Distributed Chasing of Network Intruders

Lélia Blin[1], Pierre Fraigniaud[2], Nicolas Nisse[2], and Sandrine Vial[1]

[1] IBISC, University of Evry, 91000 Evry, France
[2] LRI, CNRS and Université Paris-Sud, 91405 Orsay, France

Abstract. 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, computed in the centralized and synchronous setting.

1 Introduction

Graph searching [18] 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). In addition to network security, graph searching has several other practical motivations, such as rescuing speleologists in caves [6] or decontaminating a set of polluted pipes [19]. 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 [4].

1.1 The problem

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 $\mathtt{s}(G)$ of a graph G is NP-hard [18], and NP-completeness follows from [5, 16]. Computing the search number is however polynomial for trees [17, 18], and the corresponding *search strategy* can be computed in linear time [20]. 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 [7], and use the fact that $pw/tw \leq O(\log n)$).

The graph searching problem has given rise to a vast literature, and several variants of the problem have been considered (see, e.g., [14, 15]). Nevertheless, from a distributed systems point of view, the existing solutions for the graph searching problem (cf., e.g., [17, 18, 20]) 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 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, 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 [9], and rings and tori [8]. In this paper, we address the problem in arbitrary topologies.

1.2 The model

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 is modeling a specific zone of the local node memory that is reserved for the purpose of exchanging information between software agents). At every node, the local whiteboard is assumed to be accessible by the searchers in fair mutual exclusion. Since the content of the whiteboard at every node accessible by the intruder is corruptible, it is the role of the searchers to protect information stored at nodes' whiteboards.

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

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

All searchers start from the same node u_0, called the *entrance* of the network, or the *homebase* of the searchers. This node u_0 is also a *source* of searchers, in the sense that if the current team of searchers realize that they are not numerous enough for clearing the network, then they can ask for a new searcher, that will appear at the source. Initially, one searcher spontaneously appears at the source. The size of the team will increase until it becomes large enough to clear the network. Basically, the searchers are aiming at expanding a cleared zone around their homebase u_0, that is at expanding a *connected* sub-network of the network G, containing u_0, until the whole network is clear. In particular, as the entrance u_0 of the network is a critical node, it has to be permanently protected from the intruder in the sense that the intruder must never be able to access it.

Among all search strategies, *monotone* ones play an important role. A monotone strategy insures that, once an edge has been cleared, it will always remain clear. Monotone strategies guaranty a polynomial number of moves: exactly one move for clearing every edge, plus few moves required by the searchers to set up their positions before clearing the next edge. In the connected setting, the corresponding graph searching parameter is called *monotone connected search number* starting at u_0 (cf., [2, 3, 13]), and is denoted by $\texttt{mcs}(G, u_0)$.

1.3 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. To the best of our knowledge, 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 self-computed by the searchers in an asynchronous environment uses a number of searchers very close to the optimal. 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 u_0, i.e., is equal to $\texttt{mcs}(G, u_0)+1$. It is known [13] that $\texttt{mcs}(G, u_0) \leq \texttt{s}(G)\lceil \log n \rceil$. Hence our protocol is optimal up to a logarithmic factor.

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

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

2 Main Result and Sketch of the Protocol

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 starting from the homebase u_0, and initially unaware of G. The main characteristics of* `dist_search` *are the following: (1)* `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 $\texttt{mcs}(G, u_0) = 1$; (2) Every searcher involved in the search strategy computed by* `dist_search` *uses $O(\log k)$ bits of memory; (3) During the execution of* `dist_search`*, at most $O(m \log n)$ bits of information are stored at every whiteboard.*

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.

Let us briefly sketch Protocol `dist_search` and 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 of a monotone connected search strategy using at most k searchers (a *step* of a monotone connected search strategy starting at node u_0 is the action consisting in moving a searcher along an edge, all searchers being initially at u_0). 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 $\leq k$ searchers. Otherwise, `dist_search` tries with $k + 1$ searchers. Note that this approach is similar to the (centralized) parametrized algorithms of the literature (cf., e.g., [1, 10, 11]). 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 $\texttt{mcs}(G, u_0)$, is required for avoiding deadlocks. It is also used to schedule the moves of the other searchers and to transmit few 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 Description of 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 to try, 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. It returns a boolean *possible*. If *possible* is true then clearing the network with k searchers is possible, in which case the stack Q returned by Algorithm $\mathcal{A}$ is valid, and contains a monotone connected search strategy clearing G with k searchers.

