Contiguous Search in the Hypercube for Capturing an Intruder *

Paola Flocchini and Miao Jun Huang
School of Information Technology and Engineering
University of Ottawa, Ottawa, Canada
{flocchin,mhuang}@site.uottawa.ca

Flaminia L. Luccio
Dipartimento di Matematica e Informatica
University of Trieste, Trieste, Italy
luccio@dsm.univ.trieste.it

Abstract

In this paper we consider the problem of searching for an intruder in a network. There is a team of collaborative software agents that are deployed to capture a hostile intruder (e.g., a virus). These agents asynchronously move along the network links and the intruder has the capability of escaping arbitrarily fast. We propose two different strategies for the solution of the problem in a widely studied topology: the hypercube network. In the first strategy one of the agents acts as a coordinator making the other agents move in a precise way; this strategy requires $O(n \log n)$ moves, a team of $O(\frac{n}{\log n})$ agents and runs in $O(n \log n)$ time steps. The second strategy is devised for a model where the agents are allowed to “see” the state of their neighbours. In this case, the computation is local, i.e., there is no need of a coordinator and agents can move autonomously. In this setting the solution requires $\frac{n}{2}$ agents, but is much faster ($\log n$ time steps), and requires the same number of moves ($O(n \log n)$).

1. Introduction

1.1. The Problem

We consider a networked environment where nodes are hosts and links represent connections between hosts. A possibly dangerous piece of software (e.g., a virus) is moving in the network from host to host; we will call such an element, the intruder. We assume we have a team of collaborative software agents that can be deployed to protect the network and thus have the goal of localizing and neutralizing the intruder.

We make the following assumptions about the software agents: agents are initially located at the same node (the homebase), communicate through whiteboards located at the nodes; each agent is autonomous in the sense that has some local memory ($O(\log n)$ bits suffice for all our algorithms), can perform local computations, can write and read from a whiteboard located at the node it resides in, can move from node to neighbouring node. The agents are asynchronous in the sense that every action they perform (computing, moving, etc.) takes a finite but otherwise unpredictable amount of time.

The intruder is also a software agent. We assume that, although it could possibly harm the hosts, it cannot damage the other agents. Moreover, to concentrate on the worst case scenario, we assume that given any strategy, the intruder always behaves in the most powerful way. In other words, the intruder moves as if it can “see” the whereabouts of the team of agents, thus avoiding them as much as possible. The intruder is detected when an agent resides on the same node.

Our goal is to devise an efficient strategy for deploying the agents in the network so to localize and neutralize the intruder. Efficiency will be measured in terms of number of agents to be involved, traffic (i.e., number of moves the agents have to perform), and time (or steps)1. A possible strategy is to let the agents move in the network starting from the common homebase so to visit nodes, checking the presence of the intruder, in such a way that no “corridor” (i.e., no way out) is created. In this way the intruder would not able to escape in an already visited part of the network and thus it will be eventually discovered.

The problem of maintaining a secure environment in a distributed system is crucial. Cookies and spywares are both examples of mobile software agents freely moving in a network with different, and usually undesirable, purposes. The solutions for the intruder problem presented in this paper are a first step towards the proposal of an effective solution to “clean” a network. In fact, so to ensure that no undesirable intruders are present in a network, periodic cleaning strategies could be performed by teams of agents. To be realistically employed, these techniques would have to use as few agents as possible and these agents would have to per-

---

1 Since the system is asynchronous, we will measure ideal time, i.e., assuming that it takes one unit of time (i.e., one step) for an agent to traverse a link.
form as few moves as possible so that the cleaning overhead would not be too important compared to the normal load of the network.

1.2. Related Works

A variation of the intruder capturing problem has been extensively studied in the literature under the name of graph search (e.g., see [3, 5, 6, 7, 11]). The graph search problem consists of having a system of tunnels represented by edges of a graph. These tunnels are initially all contaminated, and have to be decontaminated (or cleaned) by a sequence of actions executed by a (minimal) set of searchers. The problem was first introduced in [9, 10], and there are several variations of it. In particular, in the node search problem an edge \((x, y)\) is decontaminated when two searchers are placed, one on \(x\) and the other on \(y\), while in the edge search problem an edge has to be also traversed to be decontaminated.

The aim of both problems is to find a sequence of actions for the searchers that permits to reach a state in which all the edges are simultaneously decontaminated, where an action is one of the following operations: (1) place a searcher on a node, (2) remove a searcher from a node. In the edge search problem a third operation is allowed: (3) move a searcher along an edge. The graph search problem has been extensively studied for many classes of graphs, and finding the optimal number of searchers has been proved to be \(NP\)-complete in most known topologies.

It is easy to see that graph search and intruder detection are equivalent problems. If the intruder is allowed to “hide” only on nodes, we are dealing with a node search problem; if the intruder can “hide” also on the edges, then we have an edge search problem. We say that a node is clean if either it contains an agent, or all its adjacent nodes are clean, then in the node search problem, the goal is to have all the nodes simultaneously clean. In the rest of the paper we will mostly use the terminology of graph search.

