INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Intruder Capture by Mobile Agents in Mesh Topologies

By
Lisa Xiuli Song

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

April 12, 2005

© 2005, Lisa Xiuli Song

Library and
Archives Canada

Bibliothèque et
Archives Canada

0-494-06833-7

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

In a mobile agents based network environment, *intruders* are defined as harmful mobile agents penetrating into the environment. Many graph search algorithms can be related to the *intruder capture problem*, and some have been developed to capture intruders by using *mobile agents*, such as capturing intruders in a tree [5] and etc. The efficiency of an *intruder capture* algorithm is the *size* of the team deployed to capture the intruders. In this paper, we propose algorithms for capturing intruders in four types of *Mesh* topologies, namely *Mesh*, *Hexagonal Mesh*, *Octagonal Mesh*, and *Toroidal Mesh*. First we show that in any $m \times n$ -Mesh ($m \leq n$), at least m agents are needed to solve the problem. We consider two models for each algorithm, one in which an agent has only local knowledge about the node it resides, and the other in which an agent has visibility to see their neighboring nodes. Then for each type of mesh, we design an algorithm in both models, prove its correctness and analyze its complexity in terms of number of agents, number of moves and time required to capture the intruder.

Acknowledgement

I extend my sincere gratitude and appreciation to those who gave me the possibility to complete this thesis. Especially I would like to give my profound thanks to my supervisor Prof. Dr. Nicola Santoro from Carleton University and Prof. Dr. Paola Flocchini from University of Ottawa, whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis. Special thanks to my family, for their patient love and continuous support enabled me to complete this work.

Contents

1	Introduction	1
1.1	Intruder Capture Problem	1
1.2	Related Work	2
1.2.1	Graph Search	2
1.2.2	Contiguous Node/Edge-Search	5
1.3	Terminology	7
1.3.1	The Network	7
1.3.2	Agent Based System	9
1.3.3	Asynchronous Model	12
1.3.4	Searching Operations	13
1.3.5	Complexity Measures	15
1.4	The Results	16
2	Node/Edge-Search in a $m \times n$-Mesh	19
2.1	Lower bound on the number of searchers in a $m \times n$ -Mesh	20
2.2	Searching Strategies - With a Synchronizer	22

2.2.1	Node-Search in a $m \times n$ -Mesh	23
2.2.2	Edge-Search in a $m \times n$ -Mesh	29
2.3	Searching Strategies - Agents with Visibility	31
2.3.1	Node-Search in a $m \times n$ -Mesh	32
2.3.2	Edge-Search in a $m \times n$ -Mesh	38
2.4	Summary of Searching Results on Different Models in a $m \times n$ -Mesh .	45
3	Node/Edge-Search in a $m \times n$-Hexagonal Mesh	48
3.1	Searching Strategies - With a Synchronizer	49
3.1.1	Node-Search in a $m \times n$ -Hexagonal Mesh	49
3.1.2	Edge-Search in a $m \times n$ -Hexagonal Mesh	55
3.2	Searching Strategy - Agents with Visibility	57
3.2.1	Node-Search in a $m \times n$ -Hexagonal Mesh	58
3.2.2	Edge-Search in a $m \times n$ -Hexagonal Mesh	62
3.3	Summary of Searching Results on Different Models in a $m \times n$ -Hexagonal Mesh	69
4	Node/Edge-Search in a $m \times n$-Octagonal Mesh	71
4.1	Searching Strategies - With a Synchronizer	72
4.1.1	Node-Search in a $m \times n$ -Octagonal Mesh	72
4.1.2	Edge-Search in a $m \times n$ -Octagonal Mesh	78
4.2	Searching Strategies - Agents with Visibility	84
4.2.1	Node-Search in a $m \times n$ -Octagonal Mesh	84

4.2.2	Edge-Search in a $m \times n$ -Octagonal Mesh	92
4.3	Summary of Searching Results on Different Models in a $m \times n$ -Octagonal Mesh	99
5	Node/Edge-Search in a $m \times n$-Toroidal Mesh	101
5.1	Searching Strategies - With a Synchronizer	101
5.1.1	Node-Search in a $m \times n$ -Toroidal Mesh	102
5.1.2	Edge-Search in a $m \times n$ -Toroidal Mesh	105
5.2	Searching Strategies - Agents with Visibility	109
5.2.1	Node-Search in a $m \times n$ -Toroidal Mesh	109
5.2.2	Edge-Search in a $m \times n$ -Toroidal Mesh	113
5.3	Summary of Searching Results on Different Models in a $m \times n$ -Toroidal Mesh	118
6	Conclusions	120
6.1	Results	120
6.2	Future Work	125

List of Tables

2.1	Node/Edge-Search in a $m \times n$ -Mesh - With a Synchronizer	25
2.2	Node-Search in a $m \times n$ -Mesh - Agents with Visibility	34
2.3	Edge-Search in a $m \times n$ -Mesh - Agents with Visibility	41
3.1	Node/Edge-Search in a $m \times n$ -Hexagonal Mesh - With a Synchronizer	51
3.2	Node-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility . .	59
3.3	Edge-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility . .	65
4.1	Node-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer . . .	74
4.2	Edge-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer . . .	80
4.3	Node-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility . .	88
4.4	Edge-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility . . .	95
5.1	Node-Search in a $m \times n$ -Toroidal Mesh - With a Synchronizer	103
5.2	Edge-Search in a $m \times n$ -Toroidal Mesh - With a Synchronizer	106
5.3	Node-Search in a $m \times n$ -Toroidal Mesh - Agents with Visibility	111
5.4	Edge-Search in a $m \times n$ -Toroidal Mesh - Agents with Visibility	115

List of Figures

1.1	An example of four types of Mesh topologies	10
1.2	Compass-Labelling in Mesh.	10
2.1	Proof of Theorem 1	21
2.2	Node-Search in a 4×5 -Mesh - With a Synchronizer	23
2.3	Node/Edge-Search in a $m \times n$ -Mesh - With a Synchronizer	24
2.4	Node-Search in a $m \times n$ -Mesh - Agent with Visibility	32
2.5	Edge-Search in a $m \times n$ -Mesh - Agent with Visibility	39
3.1	Node/Edge-Search in a $m \times n$ -Hexagonal Mesh - With a Synchronizer	49
3.2	Node-Search in a $m \times n$ -Hexagonal Mesh - Agent with Visibility	58
3.3	Edge-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility	63
4.1	Node-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer	72
4.2	Edge-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer	78
4.3	Node-Search in a $m \times n$ -Octagonal Mesh - Agent with Visibility	85
4.4	Edge-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility	92

Chapter 1

Introduction

1.1 Intruder Capture Problem

Mobile agents have been widely used in distributed databases, network management and e-commerce systems. It is an attractive paradigm for implementing communication intensive applications. A mobile agent is like a process or remote procedure call except that, while running, it can specify and change the processor on which it is executed. However, the mobility of mobile agents may cause some problems to the network environment, such as intruder problems. *Intruders* are defined as the possible harmful mobile agents penetrated into a network environment. If intruders are detected in a network, we must clean the contaminated network. Many graph search algorithms can be related to the intruder capturing problems, and some have been developed to capture intruders by using mobile agents, such as capturing intruders in a tree [5].

In an *intruder capture* problem, intruders move along the network links contaminating the nodes (and links) while they traverse, and capturers also move along the network to capture the intruders at network nodes or links. However, the intruders could be arbitrarily fast, and aware of the positions of all the capturers so that the intruders may escape from the capturers. To solve this problem, a capture strategy is to clean the contaminated network and protect the cleaned areas so that no nodes or edges would allow the intruders to escape. The capture result is to reach a state in which all the nodes (and edges) are simultaneously cleaned, and ideally the total number of guards used at any step are minimized. A strategy that achieves this result is called optimal, and the optimal number of guards is the intruder capturer number of the graph. The benchmark of the efficiency of an intruder capture algorithm is the *size* of the team to capture the intruders.

1.2 Related Work

1.2.1 Graph Search

The intruder capture problem is an instance of some well known graph-search problems whose many variants have been extensively studied in the literature and well summarized in [5, 6]. The *graph search problem* consists of having a system of tunnels represented by edges of a graph. These tunnels are initially all *contaminated* by a gas, and have to be simultaneously decontaminated by a sequence of clean operations (or moves) executed by a (minimal) set of *searchers*. The problem was first introduced

by Breish [8] and Parson [20, 21].

There are various versions of the graph search problems (node-search or edge-search) depending on whether the nodes or the edges are contaminated in the graph.

In the *node-search problem* an action consists of the following two operations:

- *place* a searcher on a node.
- *remove* a searcher from a node.

With the two operations, a node is cleaned by placing a searcher on the node. An edge (x, y) is decontaminated when two searchers are placed on it, one on node x and the other on node y . Or if one end of the edge is already cleaned, then the edge is cleaned by placing one searcher on the other contaminated end of the edge.

In the *edge-search problem*, an extra operation is allowed:

- *move* a searcher along an edge.

By introducing the extra operation in *edge-search*, a contaminated edge (x, y) can be cleaned by:

1. Placing two searchers on x and moving one of them to y along the edge (x, y) , which means to traverse the edge.
2. If all the edges incident to x except the link (x, y) are decontaminated, and x contains a searcher, the searcher can be moved along (x, y) .

The study of both types of searching problems is to find a strategy that permits the graph to reach a state in which all nodes or nodes and edges are simultaneously

decontaminated, and ideally the total number of searchers used at any step are minimized. A strategy that achieves this result is called optimal, and the optimal number of searchers is the number of the intruder capturers used in the graph. The efficiency of an intruder capture algorithm is the *size* of the searching team to capture the intruders. This number is called the *node-search number* ($ns(G)$) or the *edge-search number* ($es(G)$) respectively in the node or edge search problem.

Both types of searching problems have been extensively studied, and for many classes of graphs it has been proved to be *NP*-complete. The node search problem is *NP*-complete for bipartite graphs [13], cobipartite graphs [1], bipartite distance hereditary graphs [14], planar graph with maximum degree three [15], and starlike graphs [11]. There exist many linear-time algorithms in trees [10, 18, 23], one in k -trees for fixed k [2], in cographs [4], and an $O(pm)$ -time algorithm in permutation graphs (where p is the pathwidth) [3], and finally, for any fixed k , an $O(mn^k)$ -time and $O(m)$ space algorithm on the k -starlike graph of n vertices and m edges [22].

On the other hand, the edge search problem is *NP*-complete for general graphs [17] and planar graphs with maximum degree three [19]. It can be solved in linear time in trees [5], and given n the number of vertices and m the number of edges, in $O(mn^2)$ -time in split graphs, in $O(m+n)$ -time in interval graphs, in $O(mn^k)$ -time in k -starlike graphs, for any fixed $k \geq 2$ [22].

The existing node/edge-search problems have limitations. In both types of search problems, searchers can be arbitrarily placed and removed from any node in the searching graph. This means that searchers are allowed to “*jump*” while they perform

the searching tasks. However, searchers may be contaminated when they “*jump*” in some systems. This is because the searchers will traverse in an unknown (or may be dangerous) environment when they “*jump*” from one node to another node in the graph. Therefore, it will be impossible to perform the network cleaning task by using those damaged searchers. Hence, a variation of the node/edge-search problem without “*jump*” is described in the following section.

1.2.2 Contiguous Node/Edge-Search

In some environment, a realistic solution of an intruder capture problem requires the capturers to perform *contiguous* searches in a given network instead of making any “*jump*” moves. Thus, the variation of the node/edge-search problem is the *contiguous search problem* [5, 6]. This problem is still *NP*-complete for general graphs, but can be solved in linear time in trees [5].

The *contiguous search problem* defines three additional stricter conditions or properties as follows:

1. *Internal Search Property.*

Searchers may not be removed from the present location and eventually placed in a different part of the graph but only on a neighboring node by traversing an edge. This condition can be well represented by the realistic assumption that the tunnels are in a cave. Searchers may only move in a contiguous way, which means the entrance to the cave is unique and searchers may not “*jump*” from a tunnel to another.

2. *Connected Search Property.*

All cleaned links form a connected subgraph of the original graph. In other words searchers have to move in a team and we do not allow them to search in the cave starting from distinct and separate entrances (i.e., the entrance is unique). This implies that a contiguous search cannot be applied to non-connected graphs, therefore through out this paper we will only consider connected graphs.

3. *Monotone Search Property.*

We also consider *monotone* searches, i.e., searchers do not revisit any part of the graphs. In other words, *monotone* means a cleaned node or an cleaned edge can not be re-contaminated. [16] proves that there always exists an optimal strategy that is *Monotone* for *Edge-Search*. It provides the detailed proof to prove that recontamination does not help *Edge-Search*.

There are two distinct contiguous search problems: the *contiguous node-search problem* and the *contiguous edge-search problem*.

In the *contiguous node-search problem*, searchers need to capture an intruder in the nodes of a graph. To achieve that by following a common strategy, the searchers need to surround the intruder so that the intruder does not have an edge to escape through. Formally, the problem is solved when the intruder lies on a vertex x and a set of searchers guard all the vertices related to all the edges out-going from x . The minimal number of searchers required to capture the intruder is called the *contiguous node-search number* ($cns(G)$).

In the *contiguous edge-search problem*, the intruder lies in the edges of the graph. The searchers need to clean all edges as in the standard edge-search problem, i.e., by traversing and then protecting them. The minimal number of searchers required to decontaminate a graph G is called the *contiguous edge-search number* ($ces(G)$).

The contiguous search problems are more difficult than the non-contiguous ones. As mentioned in [17] there exist trees for which minimal search strategy without removal requires at least $\Omega(n \log n)$ steps, whereas if removal is allowed there exists a strategy in each graph that requires at most $O(n)$ steps [16]. Moreover, [6] has proved that the contiguous searching number is always greater or equal to the non-contiguous searching number.

As some variants of the contiguous node/edge-search problems, this thesis develops contiguous search algorithms (or strategies) on four various Mesh topologies in a Mobile Agent based system. Next, we are going to introduce some basic terminologies used in our algorithms in the following section.

1.3 Terminology

1.3.1 The Network

Mesh is a very special type of network topology. It has some unique properties that are distinct from other types of network topologies. For example, a $m \times n$ -Mesh ($m > 0, n > 0$) has m number of rows and n number of columns, each row has n number of nodes and each column has m number of nodes. Moreover, all the nodes

in Mesh are connected to each other by edges. The Mesh network topology has been widely used in reality, such as a local LAN network or a WAN network and so on. In order to take advantage of the particular properties of the Mesh topology, our algorithms are all developed in Mesh topology. Therefore, if we know the given contaminated network is a Mesh then we can perform the clean task by using our algorithms instead of using the general algorithms which will consume more resources.

There are many variants of Mesh topologies, and in this thesis we mainly consider four types of $m \times n$ -Mesh topologies as follows:

- $m \times n$ -Mesh
- $m \times n$ -Hexagonal Mesh
- $m \times n$ -Octagonal Mesh
- $m \times n$ -Toroidal Mesh

An example figure for each of the four types of Mesh topologies is given in the following Figure 1.1.

In a $m \times n$ -Mesh ($m > 0, n > 0$), the first row is labelled as r_0 , the second row is labelled as r_1 and etc, and the last row is labelled as r_{m-1} . Similarly, the first column is labelled as c_0 , the second column is labelled as c_1 and etc, and the last column is labelled as c_{n-1} .

To label a single node in a given Mesh topology, we use the position of the node. For example, in a $m \times n$ -Mesh ($m > 0, n > 0$), if a node i is located at row r_x ($0 \leq x < m$) and column c_y ($0 \leq y < n$) in the Mesh then the node i is labelled

as $P(x, y)$. For example, for the node located at the left upper most point, its node position is labelled as $P(0, 0)$; and the node located at the right lower most point, its node position is labelled as $P(m - 1, n - 1)$.

The position of a node in a mesh is used to label the node i . Compass-Labeling is used to label the direction of i , and all the neighboring nodes and neighboring edges of i . The technique of compass-labelling is illustrated in Figure 1.2 as follows.

In the Mesh shown in Figure 1.2, there are three types of nodes in term of their positions. That means each of the nodes may either have 4 neighbors or 3 neighbors or 2 neighbors. For example, in Figure 1.2, node x has 4 neighbors and node y and y' have 3 neighbors each, and node z and node z' have 2 neighbors each. If we are given a node i , we label E, S, W, N to represent the *EAST, SOUTH, WEST, NORTH* direction of the node i . And we label E_E, E_S, E_W, E_N to represent the *EAST, SOUTH, WEST, NORTH* direction of the neighboring links of i . Also, we label N_E, N_S, N_W, N_N to represent the *EAST, SOUTH, WEST, NORTH* direction of the neighboring nodes of i .

1.3.2 Agent Based System

The application of mobile agents is becoming increasingly popular in the networked environment. Computing in such an environment is termed as distributed mobile computing. This thesis will conduct and develop contiguous node/edge-search strategies by using mobile agents in an agent based system. In the distributed mobile computing systems using mobile agents, the environment is a set of anonymous nodes that

Figure 1.1: An example of four types of Mesh topologies

Figure 1.2: Compass-Labeling in Mesh.

are connected by edges in a network with asynchronous mobile agents. Following are the nodes' and agents' definitions defined in an agent based system :

- **The Nodes:**

1. In an agent based system, all nodes are identical except one node which is defined as *Home Base*. Each of the nodes has a *Whiteboard* and some distinct ports.

- *Whiteboard:*

Each node has a *Whiteboard* associated with it. The *Whiteboard* is a storage area with finite amount of memory (at most $\log n$ bits where n is the number of nodes in the system) in a node and is used for agents communication. The access to a *Whiteboard* is done in mutual exclusion manner.

- Ports:

Each node has some distinct ports linking to other nodes through edges.

2. *Home Base:*

The *Home Base* is a node where all mobile agents start from. In this thesis, we assume that there is only one *Home Base* in a system to achieve the contiguous search.

- **The Agents:**

Agents are processes that navigate or move from a host to its neighboring host,

and perform computations at each host according to a predefined set of behavioral rules called protocols. They have the following properties:

1. Mobility:

Agents can move from a node to neighboring nodes in the system.

2. Functionality:

An agent can read messages from *Whiteboard* and write messages onto the *Whiteboard* to communicate with other agents. It has its own memory to store information obtained from the *Whiteboard* or from the neighboring nodes. It can also perform local computations predefined by protocols (such as a searching protocol).

3. Asynchronicity:

The movement and computation of agents are asynchronous. In other word, an agent takes a finite but unpredictable amount of time to traverse a link (or an edge) or to perform a local computation.

4. Visibility:

Some agents may have limited visibility power. The visibility power allows agents to see all the other agents located at their adjacent neighboring nodes.

1.3.3 Asynchronous Model

Given a mobile agent based system, this thesis develops contiguous node/edge-search algorithms (or strategies) on variant Mesh topologies with *Asynchronous* Model. In

this thesis, we consider two variations of the *Asynchronous* model, one model is when agents do not have visibility power and the other model is when agents have visibility power.

- **Searching Strategies - with a *Synchronizer* :**

In this type of searching strategies, agents do not have visibility power. Hence, one *Synchronizer* is required. The *Synchronizer* is an agent that coordinates the movement of the other agents. Agents can not move unless they are informed by the *Synchronizer* in the search.

- **Searching Strategies - Agent with Visibility :**

Agents have visibility power in this type of searching strategies. They can see whether their neighboring nodes and edges are cleaned by another searcher or not and then decide whether they can move to the next neighboring node or guard the current node. When the agents have this power, a *Synchronizer* is not needed in the search.