In Algorithm $\mathcal{A}$, the stack Q is initially empty, and only a_1 is placed at u_0. the other searchers $a_2, \ldots, a_k$ are *available*. In addition to the centralized stack Q,

Algorithm $\mathcal{A}$ uses a global variable *state* that takes two possible values CLEAR or BACKTRACK whose meaning will appear clear later on. Finally, Algorithm $\mathcal{A}$ uses a boolean variable *decided* that is false until either a monotone connected search strategy using k searchers clearing the network is discovered, or all possible monotone connected search strategies using k searchers have been considered. Hence the main while-loop of Algorithm $\mathcal{A}$ is based on the value of *decided* (cf. Figure 1). This main while-loop mainly contains two blocks of instructions. These blocks are executed depending on the value of *state* (CLEAR or BACKTRACK).

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., has resulted in the recontamination of an edge.

Let us first 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 also occupied by a 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 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.

```
Input: k ≥ 1 searchers a_1, a_2, ··· , a_k and a node u_0 of a graph G.
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;
         /* 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') ← min valid extended move according to Q 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
   endwhile
   return(possible, Q);
end.
```

Fig. 1. The Algorithm $\mathcal{A}$

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 Property of Algorithm $\mathcal{A}$

Lemma 1. *Algorithm $\mathcal{A}$ completes for $k = \texttt{mcs}(G, u_0)$, and then the stack Q describes a monotone connected search strategy for G starting at u_0 and using k searchers.*

Sketch of proof. First we prove that, after any execution of the *while*-loop, the state of the stack is valid. The main tools for the proof in then an ordering of the states of the stack. We order them the same way we ordered extended moves. Precisely, given $S = M_1|\cdots|M_r$ and $S' = M'_1|\cdots|M'_{r'}$, two states of the stack Q, $S \prec S'$ if and only if there exists $i \leq \min\{r, r'\}$ such that $M_i \prec$

M'_i and, for any $j < i$, $M_j = M'_j$. Also, let us say that a valid sequence of extended moves is *complete* if the corresponding search strategy clears the whole network. Consider $S = M_1|\ldots|M_r$ a sequence of extended moves corresponding to a partial execution of a search strategy using at most k searchers. We prove that 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. Based on these preliminary results, we prove that if $\texttt{mcs}(G, u_0) > k$ then Algorithm $\mathcal{A}$ returns $(false, \emptyset)$ for k. Conversally, we prove that if $\texttt{mcs}(G, u_0) = k$, and if S is the smallest complete sequence of valid extended moves corresponding to a monotone connected search strategy in G starting from u_0, then Algorithm $\mathcal{A}$ returns $(true, Q)$ for k, where Q is in state S. As a direct consequence of these results, we get that Algorithm $\mathcal{A}$ computes a minimal monotone connected search strategy starting from u_0 in G. □

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 $\texttt{mcs}(G, u_0)$ searchers. Using an additional searcher for implementing communications between the $\texttt{mcs}(G, u_0)$ other searchers, `dist_search` captures the intruder with $\texttt{mcs}(G, u_0) + 1$ searchers in total. Assuming that the searchers can communicate by exchanging messages is only for the purpose of simplifying the presentation. Moreover, for the sake of simplicity, we assume that two searchers on the same node can "see" each other. Obviously, this can be implemented with the whiteboards, but 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, (1) a searcher is in state CLEAR if it has just cleared an edge; (2) a searcher is in state BACKTRACK if it has just backtracked through an edge that it has previously cleared; and (3) a searcher is in state (HELP, j) if it is aiming at joining the searcher j to help him clearing the network (i.e., one of them will guard a node, while the other will clear an edge incident to this node).

The messages that searchers can exchange are of four types: `start`, `move`, `help` and `sorry`. (1) `start` is an initialization message that is only used to start Protocol `dist_search` (only agent a_1 receives this message, at the very beginning of the protocol execution). (2) If a searcher i receives a message (`move`, j) from some 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). (3) If a searcher a_i receives a message $(\texttt{help}, j)$ from some 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). (4) If a searcher a_i had received a message $(\texttt{move}, j)$ or $(\texttt{help}, j)$ from some searcher a_j and, after having possibly performed several actions, it turns out that these actions are useless, then a_i sends a message $(\texttt{sorry}, i)$ back to searcher a_j.

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

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: (1) (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; (2) (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; (3) (JJ, i, j) means that searcher a_i decided to join (J) the searcher a_j; (4) (RT, i, j) means that searcher a_i received (R) a message from searcher a_j; (5) (ST, i, j) means that searcher a_i decided to send (S) a message to searcher a_j; (6) (AC, i, p) means that searcher a_i arrived (A) at v by port p after clearing (C) the corresponding edge; (7) (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 protocol `dist_search`

The protocol `dist_search` organizes the movements of the searchers, and the messages exchanged between the searchers, in a specific order. Based on a lexicographic order of the searchers' actions, `dist_search` orders them in order to always execute the smallest action that can be performed. 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 needed help. Conversely, the searchers identify that k searchers are not sufficient to clear the graph when they are all occupying the homebase,