In all the graph search variations studied in the literature the results are heavily based on the assumption that a searcher can be initially placed on an arbitrary node and can be arbitrarily moved to any other node. The main difference in our setting is that the agents cannot be removed from the network; they can only move from a node to a neighbouring node. This assumption is obviously motivated by the fact that we are considering software agents that are able to move only on the edges of the network. More precisely, we consider the contiguous, monotone, node search problem studied in [1] where (1) the removal of agents is not allowed, (2) at any time of the search strategy, the set of clean nodes forms a connected subnetwork, and (3) a clean node cannot be recontaminated.

The contiguous assumption considerably changes the nature of the problem and the classical results on node and edge search do not generally apply. The problem of finding the optimal number of agents is still \(NP\)-complete for arbitrary graphs, and has been shown to be solvable with a linear number of moves and with the minimal number of agents only in trees [1]. An investigation on the relationship between this model and the classical model for graph search has been studied in [2].

1.3. Our Results

In this paper, we consider one of the most common topology for interconnection network: the hypercube. In this topology we are concerned with two variations of the contiguous, monotone node search problem.

We first describe a cleaning strategy that requires \(O\left(\frac{n}{\log n}\right)\) agents, \(O(n \log n)\) time steps and \(O(n \log n)\) moves. In section 4 we consider a different model where the agents have a local visibility of the network, i.e., are capable of knowing the status of their neighbours. Under this assumption we devise a strategy that requires \(\frac{n}{2}\) agents, \(\log n\) time steps, and \(O(n \log n)\) moves.

2. Definitions and Terminology

A \(d\)-dimensional hypercube \(H_d\) has \(n = 2^d\) nodes and \(d2^{d-1}\) edges. Each node corresponds to an \(d\)-bit binary string, and two nodes are linked with an edge if and only if their binary strings differ in precisely one bit.

Let \(G = H_d = (V, E)\) be a hypercube; let \(n = |V|\) be the size of \(G\), \(E(x)\) be the edges incident on \(x \in V\). If \((x, y) \in E\) then \(x\) and \(y\) are said to be neighbors. At each node \(x\), there is a distinct label associated to each of its incident edges; the label between node \(x\) and node \(z\) is the position of the bit in which the corresponding binary strings differ. Let \(\lambda_x(x, z)\) denote the label associated at \(x\) to edge \((x, z) \in E(x)\).

Operating in \((G, \lambda)\) is a team of autonomous mobile agents. The agents can move from a node to a neighboring node in \(G\), have computing capabilities and computational storage \((O(\log n)\) bits suffice for all our algorithms), obey the same set of behavioral rules. The agents are asynchronous in the sense that every action they perform (computing, moving, etc.) takes a finite but otherwise unpredictable amount of time. The agents know that the topology they are searching is a hypercube.

Each node has a local storage area called whiteboard \((O(\log n)\) bits of memory suffice for all our algorithms). It is through the whiteboards that agents communicate, in fact they can read from and write on the whiteboards. Access to a whiteboard is gained fairly in mutual exclusion. In particular, the initial information contained in the whiteboard of a node are: its Id (binary string), and the label of the incident ports.
At each point in time, a node $x$ of the hypercube could be in one of the following three states: guarded, if an agent is currently on $x$; clean, if an agent passed by $x$ and all its neighbours are either clean or guarded; contaminated otherwise.

Initially, all agents are in the same node, (the homebase), which is guarded, and all the other nodes are contaminated; a node cleaning strategy consists of a sequence of movements of the agents along the edges of the network.

We say that a cleaning strategy is monotone if once a node is clean, it is never going to be contaminated again. In the following we will be interested only in monotone strategies.

Let us organize the hypercube $H_d$ in $d + 1$ levels and let level $i = 0, 1, \ldots, d$ consist of all the nodes whose binary representation contains $i$ ones. Clearly, all the nodes at level $i$ are connected only to nodes of level $i - 1$ and to those of level $i + 1$. For the ease of discussion let us assume that the degree of the hypercube $d$ is even, minor technical modifications are required for odd degrees.

We consider a special breadth first spanning tree of the hypercube rooted at the source (node $(00,00)$). Let $m(x)$ denote the position of the most significant bit of $x$, then there is an edge in the spanning tree between $x$ and all the nodes in the next level whose bit in which they differ is in a position higher than $m(x)$ (see Figure 1). This spanning tree is also called broadcast tree because it is employed to perform optimal broadcast in the hypercube: a node $x$ receiving a message from dimension $i$ will forward it to all nodes connected by dimension $j > i$.

The resulting spanning tree is also known as heap queue:

**Definition 1** A heap queue $T(d)$ is a rooted tree recursively defined as follows:
- $T(0)$ is a leaf;
- $T(1)$ is a node with one child;
- $T(k)$ is a node with $k$ children of type $T(0), \ldots, T(k - 1)$.