1.3.4 Searching Operations

The contiguous node/edge-search problems consist of having an intruder in the graph that has complete knowledge of the graph and of the searchers' movements, and that may move at an unbounded speed only in unguarded nodes or edges. To make the problem worse, the searchers cannot see the intruder. Our objective is to define a strategy that allows the capture of the intruder by using a *minimal* number of searchers.

To capture the intruder, there are two basic operations defined in contiguous node/edge-search strategies.

1. *place* a searcher on a node.
2. *move* a searcher along an edge so that the set of visited and cleaned edges form a connected subgraph.

The above two operations are able to capture the intruder and make the graph clean. However, the definition of cleaning of a graph is a bit different between contiguous node-search and edge-search, which is summarized as follows.

- Node-Search:

In the node-search problem, the intruder “hides” on nodes. To capture the intruder, searchers must clean the nodes contaminated by the intruder and surround it (i.e., the intruder has to lie on a node i and all the nodes belonging to all the edges out-going from i contain at least one searcher). In this case, to capture the intruder, the node has to be surrounded by searchers. Hence, in node-search, we define the following definitions.

1. A node i is *clean* (or *guarded*) :

Either i contains a searcher (i.e. it is guarded by a searcher) or all its incident edges are clean. In other word, a cleaned node i may be re-contaminated unless i is guarded or all of its incident edges are clean.

2. A graph G is *clean* :

All nodes in G are clean.

- **Edge-Search:**

In the edge-search problem, the intruder “hides” on edges. To capture the intruder, searchers must clean the edges contaminated by the intruder and surround it (i.e., the edge where the intruder lies needs to be visited by a searcher, and the intruder should not be able to escape). In this case, to capture the intruder, the edge has to be surrounded by the searchers. Hence, in edge-search, we define the following definitions.

1. A node i is *clean* (or *guarded*) :

Same as that defined in the node-search.

2. An edge e is *clean* :

e is traversed by a searcher and the nodes on both ends of e are either clean or guarded by searchers.

3. A graph G is *clean* :

All nodes and edges in G are clean.

1.3.5 Complexity Measures

The efficiency of each algorithm is evaluated in terms of number of agents, number of steps and time units consumed.

- **Number of Searchers:**

It is the total number of agents that are needed to perform the searching task.

- **Number of Searching Steps:**

In our calculation, one searching step (or movement) is defined as an agent moves from one node to a neighboring node along the edge connecting the nodes. The number of searching steps is the total number of searching steps performed by the searchers during the cleanup of a given Mesh.

- **Time Units:**

Our model is *Asynchronous*, which means it is impossible to calculate the cleanup time. Therefore, we have to consider *Ideal Time*, i.e., assuming that it takes 1 unit of time for an agent to traverse a link. Moreover, we assume that more than one searching agents can move on the same link (or edge) at the same time.

1.4 The Results

To determine the optimal searching team size for the contiguous node/edge-search problem in a general graph is a *NP*-complete problem [17].

In this thesis, we first prove that the contiguous node-search problem in a $m \times n$ -Mesh ($m \leq n$) requires at least m searchers. Based on that, using $O(m)$ agents, we proposed 14 contiguous node/edge-search algorithms on four variant types of *Mesh* topologies. Among these algorithms, some strategies use exactly m searchers and some may use more than m searchers depending on the type of Mesh topologies. However, in the worst case, our strategies only consume maximum $(2m + 1)$ number of searchers such as the node-search in a $m \times n$ -Toroidal Mesh, which is still an

optimal number in order of magnitude. By analyzing our searching algorithms, we observed that our results confirm that the $cns(G) \leq ces(G)$ equation holds in all the four types of Mesh topologies, this bound is proved in [6].

Our searching strategies are of two distinct types. In one type of strategy, a *Synchronizer* is required to coordinate the movements of the other searching agents. In the other type of the strategy, all the searching agents have visibility power. In this case, the *Synchronizer* is not required because each agents can locally decide when to move. By comparing our searching results, we conclude that in node-search and when agents have the visibility power, the searching team size is smaller. In other word, we use less agents in node-search when searching agents have the visibility power. On the other hand, in edge-search, the searching team sizes are the same in both types of searching strategies.

In addition, our results show that the number of search steps is $O(mn)$ in all of our node/edge-search algorithms. The search steps neither depend on the agent visibility power nor the Mesh topology. This is because the searchers need to traverse the whole Mesh from the first column to the last column and make sure no recontamination occurs.

Finally, we calculated the searching time and we conclude that in node-search, the searching time is $O(n)$ when agents have the visibility power except in the $m \times n$ -Octagonal Mesh. However, the searching time is $O(mn)$ in both node-search when agents do not have the visibility power and the edge-search.

The remaining chapters are organized as follows: in chapter 2, we design and

analyze the algorithms of node/edge-search in a $m \times n$ -Mesh graph. Then algorithms of node/edge-search in a $m \times n$ -Hexagonal Mesh are detailed and analyzed in chapter 3. Next algorithms of node/edge-search in a $m \times n$ -Octagonal Mesh and $m \times n$ -Toroidal Mesh are proposed and analyzed in chapter 4 and chapter 5 respectively. Finally, a conclusion is given in chapter 6.

Chapter 2

Node/Edge-Search in a $m \times n$ -Mesh

For each type of Mesh Topology as listed in the previous section, we will design search algorithms or strategies to perform the Node/Edge-Search process by using Mobile Agents.

All of our search algorithms are described in Pseudo-Code format. And for the purpose of description, the name of each algorithm starts with *Search* followed by the type of Mesh, then followed by the type of search strategies. For example, the algorithm *Search_{Mesh} – Synr* is the name for Node/Edge-Search in a $m \times n$ -Mesh and with a *Synchronizer*, and the algorithm *NodeSearch_{Mesh} – Visb* is the name for Node-Search in a $m \times n$ -Mesh and agents with viability power. W.l.o.g, we make the following assumptions that are applied to all of our Pseudo-Codes:

1. $m \leq n$ for the four types of $m \times n$ -Mesh ($m > 0, n > 0$).
2. The network and mobile agents are reliable.

3. Initially, all agents start from *Home Base* node.
4. Operations described in the Pseudo-Codes are taken by agents only in the nodes of Mesh.

In addition, each Pseudo-Code also consists of two distinct phases:

1. *Initialization Phase.*

All the searchers have entered the network in the *Home Base* which is the initial position, and then need to move to their starting points.

2. *Clean-Up Phase.*

The network is cleaned in a contiguous and monotone way.

For each search algorithm, we will provide a formal proof of its correctness. The correctness of all the algorithms presented below is based on a same principle, i.e. we prove that the cleaning process is contiguous and monotone. In addition, we calculate the complexities of searching agents, searching steps and cleanup time consumed by each algorithm.

2.1 Lower bound on the number of searchers in a $m \times n$ -Mesh

To find the number of searchers required to capture an intruder in the general graphs is a *NP*-complete problem [17]. However, we can provide the lower bound on the number of searchers in a Mesh. Let us assume that we are given a $m \times n$ -Mesh and

$m \leq n$. In this section, we will show that we need at least m number of searchers to perform the node-search in the $m \times n$ -Mesh ($m \leq n$). The proof is mainly based on the three particular interesting properties of the contiguous node/edge-search problems. As detailed in chapter 1, a search strategy is internal if searchers can only move along the graph edges once they are placed into the network, it is monotone if no recontamination ever occurs, and it is connected if the set of cleaned graph always includes a connected sub graph.

Theorem 1. *The contiguous node-search problem in a $m \times n$ -Mesh ($m \leq n$) requires at least m searchers.*

PROOF :

By contradiction let us assume that $(m-1)$ searching agents ($0 < m-1 < m \leq n$) are sufficient to solve the contiguous node-search problem in a $m \times n$ -Mesh ($m \leq n$) starting from the left upper most corner.

Figure 2.1: Proof of Theorem 1

Consider the first time, during the cleanup of a mesh, that an agent reaches a arbitrary node x in the last row, i.e. x is now guarded but all the other nodes in that

last row are still contaminated. (See, e.g., node x in Figure 2.1.) At most $(m - 1)$ searching agents are available and the mesh is composed of m rows, thus there must be at least an empty row r_i ($0 < i < m - 1$). This row has to be clean, otherwise, if at least a node is contaminated, then the whole row r_i is contaminated as well. This however contradicts the contiguous assumption. In fact r_i would divide the mesh into two disconnected regions, one containing at least a clean (or guarded) node (the *Home Base*), and the other one containing at clean node (x) and at least a contaminated node (any node in the last row different from node x). Thus row r_i must be clean.

As the number of available agents is $(m - 1) < m \leq n$ then there must be at least an empty column c_j , where ($0 < j < n$) and thus this column cannot contain x . However, at this instant of time c_j is contaminated as the node in its last row is contaminated (by construction the only clean node in this last row is x). Thus, the entire column c_j is contaminated but c_j intersects r_i , thus r_i cannot be a clean row. This contradicts what we have concluded before. Therefore, the contiguous node-search problem in a $m \times n$ -Mesh requires at least m searchers. ■

2.2 Searching Strategies - With a Synchronizer

As proved by the Theorem 1, we need at least m searching agents to capture an intruder in a $m \times n$ -Mesh. We assume that agents do not have the visibility power in this strategy. Hence, agents can not coordinate their movements. In the following, we need to use an extra agent who is designated as the *Synchronizer* to synchronize all the other m searching agents in a Mesh.

2.2.1 Node-Search in a $m \times n$ -Mesh

1. Searching Strategy

We assume that, initially, all agents are located at node $P(0,0)$ (the upper left most corner). This point is illustrated in the Step 1 in Figure 2.2 as follows.

Figure 2.2: Node-Search in a 4×5 -Mesh - With a Synchronizer

Then as captured in the Step 2 in Figure 2.2, before the cleaning, each searching agent is informed by the *Synchronizer* S to move to its starting point along the first column but S stays at node $P(0,0)$. By Step 4 in Figure 2.2 there will be one searcher in each node of the first column of the Mesh, and S and one searching agent are at the node $P(0,0)$. At that point the cleaning steps start. The main cleaning steps are captured in Figure 2.3.

Intuitively, the *Synchronizer* S moves down *SOUTH* and forces the searching agents to move *EAST* one at a time, which is shown in (a). When the whole column of agents have moved to the next column, the *Synchronizer* will move *EAST* to the next column too as shown in (b). Then, it will move up *NORTH* and continue to force the searching agents to move *EAST* one at a time, which is shown in (c). Again,

when the whole column of agents have moved to the next column, the *Synchronizer* will move *EAST* to the next column too as shown in (d). Then we repeat the above operations till the Mesh is cleaned.

The formal description of the algorithm is provided in Algorithm $Search_{Mesh} - Synr$ in Table 2.1.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize all the moves of the other m searching agents.

PROOF :

To prove the correctness of the Algorithm $Search_{Mesh} - Synr$ presented above, we need to prove that the cleaning result is contiguous and monotone. We can prove it by induction on the number of cleaned nodes.

1. *Base Case: the first column is clean.*

Initially all the agents are located at the node $P(0,0)$ and that node is clean.

Figure 2.3: Node/Edge-Search in a $m \times n$ -Mesh - With a Synchronizer

ALGORITHM *Search_{Mesh} - Synr*
INPUT:

A contaminated $m \times n$ -Mesh and $(m + 1)$ mobile agents.

OUTPUT:

The given Mesh is clean.

VARIABLES:

Integers: $i := 0, j := 0, k := 0;$

Position: $P(i, j);$

INITIALIZATION:

All $(m + 1)$ agents are located at the node $P(0, 0)$

The *Synchronizer* is labelled as S

```
for ( $i = m - 1; i < 0; i --$ )
    searcher  $i$  is labelled as  $s_i$ 
     $S$  tells  $s_i$  move to  $P(i, 0)$ 
```

CLEAN-UP

```
 $j = 0$ 
while  $j < (n - 2)$ 
     $i = 0$ 
    while  $i \leq (m - 1)$ 
         $S$  moves to  $P(i, j)$ 
         $S$  tells  $s_i$  move to  $P(i, j + 1)$ 
         $i++$ 
     $j++$ ,  $i = m - 1$ 
    while  $i \geq 0$ 
         $S$  moves to  $P(i, j)$ 
         $S$  tells  $s_i$  move to  $P(i, j + 1)$ 
         $i--$ 
     $j++$ ,  $i = 0$ 
if  $j == (n - 2)$ 
    while  $i \leq (m - 1)$ 
         $S$  moves to  $P(i, j)$ 
         $S$  tells  $s_i$  move to  $P(i, j + 1)$ 
         $i++$ 
if edge-search
    if  $j == (n - 2)$ 
         $j++$ ,  $i = m - 1$ 
         $S$  moves to  $P(i, j)$ 
        for ( $k = m - 2; k \geq 0; k --$ )
             $S$  moves to  $P(k, j)$ 
    else
         $S$  moves to  $P(0, j)$ 
        for ( $k = 1; k \leq m - 1; k ++$ )
             $S$  moves to  $P(k, j)$ 
```

Table 2.1: Node/Edge-Search in a $m \times n$ -Mesh - With a Synchronizer

During the initialization steps, *Synchronizer S* forces all the searching agents to move *SOUTH* along the first column and then guard their starting positions. In other word, all the search agents guard along the first column of the mesh after the initialization. It's obvious that the first column is clean and the initialization (or cleaning) result is contiguous and monotone. Then the cleaning steps start.

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other word, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. *Induction Step:*

As defined in our algorithm, at step $(i + 1)$, the *Synchronizer S* will force one agent to move to its *EAST* neighboring node located at the adjacent next column to clean and guard that node. We want to show that the cleaned area is increased and no recontamination occurs. By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, S will force the agent that guards the node located on the border row (either the first row or the last row) to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move. On the other hand, if the agents guard in

two columns C_j and C_{j+1} at the cleaning step i then at the following cleaning step $(i + 1)$, S will continue to move *SOUTH* or *NORTH* along the column C_j and force the agent that guards the next node located on column C_j to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move. In both cases, the node from which the agent has moved during step $(i + 1)$ is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured as S is able to synchronize all the moves of the other m searching agents. ■

Complexity

Theorem 2. *Algorithm $Search_{Mesh} - Synr$ for the solution of the node-search in a $m \times n$ -Mesh uses $(m + 1)$ agents and cleans up the Mesh in $\frac{m^2 + 4mn - 5m - 2}{2}$ search steps and $(mn - 2)$ time units.*

PROOF :

The bound on the number of agents $s = m + 1$ follows directly from the Algorithm $Search_{Mesh} - Synr$ described in the Table 2.1.

To compute the complexity of the searching steps and time, let S denotes the *Synchronizer* and $s_0, s_1, \dots, s_{m-1}$ denote the other m searching agents. In the following chapters, we will use these same notations when computing the complexities of searching steps and time.

The number of steps consumed by the agents is calculated in the following.

During the *INITIALIZATION*, the searching agents perform $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, each of the m searching agents move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps for all m agents. Therefore, the total number of steps consumed by the searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

During the *CLEAN-UP*, S moves $(n - 2)$ steps *EAST* along rows. Along columns, S repeats $\lceil \frac{n-1}{2} \rceil$ times to move *SOUTH*, and repeats $\lfloor \frac{n-1}{2} \rfloor$ times to move *NORTH*, where each column requires $(m - 1)$ steps. So, the total number of steps performed by S are $(n - 2) + (m - 1)(n - 1) = mn - m - 1$.

Therefore, the total number of steps for the node-search on a $m \times n$ -Mesh with a *Synchronizer* are $\frac{m^2 + 2mn - 3m}{2} + (mn - m - 1) = \frac{m^2 + 4mn - 5m - 2}{2}$.

Based on the ideal time assumption, the cleanup time can be calculated as the following.

During the *INITIALIZATION*, searching agents move one by one to their starting points and in 1 time unit some of them reach the next row. We can see that all searching agents start at r_0 and s_{m-1} reaches r_1 at time 1, then s_{m-1} reaches r_2 and s_{m-2} reaches r_1 at time 2, and then s_{m-1} reaches r_3 and s_{m-2} reaches r_2 s_{m-3} reaches r_1 at time 3, and so on. Hence, the searching agents take $(m - 1)$ time units to line up along the first column. During the *CLEAN-UP*, S uses $(m - 1)$ time units to move *SOUTH* or *NORTH* along the $(n - 1)$ columns in the Mesh. In the mean time, all the searching agents s move *EAST* to the next column. Also S needs $(n - 2)$ time units to move *EAST* to the next adjacent column after the searching agents have moved to

and guarded on that column. Therefore, the total cleanup time for the node-search is $(m - 1) + (m - 1)(n - 1) + (n - 2) = mn - 2$. ■

2.2.2 Edge-Search in a $m \times n$ -Mesh

1. Searching Strategy

Note that the strategy employed for the node-search presented in the previous section also performs a cleanup of all the edges except for the edges of the last column.

In edge-search, the *Synchronizer* S will move and repeat the operations illustrated in (a) to (d) in Figure 2.3. At the time when the Mesh is cleaned by node-search, the last column is guarded by the other m searching agents, but none of the m agents can move to clean the edges of the last column. Hence, S needs to perform extra $1 + (m - 1) = m$ steps to clean the last column of the Mesh, where S uses 1 step to move to the last column from the second last column where S stopped after informing all the searching agents to move to the last column in the node-search, then S takes $(m - 1)$ steps to clean the edges on the last column.

The formal description of the algorithm is provided in Algorithm $Search_{Mesh} - Synr$ in Table 2.1.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize the moves of all the other m searching agents.

PROOF :

As defined in our algorithm, the edge-search actually includes both the node-search and extra m steps performed by the *Synchronizer* S to clean the edges of the last column in the Mesh. We had proved that the intruders are captured as S is able to synchronize all the moves of the other m searching agents in the node-search algorithm. Hence, we only need to prove that the extra m steps for edge search are contiguous and monotone. According to our algorithm, the last column is cleaned as S moves along the edges on the last column while all the other m agents guard the m nodes of the last column. Thus, no recontamination can occur. Therefore, the cleaning result of the extra m steps is contiguous and monotone.

By the above proof, we conclude that in edge-search, the intruder is captured as S is able to synchronize all the moves of the other m searching agents. ■

Complexity

Theorem 3. *Algorithm $Search_{Mesh} - Synr$ for the solution of the edge-search in a $m \times n$ -Mesh uses $(m + 1)$ agents and cleans up the graph in $\frac{m^2 + 4mn - 3m - 2}{2}$ searching steps and $(mn + m - 2)$ time units.*

PROOF :

The bound on the number of agents $s = m + 1$ is given by the Algorithm $Search_{Mesh} - Synr$ described in the Table 2.1.

The cleanup steps are calculated as follows.

The number of steps consumed by the searching agents in the edge-search are exactly the same as that in the node-search. So, the total number of steps performed by s are $\frac{m^2 + 2mn - 3m}{2}$.

Before the last column is cleaned, the number of steps consumed by the *Synchronizer* S in the edge-search are same as that consumed in the node-search, which are $(mn - m - 1)$ steps. And there are extra m movements performed by S to clean the edges of the last column. Hence, S uses $(mn - m - 1) + m = mn - 1$ steps in total.