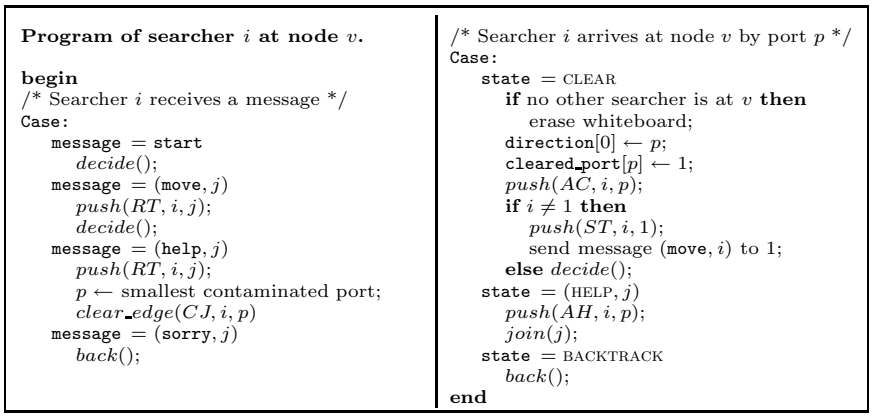

Fig. 2. Skeleton of Protocol `dist_search`

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 2-3. More precisely, Figure 2 describe the global behavior of a searchers, using subroutines described in Figure 3. 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*()).

If searcher a_i receives a message (`move`, j), then, by definition of such a message, it simply means that it is the turn of a_i to proceed. Therefore, a_i writes on the whiteboard of the node where it is currently standing that 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*(). Let us list all other cases depending on the message received by a_i. If a_i receives a message (`help`, j) then it means that a_j has just arrived at the same node as a_i to help him. Thus, a_i pushes (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 calling procedure $clear\_edge(CJ, i, p)$. 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 *back*() to figure out which searcher it can help next.

The action of searcher a_i arriving at some node v by port p depends on its local state. In state (HELP, j), a_i aims at joining a_j to help him clearing the network. Hence a_i pushes (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 *join*() to figure out what to do next in order to join a_j. 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.

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

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

```
next_searcher(searcher_ID i)
begin
    j ← i + 1;
    if i is not smallest searcher at 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
```

Fig. 3. Procedures `clear_edge`, `next_searcher` and `move`.

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. In state BACKTRACK, a_i simply calls procedure *back*() to carry on its backtracking. The case where a_i arrive 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 arases a whiteboard, it reset all local variables to 0, and the 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 *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. 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. Protocol `dist_search` always tries to perform the smallest action. This is in particular the role of procedure *next_searcher*(i) described on the right side of Figure 3. This procedure aims at determining which searcher a_j proceeds next. In the case where a_i is the searcher with smallest index occupying the node, $j = i + 1$. Otherwise, i.e., a_i is not the searcher with smallest index occupying the node, j is the smallest index $> i$ such that a_j is not occupying the same node as a_i. Once j is found, a_i offers to a_j to proceed next, by sending it a message (`move`, i). As always, a trace of this action is kept at the current node by pushing (ST, i, j) in the local stack. If there is no a_j with $j > i$ occupying a node different from the one occupied by 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). Procedures *decide*(), *back*(), and *join*() are avoided due to lack of space.

5 Sketch of Proof of `dist_search`

First, one can check that 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 *virtual stack*, $Q_{virtual}$, where we push or pop all the moves performed by the searchers, instead of pushing or popping them in and out of the distributed stacks.

Precisely, a *move* is a pair $(a_i \rightarrow a_j, p)$ to be interpreted as follows. If $i \neq j$, then $(a_i \rightarrow a_j, p)$ means that a_i leaves its current node by port p with the objective of joining a_j. The move $(a_i \rightarrow a_i, p)$ means that a_i leaves its current node by port p, for clearing the corresponding edge. Clearly, an extended move corresponds to a sequence of moves. From the interpretation above, the extended move (a_i, a_i, p) is identical to the move $(a_i \rightarrow a_i, p)$, and if $i \neq j$ then the extended move (a_i, a_j, p) is identical to the sequence of moves

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

where $p_1, \ldots, p_\ell$ is a sequence of port numbers corresponding to a clear path from the node occupied by a_i to 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) during the execution of Procedure *clear_edge*(X, i, p), then the move $(a_i \rightarrow a_i, p)$ is pushed in $Q_{virtual}$. If a_i applies *move*(p) during the execution of Procedure *join*(j), then the move $(a_i \rightarrow a_j, p)$ is pushed in $Q_{virtual}$. Finally, if a searcher applies *move*(p) 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 sequence of extended moves (possibly empty). $Q_{virtual}$ is *strongly equivalent* to Q if, for any $1 \leq j \leq r$, there exists a sequence of moves S_j equivalent to M_j such that $Q_{virtual} = S_1|\cdots|S_r$. $Q_{virtual}$ is *weakly equivalent* to Q if for any $1 \leq j \leq r$, there exists a sequence of moves S_j equivalent to M_j such that $Q_{virtual} = S_1|\cdots|S_r|S_{r+1}$ where $S_{r+1} = (a_i \rightarrow a_{i'}, p_1), (a_i \rightarrow a_{i'}, p_2), \ldots, (a_i \rightarrow a_{i'}, p_\ell)$ where $p_1, \cdots, p_\ell$ is a sequence of port numbers corresponding to a path from a searcher a_i to a searcher $a_{i'}$, in the cleared part of the graph corresponding to the configuration associated to Q in state $M_1|\cdots|M_r$.