It is very well known that a broadcast spanning tree of a hypercube of size $n$ is a heap-queue $T(\log n)$. In fact, level 0 contains the source $(00 \ldots 0)$ of the tree (which is of type $T(d)$ where $d = \log n$ is the degree of the hypercube).

To easily refer to the neighborhood of a node, the neighbors are grouped into either smaller neighbors or bigger neighbors as defined below. Let $x, y$ be two neighbouring nodes; i.e., $(x, y) \in E(x)$.

**Definition 2** Node $y$ is called a smaller neighbor of node $x$ if $\lambda_x(x, y) \leq m(x)$; $y$ is called a bigger neighbor of $x$ if $\lambda_x(x, y) > m(x)$.

Notice that the bigger neighbors of $x$ are the children of $x$ in the broadcast tree.

Figure 1. The broadcast tree $T(6)$ of the hypercube $H_6$. Normal lines represent edges in $T(6)$, dotted lines (only partially shown) the remaining edges of $H_6$.

3. Node Search

In this section we propose a strategy for the capture of the intruder in the hypercube. This topology has never been investigated neither in the classical model, nor in the contiguous one.

All the agents start the procedure at the same homebase. They are then coordinated by a special agent that, moving back and forth to the homebase, allows the agents to visit all nodes and safely protects the system from the intruder recontamination.

3.1. Basic Properties

We first introduce some basic properties that we will need in the following:

**Property 1** At level 0 there is a unique node of type $T(d)$. At level $l > 0$ there are $(d - k - 1)$ nodes of type $T(k)$ with $0 \leq k \leq d - l$.

**Proof** Nodes of type $T(k)$ start with $k$ 0s followed by a 1. Since they are at level $l$ they must contain $l - 1$ 1s in the last $d - k - 1$ positions. ■
3.2. Cleaning strategy

The cleaning strategy is carried out on the broadcast tree. The agents are all identical; one of them, however, will act as coordinator for the entire cleaning process; we will call this agent the synchronizer. The synchronizer can be locally selected by the agents simply by accessing the whiteboard; the first that gains access will become the synchronizer.

The cleaning strategy is carried out on the broadcast tree. The main idea is to place enough agents in node \( (00\ldots00) \) and then have them move level by level. The agents will move on the broadcast tree leaded by the synchronizer in such a way that, during their movement, the intruder cannot enter the area already cleaned. While cleaning, the agents (except the synchronizer) will move only along the edges of the broadcast tree.

At the beginning, all the agents are available at the root. We call the agents available at the root, set of available agents.

**Algorithm 1** \texttt{CLEAN}.

1. From the root to level 1

   The synchronizer guides for \( d \) times a distinct agent from the root of the tree to each of its \( d \) children \( T(d-1)\ldots T(0) \), each time returning back to the root.

2. From level \( l > 1 \) to level \( l+1 \). [level \( l \) has one agent per node]

   2.1 - Before starting to clean nodes at level \( l+1 \), the synchronizer moves back to the root to collect the agents needed for completing the cleaning of level \( l+1 \) (i.e., \( k-1 \) agents per node of type \( T(k) \) except for the nodes of type \( T(1) \) and \( T(0) \) which do not require any extra agent). The root sends \( k-1 \) additional agents, in no specific order, to each node of type \( T(k) \), \( k > 1 \) at level \( l \).

   2.2 - When \( k \) agents are on a node of type \( T(k) \) at level \( l \), they are sent down in the broadcast tree to the children at level \( l+1 \) guided by the synchronizer. Let \( \eta_1, \ldots, \eta_m (m = \binom{l}{0}) \) be the nodes of level \( l \) lexicographically ordered. The synchronizer sequentially chooses each node at level \( l \) following this lexicographical order and node by node guides on each outgoing edge of the broadcast tree one agent to level \( l+1 \).

   2.3 - When a leaf of level \( l \) is reached by the synchronizer, the agent on it becomes available and goes back to the root. Notice that when the synchronizer reaches the last node of level \( l \), the only active agents are the ones covering level \( l+1 \).

Notice that the whiteboard is used for any communication between the synchronizer and the agents. In particular, during the strategy the whiteboard stores the labels of the incident ports and the current number of agents present at the node; it is accessed by the synchronizer to decide when and where to send the agents to the next level. No more that \( O(\log n) \) bits are required for this local storage.

Figure 2 shows the order in which the nodes are cleaned by the agents leaded by the synchronizer.

3.2.1. Correctness and Analysis

**Correctness.** We now prove that Algorithm \texttt{CLEAN} is correct; i.e., that all nodes will be cleaned and that once a node has been cleaned, it will never be recontaminated.

Let \( x \) be a node of level \( l \), \( N(x) \) denote the neighbours of \( x \) at level \( l+1 \) and \( NT(x) \) denote the children of \( x \) at level \( l+1 \) in the broadcast tree. Notice that \( N(x) \) includes \( NT(x) \) and possibly some other neighbours at level \( l+1 \).