Therefore, the total number of steps for the edge-search on a $m \times n$ -Mesh with a *Synchronizer* are $\frac{m^2+2mn-3m}{2} + (mn - 1) = \frac{m^2+4mn-3m-2}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

The edge-search algorithm is actually the node-search plus the cleaning of the edges on the last column. As we calculated before, the node-searching time is $(mn-2)$, and the cleaning of the last column needs extra $(m - 1)$ time units. Also, S uses 1 time unit to move to the last column before cleaning the last column. Therefore, the total cleanup time for edge-search is $(mn - 2) + (m - 1) + 1 = mn + m - 2$. ■

2.3 Searching Strategies - Agents with Visibility

In this model, we assume that agents have visibility power. With the visibility power, each agent can see the other agents located at its adjacent neighboring nodes and may coordinate its searching operations according to its neighboring agents' moves. In other word, each agent moves independently. Hence, we do not need the *Synchronizer* in this model. Without the *Synchronizer*, agents communicate with each other by using the *Whiteboard* associated with each node in the Mesh. We assume that the *Whiteboard* has no messages (or empty) initially. When an agent arrives at one node, it will read messages from the *Whiteboard*. If the *Whiteboard* is empty, then the agent

writes “CLEAN” message on the *Whiteboard* and cleans the node. Other agents will not need to clean that node when they read a “CLEAN” message from the *Whiteboard*.

Without the *Synchronizer*, we will only use m searching agents to capture the intruder by performing node-search in $m \times n$ -Mesh, which is optimal as proved in Theorem 1. However, we need one extra *Helper* agent in the edge-search to clean the edges on the columns in the Mesh. This is because we need to clean both nodes and edges in the edge-search.

2.3.1 Node-Search in a $m \times n$ -Mesh

1. Searching Strategy

We assume that initially, all the m agents are located at the node $P(0,0)$. At this point the cleaning steps start. The main cleaning steps are shown in Figure 2.4 as follows.

Figure 2.4: Node-Search in a $m \times n$ -Mesh - Agent with Visibility

At the beginning of the cleaning algorithm, all agents wake up to be the searchers s . Each searcher s will independently perform the following operations:

- s reads the *Whiteboard* on the current node.

- If the *Whiteboard* has a “CLEAN” message, s moves *SOUTH* to the next row. s will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* is empty, s writes “CLEAN” message. s guards the current node till it can see that its neighboring nodes N_N , N_W , and N_S except N_E are all clean (or guarded), then s moves *EAST* to the next column. s will repeat this operation till it reaches the last column in Mesh.

The Mesh is cleaned when all the m searching agents reach the last column. For example, in Figure 2.4, after the first 5 steps, the agents will continue to move *EAST* till the last column of the Mesh.

The formal description of the algorithm is provided in Algorithm *NodeSearch_{Mesh} - Visb* in Table 2.2.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by the m agents.

PROOF :

To prove the correctness of the Algorithm *NodeSearch_{Mesh} - Visb* presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction on the number of cleaned nodes.

1. *Base Case: the node $P(0,0)$ is clean.*

Initially all the agents are located at the node $P(0,0)$ and that node is clean.

The base case is true.

ALGORITHM *NodeSearch_{Mesh} – Visb*
INPUT:

A contaminated $m \times n$ -Mesh and m mobile agents.

OUTPUT:

The given Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

Whiteboard: $WB = \emptyset$

INITIALIZATION:

All m agents are located at the node $P(0, 0)$

Agents *Wake-Up* to be the *Searchers* s

CLEAN-UP

for each s

$i = 0, j = 0$ // i, j are local variables for each s

s reads messages from WB at node $P(i, j)$

while $CLEAN$ is on WB

$i++ = 1$

s moves to node $P(i, j)$ // move *SOUTH*

s reads messages from WB at node $P(i, j)$

if $CLEAN$ is not on WB

s writes $CLEAN$ on WB

while $(j < n - 1)$

s guards node $P(i, j)$

if s can see N_N && N_W && N_S are all clean

$j++ = 1$

s moves to node $P(i, j)$ // move *EAST*

Table 2.2: Node-Search in a $m \times n$ -Mesh - Agents with Visibility

2. Assumption Step:

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other word, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, hence no recontamination occurs. The clean border may line up in one single column C_j ($0 < j < n$) or several consecutive columns $C_{j_1}, C_{j_2}, \dots, C_{j_k}$ ($0 < j, k < n$) depending on the state of the cleaning steps. For example, as shown in Figure 2.4, agents guard three consecutive columns at Step 4, the clean border is contiguous and the cleaning result is monotone.

3. Induction Step:

As defined in the Algorithm *NodeSearch_{Mesh} - Visb*, at searching step $(i + 1)$, all the searching agents move *SOUTH* to place themselves on the nodes of the first column such that there will be one agent guards one of the nodes along the first column. Once an agent guards one node of the first column, it moves *EAST* to the next adjacent column if it can see its neighboring nodes except the N_E are all clean. Otherwise, agents continue to guard their current node. By our algorithm, if all the agents guard one single column C_j at cleaning step i then at the following cleaning step $(i + 1)$, all agents can move *EAST* to the next column C_{j+1} to clean and guard the nodes on the column C_{j+1} . This is because all agents can see their neighboring nodes except the N_E are all clean. Therefore, the cleaned area is increased by one more column and no recontamination occurs. On the other hand, if all the agents guard several

consecutive columns $C_{j_1}, C_{j_2}, \dots, C_{j_k}$ at the cleaning step i then at step $(i + 1)$, there will be three possibilities that are listed as follows.

- If an agent sees its neighboring nodes except N_E are all clean or guarded, then it moves *EAST* to the next adjacent column.
- If an agent s sees a “CLEAN” message on the *Whiteboard* at the current node, then it knows this node is already cleaned by another agent s' . s will move *SOUTH* to the next adjacent row.
- Otherwise, agents continue to guard the current node and do not move.

The clean border is contiguous after the cleaning step $(i + 1)$. In both cases, the node from which the agent has moved during step $(i + 1)$, is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have shown that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders can be captured by the m agents.

■

Complexity

Theorem 4. *Algorithm $NodeSearch_{Mesh} - Visb$ for the solution of the node-search in a $m \times n$ -Mesh and agents with visibility power uses m agents and cleans up the graph in $\frac{m^2+2mn-3m}{2}$ steps and $(m + n - 2)$ time units.*

PROOF :

The bound on the number of agents $s = m$ is given by the above algorithm described in Table 2.2.

The cleanup steps are calculated as follows.

The number of steps consumed by searcher s is calculated as follows. During the *CLEAN-UP*, all the m searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to move *SOUTH* to place themselves on the first column. Then, all the m searchers move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps. So, the total number of steps for the searchers are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2+2mn-3m}{2}$.

Therefore, the total number of steps for the node-search on a $m \times n$ -Mesh and agents with visibility are $\frac{m^2+2mn-3m}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

The algorithm steps are illustrated in Figure 2.4. From that figure, we can see that all agents are at r_0 at time 0, then some awake searching agents reach r_1 at time 1. At time 2, one searching agent moves *EAST* to the next column and other searching agents continuously move *SOUTH* to the node located at the next row such that we have one searching agent places on one row in 1 time unit. This process is repeated till a searching agent reaches the last row. Hence, searching agents use $(m - 1)$ time units to line up on the first column. In the mean time, some or the other searching agents move *EAST* to clean the next column. Since we have $(n - 1)$ columns need to be cleaned and searching agents move to the next adjacent column in 1 time unit, thus, each searching agent uses $(n - 1)$ time units to clean from the first column to the last column. In summary, the searching agent located at the last row uses $(m - 1)$ time units to be at the last row and then needs another $(n - 1)$ time units to reach the last column. Therefore, the total cleanup time is $(m - 1) + (n - 1) = m + n - 2$.

2.3.2 Edge-Search in a $m \times n$ -Mesh

1. Searching Strategy

In this model, we need to clean both nodes and edges in a Mesh. As proved by Theorem 1, m searching agents are required to capture the intruder in a node search. For the edge-search in a $m \times n$ -Mesh, on top of the m searching agents, we also use an extra agent designated to be the *Helper* agent. This is because there are m rows in $m \times n$ -Mesh and we have m available agents, and our strategy is defined as one agent guards one node in one row, then no agent can move along the edges on the columns, otherwise the clean nodes will be re-contaminated. Therefore, we need the extra one *Helper* agent to clean the edges on the columns in the graph.

Similar to the node-search described in the previous section, agents in this strategy also use the “CLEAN” message on *Whiteboard* to communicate with each other. Moreover, to choose the *Helper* agent, we need to put “Helper Assigned” message on the *Whiteboard* at node $P(0,0)$. The agent decided to be the *Helper* agent will write the “Helper Assigned” message on that *Whiteboard* to inform other agents.

Let us look at the cleaning strategies in detail. Initially, all the $(m+1)$ agents are located at the node $P(0,0)$. At this point the cleaning steps start. The main cleaning steps are shown in Figure 2.5.

At the beginning of the cleaning algorithm, all agents wake up to be the searchers s . Each searcher s will independently perform the following operations:

- s reads messages from the *Whiteboard* on the current node.
- If the *Whiteboard* on node $P(0, 0)$ has no “Helper Assigned” message, s decides to be the *Helper* and writes “Helper Assigned” message on the *Whiteboard*. All the other agents will know that there is a *Helper* agent by reading the “Helper Assigned” message from the *Whiteboard* on node $P(0, 0)$.
- If the *Whiteboard* has a “CLEAN” message, s moves *SOUTH* to the next row. s will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* does not have a “CLEAN” message, s writes down “CLEAN” message. Then s guards the current node till it can see its neighboring nodes N_N, N_W, N_S except N_E and its neighboring edges E_N, E_W, E_S except E_E are all clean, then s moves *EAST* to the next column. s will repeat this operation till it reaches the last column in Mesh.

At the same time, the *Helper* H will perform the following operations:

Figure 2.5: Edge-Search in a $m \times n$ -Mesh - Agent with Visibility

- If H is in the first row (or no *NORTH* neighbor), it moves *EAST* to the next column. If it can see the current node is guarded by another searcher then H continues to move *SOUTH* till it reaches the last row.
- If H is in the last row (or no *SOUTH* neighbor), it moves *EAST* to the next column. If it can see the current node is guarded by another searcher then H continues to move *NORTH* till it reaches the first row.
- Otherwise, H guards the current node.

The Mesh is cleaned when all the m searching agents reach the last column and the last column is cleaned by the *Helper* agent.

The formal description of the algorithm is provided in Algorithm *EdgeSearch_{Mesh-Visb}* in Table 2.3.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by all the $(m + 1)$ agents.

PROOF :

To prove the correctness of the Algorithm *EdgeSearch_{Mesh-Visb}* presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

1. *Base Case: the node $P(0,0)$ is clean.*

Initially all the agents are located at the node $P(0,0)$ and that node is clean.

The base case is true.

ALGORITHM *EdgeSearch_{Mesh} - Visb*
INPUT:

A contaminated $m \times n$ -Mesh and $(m + 1)$ mobile agents.

OUTPUT:

The given Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0, x := 0, y := 0, D := \emptyset$;

Position: $P(i, j), P(x, y)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All $(m + 1)$ agents are located at node $P(0, 0)$

Agents *Wake-Up* to be the *Searchers* s

CLEAN-UP

for each s

$i = 0, j = 0$ // i, j are local variables for each s

s checks WB on Node $P(i, j)$

if Current Node is node $P(0, 0)$

if *HelperAssigned* is not on WB

s becomes as H and writes *HelperAssigned* on WB

while *CLEAN* is on WB

$i++ = 1$

s moves to $P(i, j)$ // move *SOUTH*

s checks WB on Node $P(i, j)$

if *CLEAN* is not on WB

s writes *CLEAN* on WB and guards node $P(i, j)$

while $(j < n - 1)$

if s can see nodes N_N && N_W && N_S AND edges E_N && E_W && E_S are all clean

$j++ = 1$

s moves to $P(i, j)$ // move *EAST*

for H

$x = 0, y = 1, D = SOUTH$ // $x, y, and D$ are local variables for H

while $(y < n)$

H moves to node $P(x, y)$

if $(D == SOUTH)$

while H can see $P(x, y)$ is guarded && $x < (m - 1)$

$x++ = 1, H$ moves to $P(x, y)$

if $x == (m - 1)$

$y++ = 1, D = NORTH$

else if $(D == NORTH)$

while H can see $P(x, y)$ is clean && $x > 0$

$x-- = 1, H$ moves to $P(x, y)$

if $x == 0$

$y++ = 1, D = SOUTH$

Table 2.3: Edge-Search in a $m \times n$ -Mesh - Agents with Visibility

2. Assumption Step:

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) or three consecutive columns C_j , C_{j+1} and C_{j+2} ($0 < j < n - 2$) as the cleaning result defined by our algorithm.

3. Induction Step:

As defined in Algorithm $EdgeSearch_{Mesh} - Visb$, at the searching step $(i + 1)$, all the searching agents move *SOUTH* to place themselves on the nodes of the first column such that there will be one agent guarding one of the nodes along the first column. Once a searching agent guards one node of the first column, it moves *EAST* to the next adjacent column if it can see its neighboring nodes except the N_E and its neighboring edges except the E_E are all clean. At the same time, the *Helper* agent H cleans the edges on columns by moving from the current node to the neighboring node (either *SOUTH* or *NORTH* neighboring node) located on the adjacent row if the current node is guarded by a searcher. By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, H and one searching agent s located at the border row (either the first or the last row) move *EAST* together to the next column C_{j+1} to clean and guard the neighboring nodes on the

adjacent column C_{j+1} . Other agents still guard their current nodes. Thus, the cleaned area is increased by one more node and one more edge.

On the other hand, if all the agents guard two consecutive columns C_j and C_{j+1} at the cleaning step i then at the following cleaning step $(i + 1)$, one agent moves to its *EAST* neighboring node k located at the column C_{j+1} to clean and guard k if it can see its neighboring nodes except the N_E and its neighboring edges except the E_E are all clean. At the same time, H moves to node k at step $(i + 1)$ since the node that H guards at step i has one searching agent s . Hence the cleaned area is increased by one more node and two more edges as a result of the movements performed by both H and one s . In the mean time, one searching agent s moves *SOUTH* along the first column to increase the cleaned area by one more node and one more edge. Other agents still guard their current nodes. Similarly, if all the agents guard three consecutive columns at the cleaning step i , the cleaning steps and cleaning result will be the same. In all cases, the node and edge from which the agent has moved during step $(i + 1)$ are now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders can be captured by the $(m + 1)$ agents. ■

Complexity

Theorem 5. *Algorithm $EdgeSearch_{Mesh} - Visb$ for the solution of the edge-search*

in a $m \times n$ -Mesh and agents with visibility power requires $(m + 1)$ agents and cleans up the graph in $\frac{m^2+4mn-5m}{2}$ steps and $(mn - m)$ time units.

PROOF :

The bound on the number of agents $s = m + 1$ is given by the above strategy described in Table 2.3.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s during the *CLEAN-UP* can be calculated as follows. All the m agents perform $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to move *SOUTH* to place themselves in the first column. After that, all the m agents move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps. So, the total number of steps used by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2+2mn-3m}{2}$.

The number of steps consumed by the *Helper Agent H* during the *CLEAN-UP* can be calculated as follows. H moves $(n - 1)$ steps *EAST* and $(m-1)$ steps along each single column, and there are $(n-1)$ columns in Mesh. So, H requires $(n - 1) + (n - 1)(m - 1) = mn - m$ steps.

Therefore, the total number of steps for the edge-search on a $m \times n$ -Mesh and agents with visibility power are $\frac{m^2+2mn-3m}{2} + mn - m = \frac{m^2+4mn-5m}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

The main steps defined in the algorithm are illustrated in Figure 2.5. From that figure, we can see that all agents guard r_0 at time 0. Some awake searching agents move *SOUTH* to next row at time 1. H and one searching agent s_0 move *EAST* and some other searching agents move *SOUTH* at time 2. H moves *SOUTH* to the

second row, and the searching agent s_1 moves *EAST* and also the other searching agents continue to move *SOUTH* along the first column at time 3. This process is repeated as H continuously moves *EAST* to the next column and then moves along the columns (*SOUTH* or *NORTH*) while all the searching agents move *EAST* to the next adjacent row. So, the total cleanup time is the time that H consumes to clean the edges along the columns and boarder rows in Mesh. Since H moves *EAST* along the boarder rows in $(n - 1)$ time units and moves along each column in $(m - 1)$ time units and needs to traverse $(n - 1)$ columns. Therefore, the total cleanup time is $(n - 1) + (m - 1)(n - 1) = mn - m$. ■

2.4 Summary of Searching Results on Different Models in a $m \times n$ -Mesh

By the previous calculations, the complexities of our searching algorithms in a $m \times n$ -Mesh are summarized in the following table.

Search in a $m \times n$ -Mesh			
Search Strategies	Number Of	Models	
		<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
Node Search	Searchers	$m + 1$	m
	Steps	$\frac{m^2 + 4mn - 5m - 2}{2}$	$\frac{m^2 + 2mn - 3m}{2}$
	Time	$mn - 2$	$m + n - 2$
Edge Search	Searchers	$m + 1$	$m + 1$
	Steps	$\frac{m^2 + 4mn - 3m - 2}{2}$	$\frac{m^2 + 4mn - 5m}{2}$
	Time	$mn + m - 2$	$mn - m$

As summarized in the above table, the complexity of the node-search algorithm when agents have visibility power is most efficient comparing to the complexities of the other three algorithms. It uses only m searching agents and consumes the least number of searching steps and searching time. Especially, with the agents' visibility power, the searching time is decreased significantly in this algorithm. However, in the edge-search when agents have visibility power, the visibility power of agents does not help to decrease the number of searching agents, steps and time. This is because the agents have to protect all the cleaned nodes and edges during the edge searching. In other words, the condition to move is when the neighboring nodes and edges are all clean except the *EAST* neighboring. Moreover, in our edge-search algorithms, we only use $(m + 1)$ number of agents, but we have m rows in Mesh. Therefore, during the searching, agents have to move one by one to ensure that there is no gap allowing the intruder to re-contaminate the cleaned area. The searching time is also increased

as the m searching agents need to guard the nodes on a column while the column is being cleaned by a *Helper* agent. Therefore, it consumes more steps and time in the edge-search algorithm. On the other hand, in the model with a *Synchronizer*, in both node-search and edge-search algorithms, all the searching agents have to wait the *Synchronizer* to inform them to move. In other words, the *Synchronizer* traverses the Mesh from the first column to the last column while coordinating with the other searching agents in this model. Thus there is no big difference in term of the magnitude of the complexities in these two algorithms.

Chapter 3

Node/Edge-Search in a $m \times n$ -Hexagonal Mesh

As the consequence of Theorem 1, we need at least m searching agents to capture an intruder in a $m \times n$ -Hexagonal Mesh. In fact, the lower bound of the Theorem 1 holds also in the case of the Hexagonal Mesh because the Mesh is a subgraph of the Hexagonal Mesh. Adding the extra links, the number of agents needed to clean the Mesh cannot decrease. We have proved in chapter 2, $(m + 1)$ agents are needed to do either a node-search or edge-search on a $m \times n$ -Mesh network when using a *Synchronizer*, and where 1 is the *Synchronizer* and m is the number of searching agents. Since there is a diagonal in each single cell in a $m \times n$ -Hexagonal Mesh, which is an extra edge than that in a $m \times n$ -Mesh, thus we can estimate that we need at least $(m + 1)$ agents to perform similar search in a $m \times n$ -Hexagonal Mesh. In the following sections, we will show that no extra agents are required and $(m + 1)$ agents

are enough to do either a node-search or an edge search in a $m \times n$ -Hexagonal Mesh, and where 1 is the *Synchronizer*.