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 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 (via a path in the cleared part of the graph, and without recontamination).

The proof of `dist_search` proceeds by considering the algorithm step by step, where a *step* is a moment 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 $(a_i \rightarrow a_i, p)$ 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 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 number of steps. Let us assume that the centralized stack Q and the virtual stack $Q_{virtual}$ are equivalent up to step t. We consider the next step. The difficulty of the proof is in 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 is if the graph is entirely clear. Otherwise there are 3 cases: (1) a searcher can clear a new edge alone, 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 3 other 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 additional cases.

The proof of correctness consists in a careful analysis of each of these 14 cases. Finally, every agent uses at most $O(\log k)$ bits of memory to store the label of another agent in state (HELP, j). The whiteboard size is $O(m \log n)$ by a careful analysis of the protocol.

References

1. S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. SIAM J. Alg. Disc. Meth. 8(2):277-284, 1987.
2. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA), pages 200-209, 2002.
3. L. Barrière, P. Fraigniaud, N. Santoro, and D. M. Thilikos. Searching is not jumping. In 29th Workshop on Graph Theoretic Concepts in Computer Science (WG), Springer-Verlag, LNCS 2880, pages 34–45, 2003.

4. D. Bienstock, Graph searching, path-width, tree-width and related problems (a survey), DIMACS Ser. in Discrete Mathematics and Theoretical Computer Science, 5 (1991), pp. 33–49.
5. D. Bienstock and P. Seymour. Monotonicity in graph searching. Journal of Algorithms 12:239–245, 1991.
6. R. Breisch. An intuitive approach to speleotopology. *Southwestern Cavers* VI(5):72–78, 1967.
7. U. Feige, M. Hajiaghayi, and J. Lee. Improved approximation algorithms for minimum-weight vertex separators. In 37th ACM Symposium on Theory of Computing (STOC), 2005.
8. 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.
9. 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.
10. F. Fomin, P. Fraigniaud and N. Nisse. Nondeterministic Graph Searching: From Pathwidth to Treewidth. In 30th International Symposium on Mathematical Foundations of Computer Science (MFCS), LNCS 3618, pages 364-375, Springer, 2005.
11. F. V. Fomin, D. Kratsch, and I. Todinca. Exact algorithms for treewidth and minimum fill-in. In 31st Int. Colloquium on Automata, Languages and Programming (ICALP 2004), LNCS vol. 3142, Springer, pp. 568–580, 2004.
12. P. Fraigniaud and D. Ilcinkas. Directed Graphs Exploration with Little Memory. Proc. 21st Symposium on Theoretical Aspects of Computer Science (STACS), LNCS 2296, pages 246-257, 2004.
13. P. Fraigniaud and N. Nisse. Connected Treewidth and Connected Graph Searching. In 7th Latin American Theoretical Informatics, LNCS 3887, pages 470-490, 2005.
14. L. Kirousis, C. Papadimitriou. Interval graphs and searching. Discrete Math. 55, pages 181-184, 1985.
15. L. Kirousis, C. Papadimitriou. Searching and Pebbling. Theoretical Computer Science 47, pages 205-218, 1986.
16. A. Lapaugh. Recontamination does not help to search a graph. Journal of the ACM 40(2):224–245, 1993.
17. F. S. Makedon and I. H. Sudborough, On minimizing width in linear layouts, Discrete Appl. Math., 23:243–265, 1989.
18. N. Megiddo, S. Hakimi, M. Garey, D. Johnson and C. Papadimitriou. The complexity of searching a graph. Journal of the ACM 35(1):18–44, 1988.
19. T. Parsons. Pursuit-evasion in a graph. *Theory and Applications of Graphs*, Lecture Notes in Mathematics, Springer-Verlag, pages 426–441, 1976.
20. K. Skodinis Computing optimal linear layout of trees in linear time. In 8th European Symp. on Algorithms (ESA), Springer, LNCS 1879, pages 403-414, 2000. (Also, to appear in SIAM Journal on Computing).
21. 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.

Acknowledgments: The first and fourth authors received additional supports from the project "ALGOL" of the ACI Masses de Données, and from the project "ROM-EO" of the RNRT program. The second and third 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.