**Lemma 1** If \( z \in N(y) - NT(y) \) then \( z \in NT(x) \) for some \( x \) such that \( \hat{x} < \hat{y} \).

**Proof** Let \( h \) be the position of the most significant bit of \( y \). Since \( z \notin NT(y) \) we know that it does not contain a 1 bit in position higher than \( h \). However, \( z \in N(y) - NT(y) \), i.e., it is a neighbour of \( y \) (not in the broadcast tree) at level
The children of node $y$ at level $i + 1$, each of $NT(y)$ is guarded by an agent and all neighbors of $y$ are either clean or guarded by an agent. If $y$ is a leaf, $NT(y)$ is empty. When the synchronizer reaches it, all neighbors of $y$ at level $i + 1$ are guarded and all neighbors at level $i − 1$ are clean.

**Theorem 1** The cleaning process decontaminates all nodes. During the execution clean nodes can not be recontaminated.

**Proof** First of all, by the cleaning strategy, the cleaning is performed level by level. So when it reaches level $d$, all nodes have been cleaned. The fact that a clean node will not be recontaminated directly follows from lemma 2.

**Complexity.** We first calculate the number of agents needed to perform the cleaning of the hypercube with Algorithm **CLEAN**.

**Theorem 2** In a $d$-dimensional hypercube, algorithm **CLEAN** employs $O\left(\frac{n}{\log n}\right)$ agents to clean the network.

We will prove the theorem by calculating the number of agents needed to clean from level $l$ to level $l + 1$, and by showing that the maximum number of agents is needed when cleaning from level $\frac{d}{2} − 1$ to $\frac{d}{2}$ and from level $\frac{d}{2}$ to level $\frac{d}{2} + 1$, where the corresponding number will be shown to be $(\frac{d}{2})^d + (\frac{d}{2} − 1)^d + 1 = O\left(\frac{n}{\log n}\right)$.

We first calculate the number of agents that are taken from the set of available agents before performing the cleaning from level $l$ to level $l + 1$.

**Lemma 3** Before cleaning from level $l > 0$ to level $l + 1$, $(\frac{d}{l+1})^d − (\frac{d}{l})^d + (\frac{d}{l−1})^d$ extra agents are requested from the root by the synchronizer.

**Proof** In step 2.1, $k − 1$ extra agents for a node of type $T(k)$ are sent from the root. By property 1, there are $(\frac{d}{l−1})^d$ nodes of type $T(k)$ at level $l > 0$. So totally $\sum_{k=2}^{d-l}(k−1)(\frac{d}{l−1})^d$ extra agents are sent from the root to level $l$ while cleaning from level $l$ to level $l + 1$.

Note that first choosing $i = k − 1$ and then $L = l − 1$ we have:

$$\sum_{k=2}^{d-l}(k−1)(\frac{d}{l−1})^d = \sum_{i=1}^{d−1−l} (d−(i+1)−1) = \sum_{i=1}^{d−L−2} \binom{i}{1}(d−i−2)^L.$$

Observe now that given $a, b \in \mathbb{N}$ we have $\binom{a}{b} = 0$ for $a < b$. Therefore:

$$\sum_{i=1}^{d−2} \binom{i}{1}(d−i−2)^L = \sum_{i=0}^{d−2} \binom{i}{1}(d−i−2)^L.$$
Referring to (4) we have that: \( \sum_{i=0}^{d-2} \binom{i}{d-1} \binom{d-2-i}{L} = \binom{d-1}{L+2} \), thus,

\[
\sum_{i=0}^{d-2} \binom{i}{d-i-2} \binom{d-1}{L+1} = \left( \frac{d}{l+1} \right) - \left( \frac{d-1}{l} \right) = \left( \frac{d}{l+1} \right) - \left( \frac{d-1}{l} \right).
\]

We now show that:

**Lemma 4** \( \left( \frac{d}{2} \right) + \left( \frac{d-1}{2} \right) + 1 \) agents are enough to perform the cleaning from level \( l \) to level \( l + 1 \).

**Proof** By induction.

There are enough agents to clean level 1 because only \( d \) agents are needed and \( d < \left( \frac{d-1}{2} \right) + \left( \frac{d-1}{2} \right) + 1 \) for \( d > 1 \).

Let us assume that there are enough agents to clean level \( l \), \( l \geq 1 \), using our strategy. At this point we have \( \left( \frac{d}{2} \right) + 1 \) active agents (where the synchronizer is counted too), every node of level \( l \) is guarded by one agent; all the other agents are available. We now show that there are enough agents to clean level \( l + 1 \) using our strategy.