3.1 Searching Strategies - With a Synchronizer

3.1.1 Node-Search in a $m \times n$ -Hexagonal Mesh

1 Searching Strategy

In this strategy, with the help of a *Synchronizer* agent, search agents start from the first column, and the agents move to the next column. During their movement, agents make sure that they do not create any possible path allowing the intruder to re-contaminate the area that has already been cleaned. We assume that initially all the agents are located at node $P(m-1, 0)$. The *Synchronizer* S tells each searching agent to move to its starting point so that there will be one searcher in each node of the first column of the Mesh, and S remains in the last node of the first column. At this point the cleaning process starts. The main steps are shown in Figure 3.1.

Figure 3.1: Node/Edge-Search in a $m \times n$ -Hexagonal Mesh - With a Synchronizer

Figure 3.1 is an informal description of the node-searching strategies. The *Syn-*

chronizer S starts in node $P(m-1, 0)$. S allows the searching agent located at $P(m-1, 0)$ to move *EAST* to the next column and S moves to there as well. When the searcher arrives at $P(m-1, 1)$, S uses the diagonal move *NORTHWEST* to the other searching agent located at $P(m-2, 0)$. The process continues in the same way. *Synchronizer* S lets agents move to the next column and it moves as well to make sure that the searching agent has really arrived, then S goes to the next searcher through the diagonal. When an entire column of agents have moved to the next column, S goes down *SOUTH* from the first row to the last row. After that, the same process repeats.

A strategy for the node/edge-search which requires $(m+1)$ mobile agents is presented as Algorithm $Search_{HexaMesh} - Synr$ in Table 3.1, where we need m searching agents to perform the cleaning task and 1 extra agent to be the *Synchronizer*.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize all the moves of the other m searching agents.

PROOF :

To prove the correctness of the Algorithm $Search_{HexaMesh} - Synr$ presented above, we need to prove that the cleaning strategy is contiguous and monotone. We prove it by induction.

1. Base Case: the first column is clean.

Initially all the agents are located at the node $P(m-1, 0)$ and that node is clean.

ALGORITHM *SearchHexaMesh - Synr*
INPUT:

A contaminated $m \times n$ -Hexagonal Mesh and $(m + 1)$ mobile agents.

OUTPUT:

The given Hexagonal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0, k := 0$;

Position: $P(i, j)$;

INITIALIZATION:

All $(m + 1)$ agents are located at the node $P(m - 1, 0)$

The *Synchronizer* is labelled as S

for ($i = 0; i < m; i++$)

 searcher i is labelled as s_i

for ($i = 0; i < m - 1; i++$)

S tells s_i move to $P(i, 0)$

CLEAN-UP

$j = 0$

while $j < (n - 1)$

$i = m - 1$

while $i \geq 0$

S tells s_i moves to $P(i, j + 1)$ with S

if ($i > 0$ && s_i be at $P(i, j + 1)$)

S moves to $P(i - 1, j)$

$i--$

if ($i == 0$ && $j < n - 2$)

$j++$

for ($k = 1; k < m; k++$)

S moves to $P(k, j)$

if (edge-search)

$j = n - 1$

for ($k = 1; k < m; k++$)

S moves to $P(k, j)$

Table 3.1: Node/Edge-Search in a $m \times n$ -Hexagonal Mesh - With a Synchronizer

During the initialization steps, *Synchronizer S* will force all the searching agents to move to and guard their starting positions. In other word, all the searching agents guard along the first column of the mesh after the initialization. It's obvious that the first column is clean and the initialization (or cleaning) result is contiguous and monotone. Then the cleaning steps start.

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other word, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. *Induction Step:*

As defined in our algorithm, at step $(i + 1)$, the *Synchronizer S* will force one agent to move to its *EAST* neighboring node located at the adjacent next column to clean and guard that node. *S* then moves *NORTHWEST* along the diagonal to the next searching agent. Then the cleaned area is increased by one more node and no recontamination occurs.

By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, we have two possibilities as follows.

- If S is on the first row, then S move *SOUTH* till it reaches the last row to clean the edges on the column C_j . All the other m agents guard the nodes on C_j and do not move.
- If S is on the last row, then S will force the agent that guards the node located on the last row to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move.

On the other hand, if all the agents guard two columns C_j and C_{j+1} at cleaning step i then at the following cleaning step $(i+1)$, S will move *EAST* and force the agent that guards the node located on column C_j to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node, then S moves *NORTHWEST*. All the other agents still guard their nodes and do not move. In all cases, the node from which the agent has moved during step $(i+1)$, is now totally surrounded by clean or guarded nodes and hence no recontamination can occur.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured as S is able to synchronize all the moves of the other m searching agents. ■

Complexity

Theorem 6. *Algorithm $Search_{HexaMesh} - Synr$ for the solution of the node-search in a $m \times n$ -Hexagonal Mesh uses $(m+1)$ agents and cleans up the graph in $\frac{m^2+8mn-11m-4n+6}{2}$*

steps and $(3mn - 3m - 2n + 2)$ time units.

PROOF :

The bound on the number of agents $s = m + 1$ is given by the above strategy described in Table 3.1.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents can be calculated as follows. During the *INITIALIZATION*, agents perform $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, all the m searching agents move $(n - 1)$ steps *EAST* for a total of $m(n - 1)$ number of steps. Hence, the total number of steps required by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

The number of steps consumed by a *Synchronizer S* during the *CLEAN-UP* can be calculated as follows. S moves $m(n - 1)$ steps to *EAST*. In addition, S moves $(m - 1)(n - 1)$ steps on the diagonals, and $(m - 1)(n - 2)$ steps *SOUTH* to clean edges on the columns. The total number of steps used by S are $m(n - 1) + (m - 1)(n - 1) + (m - 1)(n - 2) = 3mn - 4m - 2n + 3$.

Therefore, the total number of steps for the node-search on a $m \times n$ -Hexagonal Mesh with a *Synchronizer* are $\frac{m^2 + 2mn - 3m}{2} + (3mn - 4m - 2n + 3) = \frac{m^2 + 8mn - 11m - 4n + 6}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

During the *INITIALIZATION*, the m searching agents take $(m - 1)$ time units to be their starting points. Then during the *CLEAN-UP*, they all move *EAST* for m time units and this needs to be repeated $(n - 1)$ times. S moves *NORTHWEST*

along the diagonal for a total of $(m - 1)(n - 1)$ time units and also moves *SOUTH* along columns for another $(m - 1)(n - 2)$ time units. Therefore, the total cleanup time is $(m - 1) + m(n - 1) + (m - 1)(n - 1) + (m - 1)(n - 2) = 3mn - 3m - 2n + 2$.

■

3.1.2 Edge-Search in a $m \times n$ -Hexagonal Mesh

1. Searching Strategy

As we observe that the strategy employed for the node-search presented in the previous section also performs a cleanup of all the edges except for those edges of the last column.

The *Synchronizer* S will move and repeat the operations illustrated in Figure 3.1. At the time when the Mesh is cleaned by node-search, the last column is guarded by the m searching agents, but none of the agents can move and clean the edges of the last column. Thus, S need to move extra $(m - 1)$ steps along the last column to clean the edges on the last column in the Mesh.

The formal description of the algorithm is provided in Algorithm *Search_{HexaMesh} - Synr* in Table 3.1.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize all the moves of the other m searching agents.

PROOF :

As defined in our algorithm, the edge-search actually includes the node-search and extra $(m - 1)$ moves performed by the *Synchronizer* S to clean the edges of the last column in the Mesh. We had proved that the intruders are captured as S is able to synchronize all the moves of the other m searching agents in the node-search algorithm. Hence, we only need to prove that the extra $(m - 1)$ steps for edge search are contiguous and monotone. According to our algorithm, the last column is cleaned as S moves along the edges on the last column while all the m agents guard the m nodes of the last column. Thus, no recontamination can occur. Therefore, the clean result of the extra $(m - 1)$ steps is contiguous and monotone.

By the above proof, we conclude that in edge-search, the intruder is captured as S is able to synchronize all the moves of the other m searching agents. ■

Complexity

Theorem 7. *Algorithm $Search_{HexaMesh-Synr}$ for the solution of the edge-search in a $m \times n$ -Hexagonal Mesh uses $(m+1)$ agents and cleans up the graph in $\frac{m^2+8mn-9m-4n+4}{2}$ steps and $(3mn - 2m - 2n + 1)$ time units.*

PROOF :

The bound on the number of agents $s = m + 1$ is given by the above strategy described in Table 3.1.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s in the edge-search are exactly same as that in the node-search. So, the total number of steps performed by searching agents are $\frac{m^2+2mn-3m}{2}$.

The number of steps consumed by the *Synchronizer* S in the edge-search are a sum of the searching steps used in the node-search which are $(3mn - 4m - 2n + 3)$ steps and the extra $(m - 1)$ movements performed by S to clean the edges of the last column. So, S consumes $(3mn - 4m - 2n + 3) + (m - 1) = (3mn - 3m - 2n + 2)$ steps.

Therefore, the total steps for the edge-search on $m \times n$ -Hexagonal Mesh with a *Synchronizer* are $\frac{m^2+2mn-3m}{2} + (3mn - 3m - 2n + 2) = \frac{m^2+8mn-9m-4n+4}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

The edge-search algorithm consists of the node-search and the cleaning of the edges on the last column. As we calculated before, the node-search cleanup time is $(3mn - 3m - 2n + 1)$. And the cleaning of the last column uses extra $(m - 1)$ time units. Therefore, the total cleanup time for edge-search is $(3mn - 3m - 2n + 2) + (m - 1) = 3mn - 2m - 2n + 1$. ■

3.2 Searching Strategy - Agents with Visibility

This model is similar to the strategy of node/edge-search in $m \times n$ -Mesh and agents with visibility power where the *Synchronizer* is not required. In this model, we will use m searching agents to capture the intruder by node-search in a $m \times n$ -Hexagonal Mesh. But we need one extra *Helper* agent in the edge-search to clean the edges on the columns in Hexagonal Mesh.

Moreover, agents communicate with each other by using *Whiteboard* associated with each nodes in the Mesh.

3.2.1 Node-Search in a $m \times n$ -Hexagonal Mesh

1. Searching Strategy

Initially, all the m agents are located at the node $P(m-1, 0)$. At this point the cleaning process starts. The main cleaning steps are shown in Figure 3.2.

Figure 3.2: Node-Search in a $m \times n$ -Hexagonal Mesh - Agent with Visibility

At the beginning of the cleaning algorithm, all agents wake up to be the searchers.

Then each searcher s will independently perform the following operations:

- s reads the *Whiteboard* on the current node.
- If the *Whiteboard* has a “CLEAN” message, s moves *NORTH* to the next row. s will contiguously move *NORTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* is empty, s writes “CLEAN” message. s guards the current node till it can see its neighboring nodes N_N , N_W , N_S , and N_{SE} except N_E are all clean (or guarded), then s moves *EAST* to the next column. s will repeat this operation till it reaches the last column in Mesh.

The Mesh is cleaned when all the m searching agents have reached the last column.

ALGORITHM *NodeSearch_{HexaMesh} – Visb*
INPUT:

A contaminated $m \times n$ -Hexagonal Mesh and m mobile agents.

OUTPUT:

The given Hexagonal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All m agents are located at the node $P(m-1, 0)$

Agents *Wake-Up* to be the *Searchers* s

CLEAN-UP

for each s

$i = m-1, j = 0$ // i, j are local variables for each s

s reads messages from WB on Current Node

while $WB = CLEAN$

$i = 1$

s moves to node $P(i, j)$ // move *NORTH*

s reads messages from WB on Current Node

if $WB = \emptyset$

s writes *CLEAN* on WB

s guards node $P(i, j)$

while $(j < n-1)$

if s can see N_N && N_W && N_S && N_{ES} are all clean

$j++$

s moves to node $P(i, j)$ // move *EAST*

Table 3.2: Node-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility

The description of the strategy is provided in Algorithm *NodeSearch_{HexaMesh} – Visb* in Table 3.2.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by all the m searching agents.

PROOF :

To prove the correctness of the Algorithm *NodeSearch_{HexaMesh} – Visb* presented above, we need to prove that the cleaning result is contiguous and monotone. We

prove it by induction.

1. *Base Case: the node $P(m - 1, 0)$ is clean.*

Initially, all the agents are located at the node $P(m - 1, 0)$ and that node is clean. The base case is true.

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes. For example, as shown in Figure 3.2, we can see that cleaning border is contiguous.

3. *Induction Step:*

As defined in Algorithm *NodeSearch_{HexaMesh} - Visb*, during the cleanup, all the searching agents move *NORTH* one by one to place themselves on the nodes of the first column such that there will be one searching agent to guard each of the nodes along the first column. Once an agent guards a node on the first column, it moves *EAST* to the next adjacent column if it can see its neighboring nodes except the N_E are all clean.

By our algorithm, at the cleaning step $(i + 1)$, there will be three possibilities that are listed as follows.

- If an agent sees its neighboring nodes except N_E are all clean or guarded, then it moves *EAST* to the next adjacent column.
- If an agent s sees a "CLEAN" message on the *Whiteboard* at the current

node, then it knows this node is already cleaned by another searching agent s' . s will move *NORTH* to the next adjacent row.

- Otherwise, searching agents guard the current node and do not move.

The clean border is contiguous after the cleaning step $(i + 1)$. In all cases, the node from which the agent has moved during step $(i + 1)$, is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured by all the m searching agents. ■

Complexity

Theorem 8. *Algorithm $NodeSearch_{HexaMesh} - Visb$ for the solution of the node-search in a $m \times n$ -Hexagonal Mesh uses m agents and cleans up the graph in $\frac{m^2 + 2mn - 3m}{2}$ steps and $(m + 2n - 4)$ time units.*

PROOF :

The bound on the number of agents $s = m$ is given by the above strategy described in Table 3.2.

The number of steps consumed by searching agents s during the *CLEAN-UP* is calculated as follows. The m agents use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to move *NORTH* to place themselves on the first column. Then, all the m searchers move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps. So, the total number of steps required by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

Therefore, when agents with the visibility power, the total number of steps for the node-search on a $m \times n$ -Hexagonal Mesh are $\frac{m^2+2mn-3m}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

As calculated in the previous section of the $m \times n$ -Mesh when agents have visibility power, the m searching agents in this mesh also guard the nodes of the first column in $(m - 1)$ time units. Once all the searching agents guard the first column, let us look at the searching agents s_0 guarding the first row (this agent is the last agent arrives at its starting point). In 1 time unit, s_0 moves *EAST* to the next adjacent column, then in the next 1 time unit s_0 can not move and needs to wait at the current node. This is because s_0 needs to wait for s_1 to move *EAST* to clean its neighboring node N_{ES} in 1 time unit. Hence, s_0 moves *EAST* to next adjacent column in every 2 time units and this process is repeated till it reaches the last column. In other words, s_0 moves *EAST* to the next column in every 2 time units and this process needs to be repeated $(n - 1)$ times till it reaches the last column. And also s_0 needs only 1 time unit for the last column since it does not have a neighboring node N_{ES} when it guards the last column. Meanwhile, all the other searching agents move *EAST* till they reach the last column. Therefore, the total clean up time is: $(m - 1) + 2(n - 1) - 1 = m + 2n - 4$. ■

3.2.2 Edge-Search in a $m \times n$ -Hexagonal Mesh

1. Searching Strategy

In this model, we need to clean both nodes and edges as well in the Mesh. In

order to capture the intruder, we have to use m searching agents and an extra agent designated to be the *Helper* agent. This is because we need the *Helper* agent to clean the edges on the diagonals and columns in the Mesh.

In addition, agents use the “CLEAN” message on *Whiteboard* to communicate with each other. Moreover, the agent decided to be the *Helper* agent will write “Helper Assigned” message on the *Whiteboard* to inform the other agents.

Initially, all the $(m + 1)$ agents are located at the node $P(m - 1, 0)$. At this point the cleaning process starts. The major cleaning steps are shown in Figure 3.3.

Figure 3.3: Edge-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility

As shown in Figure 3.3, at the beginning of the cleaning algorithm, all agents wake up to be the searcher s . Each searcher s will independently perform the following operations:

- s reads messages from the *Whiteboard* on the Current Node.
- If the *Whiteboard* on node $P(m - 1, 0)$ is empty, s decides to be the *Helper* and writes “Helper Assigned” message on the *Whiteboard*.
- If the *Whiteboard* has a “CLEAN” message, s moves *NORTH* to the next row.

s will contiguously move *NORTH* till it reaches a node on which the *Whiteboard* is empty.

- If the *Whiteboard* does not have a “CLEAN” message, s writes “CLEAN” message. And s guards the current node till it can see its neighboring nodes N_N , N_W , N_S , N_{SE} except N_E and its neighboring edges E_N , E_W , E_S , E_{SE} except E_E are all clean, then s moves *EAST* to the next column. s repeats this operation till it reaches the last column in Mesh.

At the same time, the *Helper H* performs the following operations:

- H moves *EAST* and then if the current row is not the first row, H moves *NORTHWEST* along the diagonal to the neighboring node of N_{NW} if it can see that the current node and the node N_{NW} are all guarded.
- If H reaches the first row, it moves *EAST* to the next column and then moves *SOUTH* from the first row to the last row to clean the edges on the column.
- Otherwise, H guards the current node.

The Mesh is cleaned when all the m searching agents have reached the last column and the last column is cleaned by the *Helper* agent.

A description of the algorithm is provided in Algorithm *EdgeSearch_{HexaMesh} – Visb* in Table 3.3.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by all the $(m + 1)$ agents.

ALGORITHM *EdgeSearch_{HexaMesh} - Visb*
INPUT:

A contaminated $m \times n$ -Hexagonal Mesh and $(m + 1)$ mobile agents.

OUTPUT:

The given Hexagonal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0, k := 0$;

Position: $P(i, j)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All $(m + 1)$ agents are located at the node $P(m - 1, 0)$

Agents *Wake-Up* to be the *Searchers s*

CLEAN-UP

for each s

```

     $i = m - 1, j = 0$  //  $i, j$  are local variables for each  $s$ 
     $s$  reads messages from  $WB$  on node  $P(i, j)$ 
    if Current Node is node  $P(m - 1, 0)$ 
        if HelperAssigned is not on  $WB$ 
             $s$  becomes as  $H$  and writes HelperAssigned on  $WB$ 
    while CLEAN is on  $WB$ 
         $i - = 1$ ,  $s$  moves to  $P(i, j)$  // move NORTH
         $s$  reads messages from  $WB$  on node  $P(i, j)$ 
    if CLEAN is not on  $WB$ 
         $s$  writes CLEAN on  $WB$  and guards node  $P(i, j)$ 
        while ( $j < n - 1$ )
            if  $s$  can see nodes  $N_N$  &&  $N_W$  &&  $N_S$  &&  $N_{SE}$  AND
            edges  $E_N$  &&  $E_W$  &&  $E_S$  &&  $E_{SE}$  are all clean
                 $j + = 1$ ,  $s$  moves to  $P(i, j)$  // move EAST

```

for H

```

     $i = m - 1, j = 0$ 
    while ( $j < n - 1$ )
         $H$  waits at  $P(i, j)$ 
        while ( $i > 0$ )
             $H$  moves to  $P(i, j + 1)$ 
            if  $H$  can see  $N_{NW}$  and the current node at  $P(i, j + 1)$  are all guarded
                 $H$  moves to  $P(i - 1, j)$ 
                 $i - = 1$ 
        if ( $i == 0$ )
             $j + = 1$ 
             $H$  moves to  $P(i, j)$ 
            for ( $k = 1; k < m; k + +$ )
                if  $N_S$  is clean &&  $P(k, j)$  is guarded
                     $H$  moves to  $P(k, j)$ 

```

Table 3.3: Edge-Search in a $m \times n$ -Hexagonal Mesh - Agents with Visibility

PROOF :

To prove the correctness of the Algorithm $EdgeSearch_{HexaMesh} - Visb$ presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

1. *Base Case: the node $P(m - 1, 0)$ is clean.*

Initially all the agents are located at the node $P(m - 1, 0)$ and that node is clean. The base case is true.

