width in linear layouts, Discrete Appl. Math., 23: pages 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): pages 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), pages 376-385, 2005.

[29] H.A. Rollik. Automaten in planaren Graphen. Acta Informatica 13: pages 287-298, 1980.

[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.

[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.