By lemma 3, before cleaning from level \( l \) to level \( l + 1 \), the synchronizer collects extra \( \left( \frac{d}{l+1} \right) - \left( \frac{d-1}{l} \right) \) agents. This number has been calculated so to give exactly \( k \) agents for each tree of type \( T(k) \), which is just enough to proceed with the cleaning of level \( l + 1 \). In fact, by induction hypothesis, each node of level \( l \) is already guarded by one agent before the extra agents come. So in total, \( \left( \frac{d}{2} \right) + 1 \) + \( \left( \frac{d}{l+1} \right) - \left( \frac{d-1}{l} \right) \) = \( \left( \frac{d}{l+1} \right) + \left( \frac{d-1}{l} \right) + 1 \) agents are active at level \( l \). By our strategy, the \( \left( \frac{d}{l+1} \right) \) agents on the leaves do not participate in the cleaning of level \( l + 1 \), but the other \( \left( \frac{d}{l+1} \right) \) are just enough to move to level \( l + 1 \) on the broadcast tree.

In addition, it is well known that:

\[
\max_{1 \leq i \leq d-1} \left( \frac{d}{l+1} \right) + \left( \frac{d-1}{l} \right) = \left( \frac{d}{l+1} \right) + \left( \frac{d-1}{l} \right) = \left( \frac{d}{2} \right) + \left( \frac{d-1}{2} \right) \text{ for } l = \frac{d}{2} \text{ or } l = \frac{d}{2} - 1, \text{ respectively}.
\]

Therefore, \( \left( \frac{d}{2} \right) + \left( \frac{d-1}{2} \right) + 1 \) is the maximum number of agents required by the procedure and corresponds to the cleaning of the two central levels. Finally observe that \( \left( \frac{d}{2} \right) + \left( \frac{d-1}{2} \right) + 1 \) is known to be \( O\left( \frac{n}{\log n} \right) \) (4). "

We now calculate the total number of moves needed for the entire process.

**Theorem 3** The total number of moves performed by the agents is \( O(n \log n) \).

**Proof** To compute the global number of moves we have to take into account the ones performed by the agents and those performed by the synchronizer.

The number of moves performed by the agents: It takes \( 2l \) moves for an agent to arrive at a leaf of level \( l \) from the root and go back to the root. By property 2, there are \( \left( \frac{d}{l+1} \right) \) leaves at level \( l \). So totally, there are \( \sum_{l=0}^{d-1} 2\left( \frac{d}{l+1} \right) \) moves performed by the agents. To compute this quantity first note that \( \left( \frac{d}{l} \right) \) is the number of nodes at level \( l \), thus summing up over all levels we obtain, \( \sum_{l=0}^{d} \left( \frac{d}{l} \right) = 2d = n \), which is a known result [4]. It follows that \( \sum_{l=1}^{d} \left( \frac{d}{l-1} \right) = \sum_{l=0}^{d-1} \left( \frac{d}{l} \right) = 2d-1 = \frac{n}{2} \).

We have now to compute:

\[
\sum_{l=1}^{d} \left( \frac{d}{l-1} \right) = 1\left( \frac{d}{0} \right) + ... + \left( \frac{d}{2} - 1 \right)\left( \frac{d}{2} - 2 \right) + \frac{d}{2} \left( \frac{d-1}{4} \right) + \left( \frac{d+1}{2} \right)\left( \frac{d}{2} - 1 \right) + \left( \frac{d+1}{2} \right) + \left( \frac{d+2}{4} \right)\left( \frac{d-1}{4} \right) + ... + d\left( \frac{d-1}{1} \right).
\]

We already know that: \( \sum_{l=0}^{d-1} \left( \frac{d}{l} \right) = 2d-1 = \frac{n}{2} \); we also know that: \( \sum_{l=0}^{d-1} \left( \frac{d}{l} \right) = 2d-1 = \frac{n}{2} \). Thus, \( d + 1 \) \( \sum_{l=0}^{d-1} \left( \frac{d}{l} \right) = (d + 1)2d-2 = \frac{n}{2}(\log n + 1) \).

Thus, the total number of moves performed by the agents is:

\[
\sum_{l=1}^{d} 2d\left( \frac{d-1}{l-1} \right) = \frac{n}{2}(\log n + 1) = O(n \log n).
\]

The number of moves performed by the synchronizer:

1. Go to the root to get more agents. Totally, there are \( \sum_{l=1}^{d-3} l = \frac{(d-3)(d-2)}{2} = O(\log^2 n) \) moves.

2. Go to the first node of each level. Totally, there are \( \sum_{l=1}^{d-2} l = \frac{(d-2)(d-1)}{2} = O(\log^2 n) \) moves.

3. Navigate within each level to get to the next node. Recalling that the procedure is run on an hypercube structure we obtain that at level \( l \), the synchronizer needs to navigate at most \( 2l \) edges if \( l \leq \frac{d}{2} \), otherwise \( 2d - 2l \) edges to reach the next node at the same level. So totally the number of moves is at most \( \sum_{l=1}^{d/2} 2l\left( \frac{d}{l+1} \right) + \sum_{l=1}^{d/2} (2d-2l)\left( \frac{d}{l+1} \right) = 4\sum_{l=1}^{d/2} l\left( \frac{d}{l+1} \right) + d\left( \frac{d}{2} \right) + 2d = O(n \log n) \).