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other word, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes.

3. *Induction Step:*

As defined in our algorithm $EdgeSearch_{HexaMesh} - Visb$, at step $(i + 1)$, a searching agent s moves *EAST* to the next column if it can see its neighboring nodes N_N, N_W, N_S, N_{SE} except N_E and its neighboring edges E_N, E_W, E_S, E_{SE} except E_E are all clean. At the same time, H moves *EAST* and then H moves *NORTHWEST* along the diagonal if it can see that N_{NW} and the current node are all guarded. All the other searching agents still guard their nodes and do not move. In fact the cleaned area is increased by one more node and one more edge as illustrated in Figure 3.3. Therefore, the node and edge from which the agent has moved during step $(i + 1)$ is now totally surrounded by clean or

guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders can be captured by the $(m + 1)$ agents. ■

Complexity

Theorem 9. *Algorithm $EdgeSearch_{HexaMesh} - Visb$ for the solution of the edge-search in a $m \times n$ -Hexagonal Mesh uses $(m + 1)$ agents and cleans up the graph in $\frac{m^2 + 8mn - 9m - 4n + 4}{2}$ steps and $(3mn - 3m - 2n + 3)$ time units.*

PROOF :

The bound on the number of agents $s = m + 1$ is given by the above strategy described in Table 3.3.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s during the *CLEAN-UP* can be calculated as follows. The m searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to move *NORTH* to place themselves on the first column. After that, all the m agents move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps. Hence, the total number of steps used by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

The number of steps consumed by the Helper Agent H during the *CLEAN-UP* is calculated as follows. H moves $m(n - 1)$ steps *EAST*, $(m - 1)(n - 1)$ steps along the diagonal, and $(m - 1)(n - 1)$ steps *SOUTH*. The total number of steps required by H are $m(n - 1) + (m - 1)(n - 1) + (m - 1)(n - 1) = 3mn - 3m - 2n + 2$.

Therefore, when agents with the visibility power, the total number of steps for the edge-search on a $m \times n$ -Hexagonal Mesh are $\frac{m^2+2mn-3m}{2} + (3mn - 3m - 2n + 2) = \frac{m^2+8mn-9m-4n+4}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

By our algorithm, the cleanup time is 1 time unit plus the total time that the *Helper* agent H traverses the Mesh. Initially, at time 0, all agents guard the node $P(m-1, 0)$. When the edge-search starts, the wake up searching agents move *NORTH* to their starting points, and H moves *EAST* to the next adjacent column at time 1. At time 2, the searching agent guards the last row moves *EAST* to the node that H guards. Then H moves *NORTHWEST* in 1 time unit to clean the edge on diagonal and then moves *EAST* with one searching agent in another 1 time unit to clean the edge on row while the other awake searching agents either move *NORTH* or guards on node. This process is repeated till H reaches the first row, and H consumes $(2m-1)$ time units in this process. This process is only for cleaning one single column and needs to be repeated $(n-1)$ times. And then H uses $(m-1)$ time units to move *SOUTH* from the first row to the last row and clean the edges on column, and also this needs to be repeated $(n-1)$ times. During that time, all the other searching agents move *EAST* to the next adjacent column. Therefore, the total cleanup time is $1 + (2m-1)(n-1) + (m-1)(n-1) = 3mn - 3m - 2n + 3$. ■

3.3 Summary of Searching Results on Different Models in a $m \times n$ -Hexagonal Mesh

By the previous calculations, the complexities of our searching algorithms in a $m \times n$ -Hexagonal Mesh are summarized in the following table.

Search in a $m \times n$ -Hexagonal Mesh			
Search Strategies	Number Of	Models	
		<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
Node Search	Searchers	$m + 1$	m
	Steps	$\frac{m^2 + 8mn - 11m - 4n + 6}{2}$	$\frac{m^2 + 2mn - 3m}{2}$
	Time	$3mn - 3m - 2n + 2$	$m + 2n - 4$
Edge Search	Searchers	$m + 1$	$m + 1$
	Steps	$\frac{m^2 + 8mn - 9m - 4n + 4}{2}$	$\frac{m^2 + 8mn - 9m - 4n + 4}{2}$
	Time	$3mn - 2m - 2n + 1$	$3mn - 3m - 2n + 3$

By comparing the above searching results with the searching results in a $m \times n$ -Mesh, the conclusions are similar in both type of Mesh topologies, i.e., the complexity of the node-search algorithm when agents have visibility power is the most efficient one comparing to the complexities of the other three algorithms. However, in the edge-search when agents have visibility power, the visibility power of agents does not help to decrease the number of searching agents, steps and time. In addition, from the above table, we can see that the magnitudes of searching steps and times are increased and bigger than that in the $m \times n$ -Mesh, this is because the $m \times n$ -Hexagonal Mesh

has more edges due to the diagonal. Therefore, agents consume more searching steps and time to clean the extra edges on the diagonal.

Chapter 4

Node/Edge-Search in a $m \times n$ -Octagonal Mesh

As depicted in Figure 1.1 (c) in chapter 1, a $m \times n$ -Octagonal Mesh has two diagonals in each cell, one extra diagonal than that in a $m \times n$ -Hexagonal Mesh. And we have proved that we need $(m + 1)$ agents to perform either a node-search or an edge-search in a $m \times n$ -Hexagonal Mesh. Thus we know that we need at least $(m + 1)$ agents to perform a similar search in a $m \times n$ -Octagonal Mesh. However, with the extra edge or diagonal in each cell in a $m \times n$ -Octagonal Mesh, we need another extra agent to guard the diagonal. Therefore, the searching team size for our strategies is $(m + 2)$ agents, where $(m + 1)$ agents are the searchers and 1 agent is the *Synchronizer* to coordinate all the other searching agents.

4.1 Searching Strategies - With a Synchronizer

4.1.1 Node-Search in a $m \times n$ -Octagonal Mesh

1. Searching Strategy

For the node-search, the informal description of the strategy is that agents start from the first column of the Mesh and then move towards *EAST* with the help of the *Synchronizer*.

We assume that initially, all $(m + 2)$ agents are located at node $P(0, 0)$. Each searching agents moves *SOUTH* to line up on the nodes of the first column in the Mesh. The node at $P(0, 0)$ has two searching agents and the *Synchronizer* S . At this point the cleaning process starts. The main cleaning steps are captured in Figure 4.1.

Figure 4.1: Node-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer

Initially, the *Synchronizer* S forces each searcher to move *SOUTH* to its starting point such that there will be at least one searching agent in each node of the first column of the Mesh. At this point, as shown in Figure 4.1, S tells one searching agent at node $P(0, 0)$ to move *EAST* and another searching agent to move *SOUTHEAST*

with S . Then S goes back *WEST* and continues in the same way till all the searching agents have reached the next column. At this point the procedure will be repeated, but this time with a bottom-up synchronization. This process stops if S reaches the last node $P(m - 1, n - 1)$.

A formal description of the strategy for node-search which requires $(m + 2)$ agents is presented as Algorithm $NodeSearch_{OctaMesh} - Synr$ in Table 4.1.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize all the moves of the other $(m + 1)$ searching agents.

PROOF :

To prove the correctness of the Algorithm $NodeSearch_{OctaMesh} - Synr$ presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

1. Base Case: the first column is clean.

Initially all the agents are located at the node $P(0, 0)$ and that node is clean.

During the initialization steps, *Synchronizer* S forces all the searching agents to move *SOUTH* along the first column and then guard their starting positions.

In other words, all the searching agents guard along the first column of the mesh after the initialization. It's obvious that the first column is clean and the initialization (or cleaning) result is contiguous and monotone. Then the cleaning steps start.

Algorithm *NodeSearch_{OctaMesh} – Synr*
INPUT:

A contaminated $m \times n$ -Octagonal Mesh and $(m + 2)$ mobile agents

OUTPUT:

The given Octagonal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

INITIALIZATION:

All $(m + 2)$ agents are located at node $P(0, 0)$

The *Synchronizer* is labelled as S

for $(i = 0; i \leq m; i++)$

 searcher i is labelled as s_i

for $(i = m - 1; i < 0; i--) // s_m$ stays at $P(0, 0)$

S tells s_i move to $P(i, 0)$

CLEAN-UP

$j = 0$

while $j \leq n - 2$

$i = 0$

S tells s_m move to $P(i, j + 1)$ // move *EAST*

 while $i \leq m - 2$

S tells s_i move to $P(i + 1, j + 1)$ with S // move *SOUTHEAST*

 if $P(i + 1, j + 1)$ is guarded by a s

S moves to $P(i + 1, j)$ // move *WEST*

$i++ = 1$

 if $i == m - 1$

S tells s_i move to $P(i, j + 1)$ with S // move *EAST*

$j++ = 1$

 if $j \neq n - 1$

S tells s_i move to $P(i, j + 1)$ // move *EAST*

 while $i > 0$

S tells s_{i-1} move to $P(i - 1, j + 1)$ with S // move *NORTHEAST*

 if $P(i - 1, j + 1)$ is guarded by a s

S moves to $P(i - 1, j)$ // move *WEST*

$i-- = 1$

 if $i == 0$

S tells s_m move to $P(i, j + 1)$ with S // move *EAST*

$j++ = 1$

Table 4.1: Node-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line up in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. *Induction Step:*

As defined in our algorithm, at step $(i + 1)$, the *Synchronizer* S will force one agent to move to its *SOUTHEAST* (or *NORTHEAST*) neighboring node located at the adjacent next column to clean and guard that node. We want to show that the cleaned area is increased by one more node and no recontamination occurs.

By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, S will force the searching agent that guards the node located on the border row (either the first row or the last row) to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move. On the other hand, if all the searching agents guard two columns C_j and C_{j+1} at the cleaning step i then at the following cleaning step $(i + 1)$, there will be two possibilities listed as follows.

- If S has a N_{SE} (or N_{NE}) neighboring node, S will continue to move along

the diagonal and force the agent that guards the node located on column C_j to move *SOUTHEAST* (or *NORTHEAST*) to the next column C_{j+1} to clean and guard the *SOUTHEAST* (or *NORTHEAST*) neighboring node. All the other agents still guard their nodes and do not move.

- If S does not have a N_{SE} (or N_{NE}) neighboring node, S will force the agent that guards the node located on column C_j to move *EAST* the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move.

In all cases, the node from which the agent has moved during step $(i + 1)$ is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured as S is able to synchronize all the moves of the other $(m + 1)$ searching agents. ■

Complexity

Theorem 10. *Algorithm $NodeSearch_{OctaMesh} - Synr$ for the solution of the node-search in a $m \times n$ -Octagonal Mesh uses $(m + 2)$ agents and cleans up the graph in $\frac{m^2 + 6mn - 7m}{2}$ steps and $(2mn - m - n)$ time units.*

PROOF :

The bound on the number of agents $s = m + 2$ is given by the above strategy described in Table 4.1.

The number of steps consumed by searching agents s and *Synchronizer* can be calculated as follows.

During the *INITIALIZATION*, the searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps move *SOUTH* to place themselves on the first column. During the *CLEAN-UP*, all the $(m + 1)$ agents move $(n - 1)$ steps either along the diagonal or *EAST* for a total of $(m + 1)(n - 1)$ steps. The total number of steps used by searching agents are $\frac{m(m-1)}{2} + (m + 1)(n - 1) = \frac{m^2 + 2mn - 3m + 2n - 2}{2}$.

The number of steps consumed by *Synchronizer* S can be calculated as follows. During the *CLEAN-UP*, S moves $(n - 1)$ steps *EAST*, $(m - 1)(n - 1)$ steps along the diagonal, and $(m - 1)(n - 1)$ steps *WEST*. The total number of steps required by S are $(n - 1) + 2(m - 1)(n - 1) = 2mn - 2m - n + 1$.

Therefore, the total number of steps for the node-search on a $m \times n$ -Octagonal Mesh with a *Synchronizer* are $\frac{m^2 + 2mn - 3m + 2n - 2}{2} + 2mn - 2m - n + 1 = \frac{m^2 + 6mn - 7m}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

During the *INITIALIZATION*, it takes the m searching agents $(m - 1)$ time units to place themselves on the first column. During the *CLEAN-UP*, to clean one single column (from the first row to the last row), the *Synchronizer* S moves *SOUTHEAST* along the diagonal in $(m - 1)$ time units and moves *WEST* in another $(m - 1)$ time units. There are $(n - 1)$ columns need to be cleaned, this process needs to be repeated $(n - 1)$ times. Also, S uses $(n - 1)$ time units to move *EAST* to the next neighboring column. During that time, all the searching agents are forced by S to move *EAST* to the next column. Therefore, the total cleanup time is $(m - 1) + 2(m - 1)(n - 1)$

$$+ (n - 1) = 2mn - m - n. \blacksquare$$

4.1.2 Edge-Search in a $m \times n$ -Octagonal Mesh

1. Searching Strategy

In edge-search, we need to clean both nodes and edges in the Mesh. As we described in the previous sections, before the cleaning starts, each searching agent moves to its starting position so that there will be at least one searching agent in each node of the first column of the Mesh. The node at $P(0,0)$ has two searching agents and the *Synchronizer* S . At this point the cleaning process starts. The major cleaning steps are shown in Figure 4.2.

Figure 4.2: Edge-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer

As illustrated in Figure 4.2, after the initialization, *Synchronizer* S forces one searching agent at node $P(0,0)$ to move *EAST* and another one to move *SOUTHEAST* with S . After that, S traverses the edges in such a route that S goes back *NORTH* to clean the edges on the column, then goes *SOUTHWEST* to clean the edges on the diagonal, and then goes *EAST* to clean the edges on the row, finally

goes *WEST* to reach the searcher in the current row. S continues in the same way till all the searching agents reach the next column. At this point the procedure will continue, but this time with a bottom-up synchronization. This process stops if S reaches the last node $P(m-1, n-1)$.

A formal description of the strategy for edge-search which requires $(m+2)$ number of agents is provided in Algorithm $EdgeSearch_{OctaMesh} - Synr$ in Table 4.2.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize all the moves of the $(m+1)$ searching agents.

PROOF :

To prove the correctness of the Algorithm $EdgeSearch_{OctaMesh} - Synr$ presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

1. Base Case: the first column is clean.

Initially all the agents are located at the node $P(0, 0)$ and that node is clean.

During the initialization steps, *Synchronizer* S forces all the searching agents to move *SOUTH* along the first column and then guard their starting positions.

In other words, all the searching agents guard along the first column of the mesh after the initialization. It's obvious that the first column is clean and the initialization (or cleaning) result is contiguous and monotone. Then the cleaning steps start.

Algorithm *EdgeSearch_{OctaMesh} – Synr*
INPUT:

A contaminated $m \times n$ -Octagonal Mesh and $(m + 2)$ mobile agents

OUTPUT:

The given Octagonal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0;$

Position: $P(i, j);$

INITIALIZATION:

All $(m + 2)$ agents are gathered at the initial position $P(0, 0)$

The *Synchronizer* is labelled as S

for ($i = 0; i < m; i++$)
 searcher i is labelled as s_i

for ($i = m - 1; i \geq 0; i--$)
 S tells s_i move to $P(i, 0)$

CLEAN-UP

$j = 0$

while $j \leq n - 2$

$i = 0$

S tells s_m move to $P(i, j + 1)$ // move *EAST*

while $i \leq m - 2$

S tells s_i move to $P(i + 1, j + 1)$ with S // move *SOUTHEAST*

if $P(i + 1, j + 1)$ is guarded by a s

S moves to $P(i, j + 1)$ // move *NORTH*

S moves to $P(i + 1, j)$ // move *SOUTHWEST*

S moves to $P(i + 1, j + 1)$ // move *EAST*

S moves to $P(i + 1, j)$ // move *WEST*

$i++ = 1$

if $i == m - 1$

S tells s_i move to $P(i, j + 1)$ with S // move *EAST*

$j++ = 1$

if $j \neq n - 1$

S tells s_i move to $P(i, j + 1)$ // move *EAST*

while $i > 0$

S tells s_{i-1} move to $P(i - 1, j + 1)$ with S // move *NORTHEAST*

if $P(i - 1, j + 1)$ is guarded by a s

S moves to $P(i, j + 1)$ // move *SOUTH*

S moves to $P(i - 1, j)$ // move *NORTHWEST*

S moves to $P(i - 1, j + 1)$ // move *EAST*

S moves to $P(i - 1, j)$ // move *WEST*

$i-- = 1$

if $i == 0$

S tells s_m move to $P(i, j + 1)$ with S

$j++ = 1$

Table 4.2: Edge-Search in a $m \times n$ -Octagonal Mesh - With a Synchronizer

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line up in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. *Induction Step:*

As defined in our algorithm, at step $(i + 1)$, the *Synchronizer* S will force one agent to move to its *SOUTHEAST* (or *NORTHEAST*) neighboring node located at the adjacent next column to clean and guard that node. At the same time, S cleans the edges by moving *NORTH* or *SOUTH*, *SOUTHWEST* or *NORTHWEST*, and *EAST* along the edges on the column, diagonal and row. We want to show that the cleaned area is increased by one more node and three more edges, and no recontamination occurs.

By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, S will force the agent that guards the node located on the border row (either the first row or the last row) to move *EAST* to the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move. On the other hand, if all the agents guard two columns C_j and C_{j+1} at the cleaning step i then at the following cleaning step $(i + 1)$, there will be two

possibilities listed as follows.

- If S has a N_{SE} (or N_{NE}) neighboring node, S will continue to move along the diagonal and force the agent that guards the node located on column C_j to move *SOUTHEAST* (or *NORTHEAST*) to the next column C_{j+1} to clean and guard the *SOUTHEAST* (or *NORTHEAST*) neighboring node. Then S cleans the edges by moving *NORTH* or *SOUTH*, *SOUTHWEST* or *NORTHWEST*, and *EAST* along the edges on column, diagonal and row. All the other agents still guard their nodes and do not move.
- If S does not have a N_{SE} (or N_{NE}) neighboring node, S will force the agent that guards the node located on column C_j to move *EAST* the next column C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move.

In all cases, the node from which the agent has moved during step $(i + 1)$ is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured as S is able to synchronize all the moves of the other $(m + 1)$ searching agents. ■

Complexity

Theorem 11. *Algorithm $EdgeSearch_{OctaMesh} - Synr$ for the solution of the edge-search problem in a $m \times n$ -Octagonal Mesh uses $(m + 2)$ agents and cleans up the*

graph in $\frac{m^2+12mn-13m-6n+6}{2}$ steps and $(5mn - 4m - 4n + 3)$ time units.

PROOF :

The bound on the number of agents $s = m + 2$ is given by the above strategy described in Table 4.2.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows. During the *INITIALIZATION*, searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, all the $m + 1$ agents move $(n - 1)$ steps either along the diagonal or *EAST* for a total number of $(m + 1)(n - 1)$ steps. The total number of movements used by searching agents are $\frac{m(m-1)}{2} + (m + 1)(n - 1) = \frac{m^2+2mn-3m+2n-2}{2}$.