4. Go down with each agent to clean a node at the next level in the broadcast tree and then come back. Each edge of the broadcast tree is traveled twice by the synchronizer. So totally there are \( 2(2d-1) = 2(n-1) \) moves.

Totally, the number of moves performed by the cleaning process is \( O(n \log n) \).
We now consider the ideal time complexity of the cleaning strategy. We remind that we are in an asynchronous environment, so we consider the ideal time complexity (i.e., by assuming that it takes one unit of time for an agent to traverse an edge).

**Theorem 4** The cleaning strategy takes $O(n \log n)$ time units.

**Proof** The cleaning process is carried out sequentially by the synchronizer. The time required is then equal to the number of moves of the synchronizer.

4. Node Search with Visibility

In this section we consider the node search problem in the hypercube in a new different model. We assume that an agent located at a node can “see” whether its neighboring nodes are clean or guarded or contaminated. This capability could be easily achieved if the agents have also communication power and send a message (e.g., a single bit) to their neighbouring nodes after cleaning a node or guarding a node. The interesting aspect of this model is that this extra capability enables agents to correctly act without the need of being coordinated by the synchronizer.

In the following we will then make the following assumption on the agent visibility:

An agent located at node $x$ can see the state of the neighbours $N(x)$.

Thus, as we will see later in this section, the visibility assumption allows the agents to make their own decision regarding the action to take solely on the basis of their local knowledge.

4.1. Basic Properties

We now introduce some basic properties that we will need in the following. First, we group the nodes of the hypercube based on the position of the most significant bit in their binary string.

Formally, let $C_i$ be the set of nodes whose most significant bit is in the $i$-th position (see Figure 3).

**Property 5** Exactly one node is in $C_0$; $2^{i-1}$ nodes are in $C_i$ for $0 < i \leq d$.

**Proof** By definition, only node (00..00) is in $C_0$. The nodes in $C_i$ have the most significant bit in the $i$-th position. There are $2^{i-1}$ nodes with most significant bit in the $i$-th position. Hence $C_i$ has $2^{i-1}$ nodes.

**Property 6** All the leaves of the broadcast tree are in $C_d$.

Let node $x$ be any node in $C_i$. Since $C_0$ has only one node (00..00), which has no smaller neighbors, we only consider $x$ in $C_i$ where $i > 0$ for the following property.

**Property 7** One smaller neighbor of node $x$ is in $C_j$ (where $j < i$), the other smaller neighbors, if any, are in $C_i$ and all its bigger neighbors, if any, are in $C_k$ (where $k > i$).

**Proof** By definition, the most significant bit of node $x$ is in the $i$-th position. The labels of its neighbors differ only in one bit. If a neighbor $y$ is different from $x$ in the $l$-th position for $l < i$, then $y$ is a smaller neighbor of $x$ and $y$ has a 1 bit in the $i$-th position. By definition, $y$ is in $C_i$ too. If the neighbor $y$ is different from $x$ in the $i$-th position, then the most significant bit of $y$ must be in position $j$, $j < i$. So $y$ is in $C'_j$. By definition, $y$ is a smaller neighbor of $x$ too. Since there is only one neighbor different from $x$ in the $i$-th position, we have one smaller neighbors of node $x$ is in $C_j$, $0 \leq j < i$ and the other smaller neighbors are in $C_i$. If the neighbor $y$ is different from $x$ in the $k$-th position where $k > i$, then the $k$-th position of $y$ must be 1 because the most significant bit of $x$ is in the $i$-th position. By definition, $y$ is the bigger neighbor of $x$ and it is in $C_k$ where $k > i$.

Let node $x$ be any node in $C_i$. Since node (00..00) in $C_0$ has no smaller neighbors and node (00..001) in $C_1$ has one smaller neighbor in $C_0$, but no smaller neighbor in $C_1$, we only consider node $x$ in $C_i$ where $i > 1$ for the following property.

**Property 8** There must exist at least one smaller neighbor $y$ of node $x$ such that $y$ is in $C_i$ and a smaller neighbor $z$ of $y$ is in $C_{i-1}$.

**Proof** By property 7, all smaller neighbors (except one) of node $x$ are in $C_i$. Each of these neighbors is one bit different from $x$ in the $j$-th position, where $j < i$. Depending on the $(i-1)$-th position of $x$, there are two cases.
Case 1: If the \((i - 1)\)-th position of \(x\) is a 0 bit, let \(y\) be the smaller neighbor which is different from \(x\) in the \((i - 1)\)-th position. Node \(y\) is in \(C_i\) and has a 1 bit in the \((i - 1)\)-th position. Node \(y\) has a smaller neighbor \(z\) such that \(z\) is different from \(y\) in the \(i\)-th position. Node \(z\) has a 0 bit in the \(i\)-th position and a 1 bit in the \((i - 1)\)-th position, which is the most significant bit of \(z\). So node \(z\) is in \(C_{i-1}\).