The number of steps consumed by the *Synchronizer S* can be calculated as follows. During the *CLEAN-UP*, S moves $2(m - 1)(n - 1)$ steps along the diagonal, $2(m - 1)(n - 1)$ steps *EAST* and *WEST*, $(n - 1)$ steps *EAST*, and $(m - 1)(n - 1)$ steps either *NORTH* or *SOUTH*. The total number of movements used by S are $5(m - 1)(n - 1) + (n - 1) = 5mn - 5m - 4n + 4$.

Therefore, the total number of searching steps for the edge-search in a $m \times n$ -Octagonal Mesh with a *Synchronizer* are $\frac{m^2+2mn-3m+2n-2}{2} + (5mn - 5m - 4n + 4) = \frac{m^2+12mn-13m-6n+6}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

During the *INITIALIZATION*, the m searching agents use $(m - 1)$ time units to be at their starting points. During the *CLEAN-UP*, to clean one single column (from

the first row to the last row), the *Synchronizer* S consumes $2(m-1)(n-1)$ time units to move along the two Diagonals, and $2(m-1)(n-1)$ time units to move either *EAST* or *WEST* along the edges on the rows, and another $(m-1)(n-1)$ time units to move either *SOUTH* or *NORTH* along the edges on the column. Also, S takes $(n-1)$ time units to move *EAST* along the edges on the border rows of the Mesh to the next neighboring column. And during that time, all the searching agents are informed by S to move *EAST* till reaching the last column. Hence, the total cleanup time is $(m-1) + 5(m-1)(n-1) + (n-1) = 5mn - 4m - 4n + 3$. ■

4.2 Searching Strategies - Agents with Visibility

In this model, agents have the visibility power, hence we do not need the *Synchronizer* to coordinate among the searching agents. In this searching strategy, we will use m searching agents to capture the intruder in a $m \times n$ -Octagonal Mesh. However, there are two diagonals in each cell in the Mesh, thus we use one extra *Helper* agent in the node-search to guard one of the extra diagonals and two extra *Helper* agents in the edge-search to guard and clean the edges in the Mesh.

When an agent decides to be the *Helper* agent, it will write “Helper Assigned” message on the *Whiteboard* to inform the other agents.

4.2.1 Node-Search in a $m \times n$ -Octagonal Mesh

1. Searching Strategy

Initially, all the $(m+1)$ agents are located at the node $P(0,0)$. At this point the

cleaning process starts. The major cleaning steps are captured in Figure 4.3

Figure 4.3: Node-Search in a $m \times n$ -Octagonal Mesh - Agent with Visibility

At the beginning of the cleaning algorithm, all agents wake up to be the searchers s . Each searcher will independently perform the following operations:

- s reads the *Whiteboard* on the current node.
- If the *Whiteboard* on node $P(0,0)$ is empty, s decides to be the *Helper* and writes “Helper Assigned” message on the *Whiteboard*.
- If the *Whiteboard* has a “CLEAN” message, s moves *SOUTH* to the next row. s will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* does not have a “CLEAN” message, s writes “CLEAN” message and guards the current node. Then s will perform one of the following four cases.

– CASE I:

s has a neighboring node along the diagonal in the *SOUTHEAST* direction and the current column number is an even number. If it can see its

neighboring nodes $N_E, N_{NE}, N_N, N_{NW}, N_W, N_{SW}, N_S$ except N_{SE} are all clean, then s moves *SOUTHEAST* to the next column.

– CASE II:

s does not have a neighboring node along the diagonal in the *SOUTHEAST* direction and the current column number is an even number. If it can see its neighboring nodes $N_E, N_{NE}, N_N, N_{NW}, N_W$ are all clean, then it moves *EAST* to the next column and then becomes a *Helper H*.

– CASE III:

s has a neighboring node along the diagonal in the *NORTHEAST* direction and the current column number is an odd number. If it can see its neighboring nodes $N_N, N_{NW}, N_W, N_{SW}, N_S, N_{SE}, N_E$ except N_{NE} are all clean, then s moves *NORTHEAST* to the next column.

– CASE IV:

s does not have a neighboring node along the diagonal in the *NORTHEAST* direction and the current column number is an odd number. If s can see its neighboring nodes $N_W, N_{SW}, N_S, N_{SE}, N_E$ are all clean, it moves *EAST* to the next column and then becomes *Helper H*.

At the same time, the *Helper agent H* performs the following operations:

- H moves *EAST* if it can see its incident neighboring nodes $N_N, N_{NW}, N_W, N_S, N_{SW}$ except N_E are all clean. Then H becomes a searcher s .
- Otherwise, H guards the current node.

The Mesh is cleaned when all the $(m + 1)$ agents have reached the last column.

The description of the algorithm is provided in Algorithm *NodeSearch_{OctaMesh} – Visb* in Table 4.3.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by all the $(m + 1)$ agents.

PROOF :

To prove the correctness of the Algorithm *NodeSearch_{OctaMesh} – Visb* presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

1. Base Case: the node $P(0, 0)$ is clean.

Initially, all the agents are located at the node $P(0, 0)$ and that node is clean.

The base case is true.

2. Assumption Step:

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line up in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. Induction Step:

Algorithm NodeSearchOctaMesh – Visb
INPUT:

A contaminated $m \times n$ -Octagonal Mesh and $(m + 1)$ mobile agents

OUTPUT:

The given Octagonal Mesh is cleaned.

VARIABLES:

Integers: $j := 0$;

Position: $P(0, 0)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All $(m + 1)$ agents are located at the node $P(0, 0)$

Agents Wake-Up to be the Searchers s

CLEAN-UP

$j = 0$

for each s

s reads messages from WB on Current Node

 if Current Node is node $P(0, 0)$

 if $AgentAssigned$ is not on WB

s becomes H and writes $AgentAssigned$ on WB

 while $CLEAN$ is on WB

s moves *SOUTH*

s reads messages from WB on Current Node

 if $CLEAN$ is not on WB

s writes $CLEAN$ on WB and guards Current Node

 while $\neq LastColumn$

 if s can see $N_E \& \& N_{NE} \& \& N_N \& \& N_{NW} \& \& N_W \& \& N_{SW} \& \& N_S$ are clean

 if s has a Neighboring Node in *SOUTHEAST* and $j \% 2 = 0$ then it moves *SOUTHEAST*

 else s moves *EAST* and becomes H

 if s can see $N_N \& \& N_{NW} \& \& N_W \& \& N_{SW} \& \& N_S \& \& N_{SE} \& \& N_E$ are clean

 if s has a Neighboring Node in *NORTHEAST* and $j \% 2 = 1$ then it moves *NORTHEAST*

 else s moves *EAST* and becomes H

$j++ = 1$

for H

 while $\neq LastColumn$

H guards Current Node

 if H can see $N_N \& \& N_{NW} \& \& N_W \& \& N_S \& \& N_{SW}$ are all clean

H moves *EAST*

H becomes s

Table 4.3: Node-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility

As defined in Algorithm *NodeSearch_{OctaMesh-Visb}*, during the cleanup, all the searching agents move *SOUTH* one by one to place themselves on the nodes of the first column such that there will be one searching agent guarding one of the nodes along the first column. Once a searching agent guards one node of the first column, it moves *SOUTHEAST* (or *NORTHEAST*) to the next adjacent column and guards the *SOUTHEAST* (or *NORTHEAST*) neighboring node if it can see its neighboring nodes except the N_{SE} are all clean. Meanwhile, the *Helper* agent moves *EAST* along the border row (either the first row or the last row) to clean and guard the *EAST* neighboring node.

By our algorithm, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, the *Helper* agent moves *EAST* along the border row (either the first row or the last row) to clean and guard the *EAST* neighboring node on column C_{j+1} . On the other hand, if all the agents guard two columns C_j and C_{j+1} at cleaning step i then at the following cleaning step $(i + 1)$, we would have two possibilities as follows.

- If an agent has a N_{SE} (or N_{NE}) neighboring node and it can see its neighboring nodes except the N_{SE} (or N_{NE}) are all clean, it moves *SOUTHEAST* (or *NORTHEAST*) to the next adjacent column C_{j+1} and guards the *SOUTHEAST* (or *NORTHEAST*) neighboring node. All the other m agents guard the nodes on C_j and do not move.
- If an agent does not have a N_{SE} (or N_{NE}) neighboring node and it can see its neighboring nodes are all clean, it moves *EAST* to the next column

C_{j+1} to clean and guard the *EAST* neighboring node. All the other agents still guard their nodes and do not move.

In all cases, the node from which the agent has moved during step $(i + 1)$, is now totally surrounded by clean or guarded nodes and hence no recontamination can occur.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured by the $(m + 1)$ agents. ■

Complexity

Theorem 12. *Algorithm $NodeSearch_{OctaMesh} - Visb$ for the solution of the node-search in a $m \times n$ -Octagonal Mesh uses $(m + 1)$ agents and cleans up the graph in $\frac{m^2 + 2mn - 3m + 2n - 2}{2}$ steps and $(mn - 2m + n - 2)$ time units.*

PROOF :

The bound on the number of agents $s = m + 1$ is given by the above strategy described in Table 4.3.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows. During the *CLEAN-UP*, searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. And then they have to repeat $(n - 1)$ number of loops such that in each single loop all agents move $(m - 1)$ steps along the diagonal edges and 1 step move *EAST* to the next column. So, the total number of steps

required by searching agents are $\frac{m(m-1)}{2} + m(n-1) = \frac{m^2+2mn-3m}{2}$.

The number of steps consumed by the *Helper* agent H can be calculated as follows. during the *CLEAN-UP*, H moves *EAST* to reach the next column in 1 step and it has to repeat $(n-1)$ times as defined in the algorithm. So, the total number of steps used by H are $(n-1)$.

Therefore, when agents with the visibility power, the total number of steps for the node-search on $m \times n$ -Octagonal Mesh are $\frac{m^2+2mn-3m}{2} + (n-1) = \frac{m^2+2mn-3m+2n-2}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

As defined in our algorithm, initially, at time 0, all the agents guard at the node $P(0,0)$. When the node-search starts, the awake searching agents move *SOUTH* such that there will be one agent in one row in every 1 time unit. The *Helper* agent H moves *EAST* to the second column to guard the node $P(0,1)$ at time unit 1. At time unit 2, the searching agent s_0 at the node $P(0,0)$ starts to move *SOUTHEAST* to the second column. At time unit 3, the searching agent s_1 at the node $P(1,0)$ starts to move *SOUTHEAST* to the second column, and so on. Then, at time unit m , the searching agent s_{m-1} at the node $P(m-1,0)$ starts to move *EAST* to the second column. Hence, agents use $(m+1)$ time units to move to the second column. This process is only for cleaning one single column. In the following, agents repeat the same process for a total of $(n-2)$ times till they all reach the last column. Therefore, the total cleanup time is $(m+1)(n-2) = mn - 2m + n - 2$. ■

4.2.2 Edge-Search in a $m \times n$ -Octagonal Mesh

1. Searching Strategy

The mesh has two diagonals in each cell in this type of topology, and we need to clean both nodes and edges in the graph. In order to capture the intruder, we use $(m + 1)$ agents as described in the previous section and one extra *Helper* agent to clean the edges on the diagonals and columns in the Mesh. Hence, the searching team size is $(m + 2)$.

We assume that initially, all the $(m + 2)$ agents are located at the node $P(0, 0)$. At this point the cleaning process starts. The major cleaning steps are captured in Figure 4.4

Figure 4.4: Edge-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility

At the beginning of the cleaning algorithm, all agents wake up to be the searchers s . Each searcher will independently perform the following operations:

- s reads messages from *Whiteboard* on the Current Node.
- If the *Whiteboard* on node $P(0, 0)$ is empty, s decides to be the *Helper* and

writes “Helper Assigned” message on the *Whiteboard*.

- If the *Whiteboard* has a “CLEAN” message, s moves *SOUTH* to the next row. s will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* does not have a “CLEAN” message, then s writes “CLEAN” message and guards the current node. If the current node is node $P(0,0)$ and that node has no “CLEAN” message, one of the searching agents writes “CLEAN” message and then moves *EAST* to guard node $P(0,1)$. Next, s will perform one of the following four cases.

– CASE I:

s has a neighboring node along the *SOUTHEAST* diagonal. s moves *SOUTHEAST* to the next column if it can see its neighboring nodes $N_E, N_{NE}, N_N, N_{NW}, N_W, N_{SW}, N_S$ except N_{SE} and its neighboring edges $E_E, E_{NE}, E_N, E_{NW}, E_W, E_{SW}, E_S$ except E_{SE} are all clean.

– CASE II:

s does not have a neighboring node along the *SOUTHEAST* diagonal. s moves *EAST* to the next column if it can see its neighboring nodes $N_E, N_{NE}, N_N, N_{NW}, N_W$ and its neighboring edges $E_E, E_{NE}, E_N, E_{NW}, E_W$ are all clean.

– CASE III:

If s can see there are 2 searchers in the last row, only one searcher moves

NORTH to the first row.

– CASE IV:

If s can see 2 searchers in the first row, only one searcher moves *EAST* to the next column.

At the same time, the *Helper H* performs the following operations:

- H moves *EAST* if it can see the current node is guarded.
- H moves *SOUTHWEST* if it can see the current node is guarded and there is a diagonal on the *SOUTHWEST* direction.
- If H can see the current node is guarded and the row is the last row, H moves *NORTH* to the first row.
- Otherwise, H guards the current node.

The Mesh is cleaned when all the $(m + 2)$ agents reach the last column.

A description of the algorithm is provided in Algorithm *EdgeSearch_{OctaMesh} – Visb* in Table 4.4.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by the $(m + 2)$ agents.

PROOF :

To prove the correctness of the Algorithm *EdgeSearch_{OctaMesh} – Visb* presented above, we need to prove that the cleaning result is contiguous and monotone. We prove it by induction.

Algorithm *EdgeSearch_{OctaMesh} - Visb*
INPUT:

A contaminated $m \times n$ -Octagonal Mesh and $(m + 2)$ mobile agents

OUTPUT:

The given Octagonal Mesh is cleaned.

VARIABLES:

Position: $P(0,0)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All $(m + 2)$ agents are located at node $P(0,0)$

Agents *Wake-Up* to be the *Searchers s*

CLEAN-UP

for each s

```

     $s$  reads messages from  $WB$  on Current Node
    if Current Node is node  $P(0,0)$ 
        if HelperAssigned is not on  $WB$ 
             $s$  becomes  $H$  and writes HelperAssigned on  $WB$ 
    while CLEAN is on  $WB$ 
         $s$  moves SOUTH to next row
         $s$  reads messages from  $WB$  on Current Node
    if CLEAN is not on  $WB$ 
        if Current Node is the node at  $P(0,0)$  and CLEAN' is not on  $WB$ 
             $s$  writes CLEAN' on  $WB$  and moves EAST to guard node  $P(0,1)$ 
        else
             $s$  writes CLEAN on  $WB$  and guards Current Node
    if  $s$  guards its starting point
        while  $\neq$  LastColumn
            if  $s$  can see Neighboring Nodes and Edges are all clean except  $N_{SE}$  &&  $E_{SE}$ 
                if  $s$  has a Neighboring Node in SOUTHEAST then it moves SOUTHEAST
            else  $s$  moves EAST
            if Current Row = Last Row
                if 2  $s$  at Current Node and  $N_N$  is guarded then one  $s$  moves NORTH to First Row
            if Current Row = First Row
                if 2  $s$  at Current Node then one  $s$  moves EAST

```

for H

```

    while  $\neq$  LastColumn
         $H$  guards Current Node
        if  $H$  can see Current Node is guarded
            if  $H$  has a uncleaned Neighboring Node in SOUTHWEST
                 $H$  moves SOUTHWEST
             $H$  moves EAST
        if  $H$  reaches Last Row and Current Node and  $N_N$  are guarded
             $H$  moves NORTH to First Row

```

Table 4.4: Edge-Search in a $m \times n$ -Octagonal Mesh - Agents with Visibility

1. *Base Case: the node $P(0, 0)$ is clean.*

Initially, all the agents are located at the node $P(0, 0)$ and that node is clean.

The base case is true.

2. *Assumption Step:*

Let's assume that at step i ($i > 0$), the cleaning result is contiguous and monotone. In other words, at step i , all the clean nodes are connected and all the agents guard the contiguous border of the clean nodes, thus no recontamination occurs. The clean border may line up in one single column C_j ($0 < j < n$) or two consecutive columns C_j and C_{j+1} ($0 < j < n - 1$) as the cleaning result defined by our algorithm.

3. *Induction Step:*

As defined in Algorithm *EdgeSearch_{OctaMesh-Visb}*, during the cleanup, all the searching agents move *SOUTH* one by one to place themselves on the nodes of the first column such that there will be one searching agent guarding one of the nodes along the first column. Once the searching agents line up on the first column, both the searching agents and the *Helper* agent H move along the graph to clean both the nodes and edges in the Mesh.

By our algorithm, at step $(i + 1)$, if all the agents guard one single column C_j at the cleaning step i then at the following cleaning step $(i + 1)$, H moves *NORTH* to the first row to clean the edges on column C_{j+1} , all the other searching agents still guard their nodes and do not move. On the other hand, if all the agents guard two columns C_j and C_{j+1} at cleaning step i then at the following cleaning

step $(i + 1)$, there will be three possibilities that are listed as follows.

- If an agent sees its neighboring nodes except N_{SE} are all clean or guarded, then it moves *SOUTHEAST* to the next adjacent column C_{j+1} .
- If an agent s sees a “CLEAN” message on the *Whiteboard* at the current node, then it knows this node is already cleaned by another searcher s' . s will move *SOUTH* to the next adjacent row.
- If H sees the current node at C_{j+1} is guarded then H moves *SOUTHWEST* to C_j and then moves *EAST* to C_{j+1} .
- Otherwise, searchers guard the current node and do not move.

The clean border is contiguous after the cleaning step $(i + 1)$. In both cases, the node and edges from which the agent has moved during step $(i + 1)$, is now totally surrounded by clean or guarded nodes and thus no recontamination occurs.

By the above proof, we have proved that the cleaning result is contiguous and monotone. Therefore, we proved that the intruders are captured by all the $(m + 2)$ agents. ■

Complexity

Theorem 13. *Algorithm $EdgeSearch_{OctaMesh} - Visb$ for the solution of the edge-search in a $m \times n$ -Octagonal Mesh uses $(m + 2)$ agents and cleans up the graph in $\frac{m^2 + 10mn - 11m - 4n + 2}{2}$ steps and $(3mn - 3m - 2n + 2)$ time units.*

PROOF :

The bound on the number of agents $s = m + 2$ is given by the above strategy described in Table 4.4.

The cleanup steps consumed by searching agents and help Agent in the edge-search are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows. During the *CLEAN-UP*, searching agents use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. And then they have to repeat $(n - 1)$ number of loops such that in each single loop 2 agents move 2 steps *EAST*, one agent moves $(m - 1)$ steps *NORTH* from the last row to the first row, and all the remaining $(m - 1)$ searching agents move $(m - 1)$ steps on the diagonal. So, the total number of steps required by searching agents are $\frac{m(m-1)}{2} + (2 + m - 1 + m - 1)(n - 1) = \frac{m^2 + 4mn - 5m}{2}$.

The number of steps consumed by the *Helper H* can be calculated as follows. During the *CLEAN-UP*, H moves $(m - 1)(n - 1)$ steps on the diagonal, $m(n - 1)$ steps *EAST*, and $(m - 1)(n - 1)$ steps *NORTH*. The total number of steps used by H are $2(m - 1)(n - 1) + m(n - 1) = 3mn - 3m - 2n + 2$.

Therefore, when agents with the visibility power, the total number of steps for the edge-search on a $m \times n$ -Octagonal Mesh are $\frac{m^2 + 4mn - 5m}{2} + (3mn - 3m - 2n + 2) = \frac{m^2 + 10mn - 11m - 4n + 2}{2}$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

As defined in our algorithm, during the cleanup, to clean one single column, the m searching agents clean the edges on rows and the *Helper agent H* cleans the edges on

the diagonals and columns. Once the edge-search starts, the searching agents move from the first column to the second column using m time units, and H and searching agents move along the two diagonals in $(m - 1)$ time units, and also H moves from the last row to the first row to clean edges on the column in another $(m - 1)$ time units. Hence, agents consume $m + 2(m - 1)$ time units to clean from one column to the next adjacent column. This process will be repeated $(n - 1)$ times. Therefore, the total time is $(3m - 2)(n - 1) = 3mn - 3m - 2n + 2$. ■

4.3 Summary of Searching Results on Different Models in a $m \times n$ -Octagonal Mesh

By the previous calculations, the complexities of our searching algorithms in a $m \times n$ -Octagonal Mesh are summarized in the following table.

Search in a $m \times n$ -Octagonal Mesh			
Search Strategies	Number Of	Models	
		<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
Node Search	Searchers	$m + 2$	$m + 1$
	Steps	$\frac{m^2 + 6mn - 7m}{2}$	$\frac{m^2 + 2mn - 3m + 2n - 2}{2}$
	Time	$2mn - m - n$	$mn - 2m + n - 2$
Edge Search	Searchers	$m + 2$	$m + 2$
	Steps	$\frac{m^2 + 12mn - 13m - 6n + 6}{2}$	$\frac{m^2 + 10mn - 11m - 4n + 2}{2}$
	Time	$5mn - 4m - 4n + 3$	$3mn - 3m - 2n + 2$

From the above summarized results, we can see that the complexities of searching agents, steps and time are all similar in our algorithms in a $m \times n$ -Octagonal Mesh. This is because that there are two diagonals in a $m \times n$ -Octagonal Mesh, the extra edges in each diagonal allow more chance for the intruder to hide or escape. Therefore, regardless of node-search or edge-search, we have to repeat almost the same traversal path along the Mesh nodes and links to perform the contiguous search. Moreover, the visibility power of searching agents can not help decrease the searching steps and time due to the two diagonals in the Mesh. This is because searching agents need to guard the nodes on a column and wait the extra edges on the two diagonals are cleaned by a *Helper* agent.

Chapter 5

Node/Edge-Search in a $m \times n$ -Toroidal Mesh

5.1 Searching Strategies - With a Synchronizer

Interestingly the technique of node/edge-search in a $m \times n$ -Toroidal Mesh is almost identical to the strategy used by the node/edge-search with a *Synchronizer* in a $m \times n$ -Mesh, where m agents are as the searchers, and one extra agent is designated as the *Synchronizer* to synchronize all the other searching agents to search or move in a Mesh. However, for this type of Mesh, we deploy extra m agents to be guards. This is because the nodes on the first row and the last row are connected by direct links, and the nodes on the first column and the last column are also connected by direct links. In order to protect the cleaned areas, agents are needed to guard nodes $P(0,0)$, $P(1,0)$, $\dots$, $P(m-2,0)$, $P(m-1,0)$ and remain there till the Mesh is cleaned. In

the following sections, we will show that the searching team size for a $m \times n$ -Toroidal Mesh is $(2m + 1)$ agents in total.

5.1.1 Node-Search in a $m \times n$ -Toroidal Mesh

1. Searching Strategy

The major cleaning steps are similar to those listed in the four cases shown in Figure 2.3. The only difference is that there will be two agents in each node of the first column of the Mesh after the initialization.

The formal description of the Algorithm $NodeSearch_{ToroMesh} - Synr$ is provided in Table 5.1.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as S is able to synchronize the moves of all the other $2m$ agents.

PROOF :

As defined in the Algorithm $NodeSearch_{ToroMesh} - Synr$, the searching steps are almost identical to those defined by the strategy in the Algorithm $Search_{Mesh} - Synr$, except that there are m guards guarding the first column of the Toroidal Mesh till the Mesh is cleaned. We proved that the Algorithm $Search_{Mesh} - Synr$ is contiguous and monotone in chapter 2. And the extra m guards do not create any recontamination during the cleanup. Therefore, we conclude that in node-search of the Algorithm $NodeSearch_{ToroMesh} - Synr$, the intruder is captured as S is able to synchronize all

ALGORITHM *NodeSearch_{ToroMesh} - Synr*
INPUT:

A contaminated $m \times n$ -Toroidal Mesh and $(2m + 1)$ mobile agents

OUTPUT:

The given Toroidal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

INITIALIZATION:

All $(2m + 1)$ agents are located at node $P(0, 0)$

The *Synchronizer* is labelled as S

```
for ( $i = m - 1; i \geq 0; i--$ )
    searcher  $i$  is labelled as  $s_i$ 
    guard  $i$  is labelled as  $g_i$ 
     $S$  tells  $g_i$  move to and guard node  $P(i, 0)$ 
     $S$  tells  $s_i$  move to  $P(i, 0)$ 
```

CLEAN-UP

$j = 0$

```
while  $j < n - 1$ 
     $i = 0$ 
    while  $i \leq m - 1$ 
         $S$  moves to  $P(i, j)$ 
         $S$  tells  $s_i$  moves to  $P(i, j + 1)$ 
         $i++ = 1$ 
     $j++ = 1$ 
     $S$  moves to  $P(i, j)$ 
    if ( $j \neq n - 1$ )
        while  $i \geq 0$ 
             $S$  moves to  $P(i, j)$ 
             $S$  tells  $s_i$  moves to  $P(i, j + 1)$ 
             $i-- = 1$ 
     $j++ = 1$ 
```

Table 5.1: Node-Search in a $m \times n$ -Toroidal Mesh - With a Synchronizer

the moves of the other $2m$ agents. ■

Complexity

Theorem 14. *Algorithm $NodeSearch_{ToroMesh} - Synr$ for the solution of the node-search in a $m \times n$ -Toroidal Mesh uses $(2m + 1)$ agents and cleans up the graph in $(m^2 + 2mn - 3m - 1)$ steps and $(mn - 2)$ time units.*

PROOF :

The bound on the number of agents $s = 2m + 1$ is given by the above strategy described in Table 5.1.

To compute the complexity, let g be guards, g_i ($0 \leq i < m$) be one of the guards.

The cleanup steps are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows.

During the *INITIALIZATION*, the searching agents require $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, all m searching agents move $(n - 1)$ steps *EAST* for a total number of $m(n - 1)$ steps. So, the total number of steps used by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

The number of steps consumed by the *Synchronizer* S can be calculated as follows.

During the *CLEAN-UP*, S moves $(n - 2)$ steps *EAST* and $(m - 1)(n - 1)$ steps on columns. The total number of steps required by S are $(n - 2) + (m - 1)(n - 1) = mn - m - 1$.

The number of steps consumed by guards g can be calculated as follows. During the *INITIALIZATION*, guards need $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps. The total

number of steps performed by guards are $\frac{m(m-1)}{2}$.

Therefore, the total number of steps for the node-search on a $m \times n$ -Toroidal Mesh with a *Synchronizer* are $\frac{m^2+2mn-3m}{2} + (mn - m - 1) + \frac{m(m-1)}{2} = m^2 + 2mn - 3m - 1$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

During the *INITIALIZATION*, all agents (both the searching agents s and the guarding agents g) need $(m - 1)$ time units to place themselves on the first column. During the *CLEAN-UP*, the *Synchronizer* S uses $(m - 1)$ time units to move *SOUTH* or *NORTH* along the $(n - 1)$ columns in Mesh. At the same time, all the searching agents are forced by S to move *EAST*. Also, S uses $(n - 2)$ time units to move *EAST* to the next column after the searching agents have moved to and lined up on the next adjacent column. Therefore, the total cleanup time for the node-search is $(m - 1) + (m - 1)(n - 1) + (n - 2) = mn - 2$. ■

5.1.2 Edge-Search in a $m \times n$ -Toroidal Mesh

1. Searching Strategy

Same as the node-search described in the previous sections, we require $(2m + 1)$ agents in total. Before the cleaning starts, each searching agents and guard agents move to their starting position so that there will be two agents in each node of the first column in the Mesh, and the *Synchronizer* is in the uppermost node of the first column. At this point the cleaning process starts. The major cleaning steps are similar to those of the node-search. However, some extra edges connecting the first row and last row have to be cleaned by the *Synchronizer*.

ALGORITHM *EdgeSearch_{ToroMesh} – Synr***INPUT:**

A contaminated $m \times n$ -Toroidal Mesh and $(2m + 1)$ mobile agents

OUTPUT:

The given Toroidal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

INITIALIZATION:

All $(2m + 1)$ agents are gathered at the initial point $P(0, 0)$

The *Synchronizer* is labelled as S

for ($i = m - 1; i \geq 0; i--$)

 searcher i is labelled as s_i

 guard i is labelled as g_i

S tells g_i move to and guard node $P(i, 0)$

S tells s_i move to $P(i, 0)$

CLEAN-UP

$i = 0, j = 0$

S moves to $P(m - 1, 0)$ // to clean the extra one edge

while $j \leq n - 1$

while $i \geq 0$

S tells s_i moves to $P(i, (j + 1) \% n)$

$i--$

S moves to $P(i, j)$

$j++$

if ($j \neq 0$)

S moves to $P(i, j)$ // to next column

S moves to $P(m - 1, j)$ // to clean the edge that between the first row and the last row

Table 5.2: Edge-Search in a $m \times n$ -Toroidal Mesh - With a Synchronizer

The formal description of the Algorithm *EdgeSearch_{ToroMesh} – Synr* is provided in Table 5.2.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured as the *Synchronizer* S is able to synchronize the moves of all the other $2m$ agents.

PROOF :

As defined in our algorithm, the searching steps are similar to those defined by the strategy in Algorithm $Search_{Mesh} - Synr$, but with some extra moves to clean those edges connecting the nodes from the first row to the last row, and also with m guards guard the first column of the Toroidal Mesh till the Mesh is cleaned. We proved that the Algorithm $Search_{Mesh} - Synr$ is contiguous and monotone in chapter 2. And for those extra movements in Algorithm $EdgeSearch_{ToroMesh} - Synr$, those steps are achieved by the *Synchronizer* S while other searching agents guard the nodes along the column. Hence, those extra steps are contiguous and monotone. Also the extra m guards do not create any recontamination during the cleanup. Therefore, we conclude that in edge-search of the Algorithm $EdgeSearch_{ToroMesh} - Synr$, the intruder is captured as S is able to synchronize all the moves of the other $2m$ agents.

■

Complexity

Theorem 15. *Algorithm $Edge - Search_{ToroMesh} - Synr$ for the solution of the edge-search problem in a $m \times n$ -Toroidal Mesh uses $(2m + 1)$ agents and cleans up the graph in $(m^2 + 2mn - m + n - 1)$ steps and $(mn + m + n - 2)$ time units.*

PROOF :

The bound on the number of agents $s = 2m + 1$ is given by the above strategy described in Table 5.2.

The cleanup steps consumed by the searching agents and the *Synchronizer* are calculated as follows.

The number of steps consumed by the searching agents s can be calculated as

follows. During the *INITIALIZATION*, searching agents need $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, all the m searching agents move n steps *EAST* for a total number of mn steps. So, the total number of steps required by searching agents are $\frac{m(m-1)}{2} + mn = \frac{m^2+2mn-m}{2}$.

The number of steps consumed by the *Synchronizer S* can be calculated as follows. During the *CLEAN-UP*, S moves $(n - 1)$ steps *EAST*, mn moves *NORTH* along the n columns. Hence, totally S requires $(n - 1) + mn = mn + n - 1$ number of moves.

The number of steps consumed by guards g can be calculated as follows. During the *INITIALIZATION*, guards need $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps. The total number of moves used by guards are $\frac{m(m-1)}{2}$.

Therefore, the total number of steps for the edge-search on a $m \times n$ -Toroidal Mesh with a *Synchronizer* are $\frac{m^2+2mn-m}{2} + (mn + n - 1) + \frac{m(m-1)}{2} = m^2 + 2mn - m + n - 1$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

During the *INITIALIZATION*, all the agents (both the searching agent s and the guarding agent g) use $(m - 1)$ time units to place themselves on the first column. During the *CLEAN-UP*, the *Synchronizer S* uses m time units to move along each n columns, and also needs another $(n - 1)$ time units to move *EAST*. At the same time, all the searching agents move *EAST* to the next column. So, the total cleanup time for the edge-search is $(m - 1) + mn + (n - 1) = mn + m + n - 2$. ■

5.2 Searching Strategies - Agents with Visibility

As described in the previous chapters of Asynchronous Model and agents with visibility power, the *Synchronizer* is not necessary in this model. But in this type of Mesh, the nodes on the first row and the last row are connected by direct links and the nodes on the first column and the last column are also connected by direct links. Thus, we use $2m$ agents to guard the nodes on the first column of the Mesh to avoid re-contamination, where m agents are the searching agents and the other extra m agents are the guards. Moreover, in the edge-search, we need one extra *Helper* agent to clean some edges in the graph.

Agents communicate with each other by using *Whiteboard* associated with each node in the Mesh. In this model, the *Whiteboard* may contain two messages, namely the “CLEAN” message and the “GUARD” message. We assume the *Whiteboard* is empty (or no message) initially. When an agent arrives at a node, it reads messages from the *Whiteboard*. If the *Whiteboard* has no “CLEAN” message then the agent writes “CLEAN” message on the *Whiteboard* and starts to clean that node. If the *Whiteboard* has no “GUARD” message then the agent writes “GUARD” message on the *Whiteboard* and guards that node. Otherwise, the agent will move *SOUTH* to the next row if the *Whiteboard* has both “CLEAN” and “GUARD” messages.

5.2.1 Node-Search in a $m \times n$ -Toroidal Mesh

1. Searching Strategy

Initially, all the $2m$ agents are located at node $P(0, 0)$. At this point the cleaning

steps start. The major cleaning steps are similar to the steps described in Figure 2.4.

At the beginning of the cleaning algorithm, all agents wake up to be the searchers

s. Each searcher independently performs the following operations:

- *s* reads messages from the *Whiteboard* on the current node.
- If the *Whiteboard* has both “CLEAN” and “GUARD” messages, *s* moves *SOUTH* to the next row. *s* will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* has no “GUARD” message, *s* writes “GUARD” on the *Whiteboard*. *s* becomes a guard *g* and guards the current node till the Mesh is cleaned.
- If the *Whiteboard* has no “CLEAN” message, *s* writes “CLEAN” on the *Whiteboard*. *s* waits at the current node till it can see its neighboring nodes N_N , N_W , N_S except N_E are all clean, then *s* moves *EAST* to the next column.

The formal description of the Algorithm $NodeSearch_{ToroMesh} - Visb$ is provided in Table 5.3.

2. Correctness and Analysis

Correctness

Let us prove that the intruder is captured by all the $2m$ agents.

PROOF :

As defined in our algorithm, this strategy is almost identical to the strategy defined in Algorithm $NodeSearch_{Mesh} - Visb$ except that there are m guards guard the first column of the Toroidal Mesh till the Mesh is cleaned. We proved that the Algorithm

ALGORITHM *NodeSearchToroMesh - Visb*
INPUT:

A contaminated $m \times n$ -Toroidal Mesh and $2m$ mobile agents

OUTPUT:

The given Toroidal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0$;

Position: $P(i, j)$;

Whiteboard: $WB = \emptyset$

INITIALIZATION:

All $2m$ agents are located at the node $P(0, 0)$

Agents *Wake-Up* to be the *Searchers* s

CLEAN-UP

for each s

$i = 0, j = 0$ // i, j are local variables for agent

s reads messages from WB at node $P(i, j)$

while *CLEAN* and *GUARD* are on WB

$i++ = 1$

s moves to node $P(i, j)$ // move *SOUTH*

s reads messages from WB at node $P(i, j)$

if *GUARD* is not on WB

s becomes g

g writes *GUARD* on WB

g guards node $P(i, j)$ till the Mesh is cleaned.

else if *CLEAN* is not on WB

s writes *CLEAN* on WB

while $(j < n - 1)$

s waits at $P(i, j)$

if s can see N_N && N_W && N_S are all clean

$j++ = 1$

s moves to node $P(i, j)$ // move *EAST*

Table 5.3: Node-Search in a $m \times n$ -Toroidal Mesh - Agents with Visibility

$NodeSearch_{Mesh} - Visb$ is contiguous and monotone in chapter 2. And the extra m guards do not create any recontamination during the cleanup. Therefore, we conclude that in node-search of the Algorithm $NodeSearch_{ToroMesh} - Visb$, the intruder is captured by all the moves of the $2m$ agents. ■

Complexity

Theorem 16. *Algorithm $NodeSearch_{ToroMesh} - Visb$ for the solution of node-search in a $m \times n$ -Toroidal Mesh uses $2m$ agents and cleans up the graph in $(m^2 + mn - 2m)$ steps and $(m + n - 2)$ time units.*

PROOF :

The bound on the number of agents $s = 2m$ is given by the above strategy described in Table 5.3.

The cleanup steps consumed by the searchers and guards are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows. During the *INITIALIZATION*, searching agents use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. During the *CLEAN-UP*, all the m searching agents move $(n - 1)$ steps *EAST* for a total of $m(n - 1)$ number of steps. So, the total number of steps required by searching agents are $\frac{m(m-1)}{2} + m(n - 1) = \frac{m^2 + 2mn - 3m}{2}$.

The number of steps consumed by guards g can be calculated as follows. During the *INITIALIZATION*, guards use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to place themselves on the first column. The total number of moves used by guards are $\frac{m(m-1)}{2}$.

Therefore, when agents with the visibility power, the total number of steps for the node-search on a $m \times n$ -Toroidal Mesh are $\frac{m^2 + 2mn - 3m}{2} + \frac{m(m-1)}{2} = m^2 + mn - 2m$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

The algorithm steps are similar to the steps described in Figure 2.4. From that figure, we can see that all the searching agents and the guarding agents line up on the first column in $(m - 1)$ time units. After that, all the searching agents move *EAST* to the next adjacent column and clean the nodes. There are $(n - 1)$ columns need to be cleaned and the searching agents move *EAST* to the next column in 1 time unit, hence, they use another $(n - 1)$ time units to reach the last column. Therefore, the total clean up time for the node-search is $(m - 1) + (n - 1) = m + n - 2$. ■

5.2.2 Edge-Search in a $m \times n$ -Toroidal Mesh

1. Searching Strategy

In this model, we need to clean both nodes and edges in the Mesh. In order to capture the intruder, we have to use $2m$ agents and one extra *Helper* agent to clean the edges on the columns in the graph.

Initially, all the $(2m + 1)$ agents are located at the node $P(0, 0)$. At this point the cleaning process starts. The major cleaning steps are similar to the steps described in Figure 2.5.

At the beginning of the cleaning algorithm, all searchers wake up to be the searcher s . Each searcher s will perform the following operations:

- s reads messages from the *Whiteboard* on the current node.
- If the *Whiteboard* on node $P(0, 0)$ is empty, s decides to be the *Helper* and write “Helper Assigned” message on the *Whiteboard*.