Case 2: If the \((i - 1)\)-th position of \(x\) is a 1 bit, let \(y\) be any smaller neighbor which is different from \(x\) in the \(j\)-th position, \(j < i - 1\). Node \(y\) is in \(C_i\) and has a 1 bit in the \(i\)-th and in the \((i - 1)\)-th position. Let \(z\) be a smaller neighbor of \(y\) which is different from \(y\) in the \(i\)-th position. \(z\) has a 0 bit in the \(i\)-th position and a 1 bit in the \((i - 1)\)-th position, which is the most significant bit of \(z\). So node \(z\) is in \(C_{i-1}\).

Hence, in any case, there exists a smaller neighbour \(z\) of \(y\) such that \(z\) is a node of \(C_{i-1}\).

### 4.2. Cleaning strategy

All the agents are initially located at the source; they are identical and autonomous, and they all follow the same local rule.

The agents are still moving on the broadcast tree, but they do not have to follow the order imposed by the coordinator like in the previous section. In fact, the agents on node \(x\) can proceed to clean the children of \(x\) in the broadcast tree when they “see” that the other neighbours of \(x\) are either clean or guarded. In other words, the agents on node \(x\) can proceed to clean the bigger neighbours of \(x\) when they “see” that the smaller neighbours of \(x\) are either clean or guarded.

**Algorithm 2** Clean with Visibility

**Rule for the agents on node** \(x\) **of type** \(T(k) (0 \leq k \leq d)\):

- If the number of agents on \(x\) is less than \(2^{k-1}\), then the agents wait on \(x\).
- When \(2^{k-1}\) agents are on \(x\): when all the smaller neighbours of \(x\) are clean or guarded, one agent moves to the bigger neighbour of type \(T(0)\); \(2^{i-1}\) agents move to each of the bigger neighbours of type \(T(i)\) for \(0 < i < k\); if there are no bigger neighbours, terminate.

Notice that the whiteboard is used for storing the status of the node (contaminated, clean, guarded), the current number of agents present at the node. During the strategy the agents can access the local whiteboard and the whiteboards of the neighbours. Which agent go to which node is also determined by accessing the whiteboard. Also in this case, no more that \(O(\log n)\) bits are required for the whiteboard.

Figure 4 shows the order in which the nodes get cleaned with our strategy. As opposed to the strategy of the previous section, nodes are not cleaned sequentially; several nodes, in fact, could be cleaned independently.

### 4.3. Correctness and Complexity

**Correctness.**

**Theorem 5** The total number of agents needed to clean the \(d\)-dimensional hypercube using Algorithm Clean with Visibility is \(\frac{n}{2}\).

**Proof** By definition Algorithm Clean with Visibility is sending from level 0 to level 1 one agent for \(T(0)\) and \(2^{i-1}\) agents for each \(T(i)\) for a total of \(1 + \sum_{i=1}^{d-1} 2^{i-1} = 1 + \sum_{i=0}^{d-2} 2^i = 2^{d-1} - 1 = \frac{n}{2}\) agents. Moreover, a node of type \(T(k)\) receives \(2^{k-1}\) agents and \(2^{k-1}\) is exactly the number of agents needed to continue the cleaning strategy, in fact, \(2^{k-1} = 1 + \sum_{i=1}^{k-1} 2^{i-1}\). Thus, with \(\frac{n}{2}\) agents the strategy can be completed.

We now prove that algorithm 2 is correct; i.e., that the network is clean and once a node has been cleaned, it will never be recontaminated.

**Lemma 5** In Algorithm Clean with Visibility, when agents leave a node in \(C_i\) (leaving it unguarded) all its smaller neighbours are either clean or guarded.

**Proof** Let us consider a node \(x\) in \(C_i\). By the cleaning strategy, when an agent arrives at node \(x\), it cleans the node. By theorem 5 every node will have eventually enough agents...
to continue the cleaning process, by definition of the cleaning rule, the agents on $x$ move to the bigger neighbours only when all other neighbours are clean or guarded.

**Theorem 6** During the cleaning process the agents clean all nodes and a clean node will not be recontaminated.

**Proof** First of all, by the cleaning strategy, the edges and nodes traversed by the agents form the broadcast tree. All the nodes are visited by an agent. The fact that a clean node will not be recontaminated directly follows from lemma 5.

**Complexity.** We now consider the time complexity of the cleaning strategy.

**Theorem 7** Cleaning the entire network takes $O(\log n)$ time units.

**Proof** We will prove it by showing that at time $i$, all nodes in $C_i$ are clean; only the agents in $C_i$ can move to clean the bigger neighbours, which are in $C_j$ for $j > i$. We prove it by induction.

Base case: at time $i = 0$, all the agents are placed on node (00...00) (the source). First notice that, since there is no agent on any other node at time 0, only the agents in $C_0$ might move at this time. By the cleaning strategy, the agents clean the source, then they move to clean the $d$ bigger neighbours, which are in $C_j$ for $0 < j \leq d$. Node (00...00) becomes clean at time 0; obviously, no recontamination can occur to it. The claim then holds for $i = 0$.

Assume the claim is true up to time $i, i \geq 0$. We show that it holds at time $i + 1$.

By induction hypothesis and theorem 6 all the nodes in $C_k$ are clean for any $0 \leq k \leq i$. The agents ever on them have left. Let node $x$ be an arbitrary node in $C_{i+1}$. By lemma 7, exactly one smaller neighbour of $x$ is in $C_k$ for $k \leq i$. By induction hypothesis, at time $k$, the agents arrive at $x$ and clean it upon arrival. So at time $i + 1$, every node in $C_{i+1}$ is guarded by at least an agent; and, thus, all smaller neighbours of any node in $C_{i+1}$ are clean or guarded. So at time $i + 1$ every agent in $C_{i+1}$ execute the algorithm; they go to clean the bigger neighbours which, by lemma 7, are in $C_j$ with $j > i + 1$. Since the nodes in $C_{i+1}$ are already cleaned by their guarding agents upon arrival, together with the fact that a clean node will not be recontaminated by theorem 6, we know that at time $i + 1$ the nodes in $C_{i+1}$ become clean after the agents on them move.

Notice that, by our cleaning algorithm, the other agents in $C_j$ for $i + 1 < j \leq d$ cannot move because one or more of their smaller neighbours are not guarded. Let us consider any node $y$ in $C_j$ on which there are agents. By lemma 8, there exists one smaller neighbour $z$ of $y$ which is in $C_{j-1}$ too and a smaller neighbour $w$ of $z$ which is in $C_{j-1}$. We know that the agents on $z$ come from its smaller neighbour $w$ which is in $C_{j-1}$. If $j - 1 = i + 1$, in other words, $w$ is in $C_{i+1}$, the agents on $w$ move at time $i + 1$. So at time $i + 1$ no agent is on $z$ yet. If $j > i + 1$, even if there are agents on $w$, by induction hypothesis, they have not moved before time $i + 1$. Hence, in any case, $z$ is not guarded at time $i + 1$ and the agents on $y$ cannot move because at least one of its smaller neighbors is not guarded.

We now calculate the total number of moves performed by the agents.

**Theorem 8** The number of moves performed by the agents for the entire cleaning is $O(n \log n)$.

**Proof** All the agents start from the source and each terminates on a leaf. There are $l_{i}^{d-1}$ leaves at level $l > 0$, thus the total number of moves is: $\sum_{i=0}^{d} l_{i}^{d-1} = O(n \log n)$ (the calculation is similar to the one of the proof of theorem 3).

5. Conclusion

In this paper we have considered the problem of searching for an intruder in a hypercube network. We have presented two different strategies that run in two distinct models. The first strategy is based on the coordination of a synchronizer and the solution requires $O(n \log n)$ agents, $O(n \log n)$ steps, and $O(n \log n)$ moves. The second strategy works in a more powerful model where the agents are allowed to “see” the state of their neighbours. On the other hand computation is local, i.e., there is no need of a coordinator. In this setting the solution requires $\frac{n}{\log n}$ agents, but is much faster ($O(n \log n)$ steps), and requires the number of moves $O(n \log n)$.

**Observations on Cloning.** Our second strategy (for the visibility model) would be particularly suitable if the agents have cloning capabilities, i.e., if an agent can create a copy of itself. In this case only one agent would be initially placed at the homebase and agents would be cloned when needed. With this cloning power, the second strategy still requires $\frac{n}{\log n}$ agents and $O(n \log n)$ steps, but the number of moves performed by the agents is reduced to $n - 1$. Notice however, that cloning would not introduce any advantage in our first strategy, instead it would increase the number of agents employed to $\frac{n}{\log n}$.

**Observations on Synchronicity.** Let us briefly consider what happens if the agents move synchronously (instead of asynchronously) and they start simultaneously. In this setting synchronicity could be exploited by using a strategy very similar to the one of Algorithm CLEAN WITH VISIBILITY (and with the same complexity), but without needing the visibility assumption. Instead of waiting for all smaller neighbors to become clean or guarded, the agents on a node wait for the appropriate time to move and clean the bigger neighbors. Recall that $m(x)$ denote the position of the most significant bit of $x$; in the synchronous model, the
agents on $x$ can move when time $t = m(x)$. In this strategy, when $t = m(x)$, the agents on $x$ implicitly know that all the smaller neighbor(s) of $x$ are clean or guarded. Hence, when they move to the bigger neighbors according to the rule: one agent is sent to the bigger neighbor of type $T(0)$, and $2^{i-1}$ agents are sent to the bigger neighbor of type $T(i)$, no re-contamination can occur.

More details about the cloning and the synchronous variations can be found in [8].

Finally, an interesting open problem is to determine whether our strategy for the first model is optimal in terms of number of agents; i.e., if the lower bound on the number of agents is $\Omega(\frac{n}{\log n})$.

References