- If the *Whiteboard* has both “CLEAN” and “GUARD” messages, s moves *SOUTH* to the next row. s will contiguously move *SOUTH* till it reaches a node on which the *Whiteboard* is empty.
- If the *Whiteboard* has no “GUARD” message, s writes “GUARD” on the *Whiteboard*. s becomes a guard g and guards the current node till the Mesh is cleaned.
- If the *Whiteboard* has no “CLEAN” message, s writes “CLEAN” on the *Whiteboard*. s waits at the current node till it can see its neighboring nodes N_N, N_W, N_S except N_E and its neighboring edges E_N, E_W, E_S except E_E are all clean, then s moves *EAST* to the next column.

The *Helper H* will first clean the one extra edge that connects the node at $P(0, 0)$ and the node at $P(m-1, 0)$ if node $P(0, 0)$ is guarded. Then H repeats the following operations till the node at $P(m-1, n-1)$ is reached:

- moves *EAST* to the next column if the current node is clean.
- H moves *SOUTH* if it can see the current node is clean.
- If H reaches the last row, it moves *NORTH* back to the first row and it moves *EAST* to the next column.
- Otherwise, H waits at the current node.

The formal description of the Algorithm $EdgeSearch_{ToroMesh} - Visb$ is provided in Table 5.4.

2. Correctness and Analysis

ALGORITHM *EdgeSearchToroMesh - Visb*
INPUT:

A contaminated $m \times n$ -Toroidal Mesh and $(2m + 1)$ mobile agents

OUTPUT:

The given Toroidal Mesh is cleaned.

VARIABLES:

Integers: $i := 0, j := 0, x := 0, y := 0$;

Position: $P(i, j), P(x, y)$;

Whiteboard: $WB = \emptyset$;

INITIALIZATION:

All $(2m + 1)$ agents are located at node $P(0, 0)$

Agents *Wake-Up* to be the *Searchers s*

CLEAN-UP

for each s

```

     $i = 0, j = 0$  //  $i, j$  are local variables for agent
     $s$  reads messages from  $WB$  at the node  $P(i, j)$ 
    if Current Node is node  $P(0, 0)$ 
        if HelperAssigned is not on  $WB$ 
             $s$  becomes  $H$  and writes HelperAssigned on  $WB$ 
        while CLEAN and GUARD are on  $WB$ 
             $i++ = 1$ ,  $s$  moves to node  $P(i, j)$  // move SOUTH
             $s$  reads messages from  $WB$  at the node  $P(i, j)$ 
            if GUARD is not on  $WB$ 
                 $s$  becomes  $g$ 
                 $g$  writes GUARD on  $WB$ 
                 $g$  guards node  $P(i, j)$  till the Mesh is cleaned.
            else if CLEAN is not on  $WB$ 
                 $s$  writes CLEAN on  $WB$ 
                while ( $j < n$ )
                     $s$  waits at  $P(i, j)$ 
                    if  $s$  can see its neighboring nodes  $N_N$  &&  $N_W$  &&  $N_S$ 
                       AND neighboring edges  $E_N$  &&  $E_W$  &&  $E_S$  are all clean
                         $j++ = 1$ ,  $s$  moves to node  $P(i, j \% n)$  // move EAST

```

for H

```

     $H$  moves to node  $P(m - 1, 0)$ 
     $x = m - 1, y = 0$  //  $x, y$  are local variables for  $H$ 
    if  $H$  can see node  $P(x, y)$  is clean
         $H$  moves to node  $P(x, y + 1)$ 
    while ( $y < n$ )
         $H$  waits at node  $P(x, y)$ 
        if  $H$  can see node  $P(x, y)$  is clean
             $x-- = 1$ ,  $H$  moves to node  $P(x \% m, y)$ 
        if  $x == 0$ 
             $x = m - 1$ ,  $H$  moves to node  $P(x, y)$  // move to the last row
             $y++ = 1$ ,  $H$  moves to node  $P(x, y)$  // move EAST

```

Table 5.4: Edge-Search in a $m \times n$ -Toroidal Mesh - Agents with Visibility

Correctness

Let us prove that the intruder is captured by all the $(2m + 1)$ agents.

PROOF :

As defined in our algorithm, the edge-search strategy is similar to that defined in Algorithm $EdgeSearch_{Mesh} - Visb$, but with some extra moves to clean those edges connecting the nodes from the first row to the last row, and also with m guards to guard the first column of the Toroidal Mesh till the Mesh is cleaned. We proved that the Algorithm $EdgeSearch_{Mesh} - Visb$ is contiguous and monotone in chapter 2. And for those extra movements in the Algorithm $EdgeSearch_{ToroMesh} - Visb$, those steps are achieved by the *Helper Agent H* while other searching agents guard the nodes along the column. Hence, those extra steps are contiguous and monotone. Also the extra m guards do not create any recontamination during the cleanup. Therefore, we conclude that in edge-search of the Algorithm $EdgeSearch_{ToroMesh} - Visb$, the intruder is captured by all the moves of the $(2m + 1)$ agents. ■

Complexity

Theorem 17. *Algorithm $EdgeSearch_{ToroMesh} - Visb$ for the solution of the edge-search problem in a $m \times n$ -Toroidal Mesh uses $(2m + 1)$ agents and cleans up the graph in $(m^2 + 2mn - 2m + n)$ steps and $(mn + n - 1)$ time units.*

PROOF :

The bound on the number of agents $s = 2m + 1$ is given by the above strategy described in Table 5.4.

The cleanup steps by the agents are calculated as follows.

The number of steps consumed by searching agents s can be calculated as follows. During the *CLEAN-UP*, the m searching agents use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps to move *SOUTH* to cover the nodes in the first column. Then, they move n steps *EAST* for a total number of mn steps. The total number of moves required by searching agents are $\frac{m(m-1)}{2} + mn = \frac{m^2+2mn-m}{2}$.

The number of steps consumed by *Helper H* can be calculated as follows. During the *CLEAN-UP*, H moves 1 step to the $P(m-1, 0)$. That means H moves 1 step on the first column, then $n-1$ steps *EAST*, and $m(n-1)$ steps on the column. i.e., totally H requires $1 + (n-1) + m(n-1) = mn - m + n$ number of moves.

The number of steps consumed by guards g can be calculated as follows. During the *INITIALIZATION*, guards use $1 + 2 + \dots + m - 1 = \frac{m(m-1)}{2}$ steps. The number of steps required by guards are $\frac{m(m-1)}{2}$.

Therefore, the total number of steps for the edge-search on $m \times n$ -Toroidal Mesh and agents with visibility are $\frac{m^2+2mn-m}{2} + (mn-m+n) + \frac{m(m-1)}{2} = m^2+2mn-2m+n$.

Based on the ideal time assumption, the cleanup time is calculated as follows.

This algorithm steps are illustrated in Figure 2.5. During the *CLEAN-UP*, all the searching agents and the guarding agents line up on the first column in $(m-1)$ time units. Then, the *Helper agent H* moves from the first row to the last row in 1 time unit. And then, H uses $(n-1)$ time units to move *EAST* to the next column, and m time units to move and clean edges on each column. And there are $(n-1)$ columns need to be cleaned. At the same time, all searchers move *EAST* to the next adjacent column and clean the nodes on it. Therefore, the total cleanup time is $(m-1) + 1$

$$+ (n - 1) + m(n - 1) = (mn + n - 1). \blacksquare$$

5.3 Summary of Searching Results on Different Models in a $m \times n$ -Toroidal Mesh

By the previous calculations, the complexities of our searching algorithms in a $m \times n$ -Toroidal Mesh are summarized in the following table.

Search in a $m \times n$ -Toroidal Mesh			
Search Strategies	Number Of	Models	
		<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
Node Search	Searchers	$2m + 1$	$2m$
	Steps	$m^2 + 2mn - 3m - 1$	$m^2 + mn - 2m$
	Time	$mn - 2$	$m + n - 2$
Edge Search	Searchers	$2m + 1$	$2m + 1$
	Steps	$m^2 + 2mn - 2m + n$	$m^2 + 2mn - 2m + n$
	Time	$mn + m + n - 1$	$mn + n - 1$

As summarized in the above table, the complexities of searching steps and time consumed in our algorithms are similar to the results in a $m \times n$ -Mesh. However, in a $m \times n$ -Toroidal Mesh, one interesting result in our algorithms is that we use $2m$ or $(2m + 1)$ searchers to capture the intruder. In the Toroidal Mesh, each single row or each single column is in a loop (or a ring) topology, and all the rows and columns are connected together. If the searching agent moves along a single row, the intruder

can always escape along the loop. Therefore, we need to use one extra agent guards one node to break the circle topology of each row. There are m rows in the Mesh, the extra m agents used in our algorithms are designated to be guards guarding the nodes on the first column in Mesh.

Chapter 6

Conclusions

6.1 Results

In this thesis, we have studied the problem of intruder capture by mobile agents in Mesh topology networks. Our objective was to use the *minimal* number of searching agents to contiguously capture the intruder in a Mesh, and we have shown that our algorithms have achieved this goal in order of magnitude.

We only considered the *Asynchronous* model in this thesis. Based on the *Asynchronous* model, we proposed 14 contiguous node/edge-search algorithms as described in the previous chapters. Our algorithms were conducted on four variant types of *Mesh* topologies, namely $m \times n$ -Mesh, $m \times n$ -Hexagonal Mesh, $m \times n$ -Octagonal Mesh, and $m \times n$ -Toroidal Mesh.

Our first conclusion is that the contiguous node-search problem in a $m \times n$ -Mesh ($m \leq n$) can be solved by employing $O(m)$ searchers, which is optimal.

In addition, we proved the correctness of our algorithms and a detailed analysis is also provided for each algorithm. In the following, our research results on contiguous node/edge-search are summarized and compared.

- ***Contiguous Node-Search***

- *Number of Searching Agents*

In our algorithms $NodeSearch_{Mesh} - Visb$ and $NodeSearch_{HexaMesh} - Visb$, when agents have visibility power, m agents are able to capture the intruder in both the $m \times n$ -Mesh and the $m \times n$ -Hexagonal Mesh, which is optimal and was proved by Theorem 1 in chapter 2. However, in other strategies, we need to use more agents to clean the network depending on the visibility power of searching agents and the type of Mesh topology. This is because we need more agents to guard some nodes to avoid re-contamination. The number of agents used in the different algorithms are listed in the following tables.

Searching Team Size in Node-Search		
Mesh Topologies	Models	
	<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
$m \times n$ -Mesh	$m + 1$	m
$m \times n$ -Hexagonal Mesh	$m + 1$	m
$m \times n$ -Octagonal Mesh	$m + 2$	$m + 1$
$m \times n$ -Toroidal Mesh	$2m + 1$	$2m$

From the above table, we can observe that when agents have the visibility

power, the searching team size is smaller than the searching team size when agents do not have the visibility power. Therefore, the more powerful agents are the less number of agents are consumed.

– *Number of Searching Steps*

In node-search, the number of searching steps are $O(mn)$ in all types of Mesh topologies. This number is optimal, regardless of the visibility power of agents, agents need to clean all the nodes in the Mesh. In fact, agents need to traverse from the first column to the last column in the given contaminated Mesh. Therefore, the number of searching steps are the same in all of our node-search algorithms.

– *Searching Time*

In three of our algorithms, $NodeSearch_{Mesh-Visb}$, $NodeSearch_{HexaMesh-Visb}$, and $NodeSearch_{ToroMesh-Visb}$, when the agents have visibility power, they are able to capture the intruder in $O(n)$ time on the $m \times n$ -Mesh, the $m \times n$ -Hexagonal Mesh, and the $m \times n$ -Toroidal Mesh respectively. However, in the other algorithms, the searching time is $O(mn)$ which does not depend on the visibility power of agents and the type of Mesh topology. In the following table, the searching time units consumed in different algorithms are listed.

Searching Time in Node-Search		
Mesh Topologies	Models	
	<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
$m \times n$ -Mesh	$O(mn)$	$O(n)$
$m \times n$ -Hexagonal Mesh	$O(mn)$	$O(n)$
$m \times n$ -Octagonal Mesh	$O(mn)$	$O(mn)$
$m \times n$ -Toroidal Mesh	$O(mn)$	$O(n)$

From the above table, we can conclude that the searching time is more efficient when agents have visibility power.

- **Contiguous Edge-Search**

- *Number of Searching Agents*

Depending on the type of Mesh topology, different algorithms require different number of searching agents. The number of agents consumed in different algorithms are listed in the following table.

Searching Team Size in Edge-Search		
Mesh Topologies	Models	
	<i>With a Synchronizer</i>	<i>Agents with Visibility</i>
$m \times n$ -Mesh	$m + 1$	$m + 1$
$m \times n$ -Hexagonal Mesh	$m + 1$	$m + 1$
$m \times n$ -Octagonal Mesh	$m + 2$	$m + 2$
$m \times n$ -Toroidal Mesh	$2m + 1$	$2m + 1$

From the above table, we can see that the searching team sizes are the same in both edge-search models on the same type of Mesh topology. In fact, the agents visibility power does not help decrease the number of agents consumed in edge-search. Moreover, comparing to the searching team size in node-search where agents are with visibility power, edge-search uses one extra agent than that used in node-search to avoid re-contamination.

– *Number of Searching Steps*

Similar to the node-search, in edge-search, the searching steps are $O(mn)$ in all the four types of Mesh topologies. Regardless of whether the searching agents have visibility power or not, they need to clean all the nodes and edges in the Mesh. Therefore, the number of searching steps are the same in all of our edge-search algorithms.

– *Searching Time*

In edge-search, the searching time is $O(mn)$ in all types of Mesh topology. The elapsed searching time units are the same in all of our edge-search algorithms. Moreover, comparing to the searching team size in node-search where agents are with visibility power, edge-search consumes more time units than that consumed in node-search.

From the above comparison, we can see that with our algorithms, the number of agents used in the contiguous node-search is always less than or equal to that consumed in the edge-search. In addition, the searching team size is smaller when agents have the visibility power in some type of network topologies.

6.2 Future Work

Besides what we have studied in this thesis, there are other works and open issues related to the contiguous node/edge-search problem. We only covered a specific aspect in this research area. For future studies, we list some possible future work on the contiguous search problem as follows.

- **Improve on the Lower Bounds**

In this thesis, we first proved that the contiguous node-search problem in a $m \times n$ -Mesh ($m \leq n$) requires at least m searchers. But some of our node-search algorithms use more than m searching agents and $(2m + 1)$ agents are used in the worst case. An interesting problem would be to close the gap that exists between lower and upper bounds in some of our models.

- **Improve on the Searching Time**

As we summarized in the previous section, when agents have the visibility power, they use less time to perform the node-search algorithm. However, in the edge-search, the searching time is not improved in the algorithms when the searching agents have the visibility power. In future, we may study to reduce the searching time consumed in edge-search. One possible solution we may consider is to use more agents to perform the searching tasks if the cost of agents is not a critical issue.

- **Fault Tolerance Issue**

We have assumed that the network and the agents are reliable. Our algorithms

do not consider any broken network links, failed network nodes and agents.

Fault tolerance may be added in our algorithms in the future.

- **Intruder Capture in other Network Topologies**

Our algorithms are applied for four types of Mesh topologies in this thesis. However, there are many other types of Mesh topologies, such as multi-dimensional Meshes. We may study those type of Mesh topologies in future. In addition, the existing researches in this problem are only on Tree [5], Hypercube [7], Butterfly and Chordal Ring topologies. We may also study other types of network topologies such that the contiguous search problem can be applied in more network topologies.

Bibliography

- [1] S. Arnborg and D.G. Cornei and A. Proskurowski. Complexity of finding embeddings in a k -tree. *SIAM Journal of Alg. Disc. Meth.*, **8**, 277-284, 1987.
- [2] H.L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and Treewidth of Graphs. *Journal of Algorithms*, **21**, 358-402, 1996.
- [3] H.L. Bodlaender and T. Kloks and D. Kratsch. Treewidth and Pathwidth of Permutation Graphs. *Proc. of International Conference in Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, **700**, 114-125, 1993.
- [4] H.L. Bodlaender and R.H. Moehring. The pathwidth and treewidth of cographs. *SIAM Journal of Discrete Mathematics*, **6** (2), 181-188, 1993.
- [5] L. Barrière and P. Flocchini and P. Fraignaud and N. Santoro. Capture of an Intruder by Mobile Agents. *Proc. 14-th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Canada, 2002.
- [6] L. Barrière and P. Fraignaud and N. Santoro and D.M. Thilikos. Connected and Internal Graph Searching. Research Report LSI-02-58-R, Departament de Llen-

guatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 2002.

- [7] P. Flocchini and M.J. Huang and F.L. Luccio. Contiguous Search in the Hypercube for Capturing an Intruder. In Proceedings of 19th IEEE International Parallel & Distributed Processing Symposium, Denver, Colorado, USA, 2005.
- [8] R. Breish. An intuitive approach to speleotopology. *Southwestern cavers*, **VI** (5), 72-28, 1967.
- [9] T.H. Cormen and C.E. Leiserson and R.L. Rivest. Introduction to Algorithms. The MIT Press, 1991
- [10] J.A. Ellis and I.H. Sudborough and J.S. Turner. The Vertex Separation and Search Number of a Graph. *Information and Computation*, **113**, 50-79, 1994.
- [11] J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, **45** (3), 233-248, 1993.
- [12] N.G. Kinnersley. The vertex separation number of a graph equals its path width. *Information Processing Letters*, **42**, 345-350, 1992.
- [13] T. Kloks. Treewidth. PhD thesis, Utrecht University, Utrecht, The Netherlands, 1993.
- [14] T. Kloks and H. Bodlaender and H. Muller and D. Kratsch. Computing treewidth and minimum fill-in: All you need are the minimal separators. *Proc. 1st Annual*

- European Symposium on Algorithms (ESA)*, in T. Lengauer, editor, Springer Verlag, Lecture Notes on Computer Science, vol. 726, 260-271, Berlin, 1993.
- [15] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, **47**, 205-218, 1986.
 - [16] A. Lapaugh. Recontamination does not help to search a graph. *Journal of the ACM*, **40** (2), 224-245, 1993.
 - [17] N. Megiddo and S. Hakimi and M. Garey and D. Johnson and C. Papadimitriou. The complexity of searching a graph. *Journal of the ACM*, **35** (1), 18-44, 1988.
 - [18] R.H. Moehring. Graph problems related to gate matrix layout and PLA folding. *Computational Graph Theory*, G. Tinnhofer et al. editors, Springer, Wien, 17-32, 1990.
 - [19] B. Monien and I.H. Sudborough. Min cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, **58**, 209-229, 1988.
 - [20] T. Parson. Pursuit-evasion problem on a graph. *Theory and applications in graphs*, Lecture Notes in Mathematics, Springer-Verlag, 426-441, 1976.
 - [21] T. Parson. The search number of a connected graph. *Proc. of 9-nth Southeastern Conference on Combinatorics, Graph Theory and Computing*, Utilitas Mathematica, 549-554, 1978.
 - [22] S. Peng and M. Ko and C. Ho and T. Hsu and C. Tang. Graph Searching on Chordal Graphs. *Algorithmica*, **27**, 395-426, 2000.

- [23] P. Scheffler. A linear algorithm for the pathwidth of trees. *Proc. of Topics in combinatorics and graph theory*, in R. Bodendiek and R. Henn, editors, Physica-Verlag, 613-620, Heidelberg, 1990.