uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Arno Schulz

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Decontamination of Arbitrary Networks with Multiple Mobile Agent Home Bases

TITRE DE LA THÈSE / TITLE OF THESIS

Paola Flocchini

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Amiya Nayak

Chung-Horn Lung

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Decontamination of Arbitrary Networks With Multiple Mobile Agent Home Bases

A Thesis Submitted to the

Faculty of Graduate and Postdoctoral Studies

in Partial Fulfillment of the Requirements

for the degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

in the University of Ottawa

Ottawa, Ontario

By

Arno Schulz

Your file Votre référence
ISBN: 978-0-494-18465-3
Our file Notre référence
ISBN: 978-0-494-18465-3

ABSTRACT

In this thesis we consider the problem of searching for an intruder in arbitrary networks. A team of mobile software agents is deployed to capture the intruder (i.e., a virus) and decontaminate the network. These agents follow a common protocol, derived from the Breadth-First Search (BFS) algorithm, that is independent of the intruders' speed or knowledge and is designed to prevent any further re-contamination. To measure the efficiency of our protocols and find the ones which give the best results, we conduct experiments on arbitrary synchronous and asynchronous networks. In each experiment, we study a different aspect of the impact multiple starting locations and different levels of visibility have on minimizing the number of mobile software agents required to decontaminate the network and the time used in the process.

ACKNOWLEDGEMENTS

I wish to acknowledge and thank everyone who made the completion of the thesis possible. In particular I wish to thank Dr. Paola Flocchini as well as Dr. Amiya Nayak from the University of Ottawa, whose help, guidance and encouragement helped me. I also want to thank my family, girlfriend and friends for their patience, love and unfaltering support which helped me to complete this work.

I dedicate this Thesis to my Amélie.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The Intruder Capture Problem

Let us consider a network where nodes represent hosts and edges represent connections between hosts. An *intruder* is a dangerous piece of software (i.e., a virus or a malicious/malfunctioning mobile software agent) that can move from host to host contaminating the nodes as it goes through them. The intruder capture problem consists of deploying a team of collaborative mobile software agents to capture the intruder as efficiently as possible.

Since we are interested in solutions that are independent of the knowledge and speed of the intruder, we assume that the intruder can move arbitrarily fast and that it can *see* the other mobile software agents.

We can under these assumptions formulate the intruder capture problem in terms of a *decontamination (or cleaning)* problem in which each node of the network can be in one of three possible states: *clean (or decontaminated)*, *contaminated* or *guarded*. Initially all nodes are *contaminated* except for the nodes containing a mobile software agent (which are *guarded*) and *guarded* nodes are also *clean*. A node becomes *clean* when a mobile software agent passes by it, and an unguarded node becomes *contaminated* if one of its neighbors is contaminated. In other words, we assume that the intruder is able to recontaminate a clean node as soon as there exists an unguarded path between it and a contaminated node.

The goal of the decontamination problem consists of reaching a situation where all the nodes are simultaneously *clean*. In particular, we are interested in decontamination strategies where once a node is clean it remains so until the end of the process. These

types of strategies are called *monotone* strategies.

A mobile software agent is a mobile entity that can move from a node to a neighboring node along the edge connecting them. Mobile Software Agents can communicate by accessing local small whiteboard located at the nodes. The whiteboard of a node will contain the state of the node (*clean*, *contaminated* or *guarded*) and any other information the mobile software agents need to communicate to the other mobile software agent.

1.2 Assumptions and Terminology

In the following section we define our model and our assumptions that will hold true for this Thesis. As stated earlier, since our strategy and the decontamination problem are similar we will mostly use the decontamination problem terminology for this thesis.

The Network:

The network is represented by a simple undirected graph $G = (V,E)$ with vertices representing the hosts and edges representing the communication links between them.

Each node has a bounded amount of local storage, called *whiteboard*. The state of the node is written on the node's whiteboard. Also written on the whiteboard of a node are the port numbers. The whiteboard can also record useful information for the mobile software agents such as the number of mobile software agents present on the node.

Mobile Software Agents:

Operating in G is a team of autonomous mobile software agents. The mobile software agents can move from a node to a neighboring node in G, have computing capabilities, have bounded local storage and obey the same set of behavioral rules (the *protocol*). Initially, the mobile software agents are in nodes, called *homebases*, which are *guarded*, and all the other nodes are *contaminated*. Mobile software agents also have distinct ids, that can be used for selecting a leader. Also note, that in this thesis Mobile Software Agents will also be refered to as agents. While a mobile software agent is moving through the edge, the intruder cannot use that edge.

Mobile Software Agents communicate with each other through the whiteboards. In

fact they can read from and write on the whiteboards. Access to a whiteboard is gained in mutual exclusion. In our thesis we also assume that agents have visibility and/or cloning capability.

- **Timing Assumption** :

  Local computation is considered instantaneous. In this thesis we will consider both synchronous and asynchronous models. In the asynchronous model, every movement of a mobile software agent takes a finite but otherwise unpredictable amount of time. In the synchronous model, we assume there is a global clock: when a mobile software agent leaves a node and moves along one of the edges, it will arrive at the endpoint of the edge in one time unit.

- **Local Knowledge (Visibility 0)**:

  When we consider local knowledge we assume that a mobile software agent located at a node can *see* the whiteboard of its node only and has no other information about the network and the state of the other nodes. In particular, a mobile software agent cannot *see* whether its neighboring nodes are clean, guarded or contaminated.

- **Visibility 1 (local knowledge of distance 1)**:

  With Visibility 1 we assume that a mobile software agent located at a node can also *see* the whiteboards of its neighboring nodes. Specifically, a mobile software agent can *see* whether its neighboring nodes are clean, guarded or contaminated. Notice that this capability could be achieved if the mobile software agents have communication power and can send messages to their neighboring nodes after cleaning a node or guarding a node.

- **Visibility 2 (local knowledge of distance 2)**:

  When we consider Visibility 2 we assume that a mobile software agent located at a node cannot only *see* the whiteboards of its neighboring nodes (as with Visibility 1), but also the whiteboards of the neighboring nodes of its neighbors. In other words a mobile software agent is able to "see" other nodes' whiteboards at a distance of 2. Specifically, a mobile software agent can *see* whether its neighbors and their neighboring nodes are clean, guarded or contaminated, thus enabling the elaboration of a more advanced strategy for the mobile software agents. Notice that this capability

could also be achieved if the mobile software agents have communication power and can send messages to their neighbors neighboring nodes after cleaning a node or guarding a node.

- **Cloning** :

  The cloning capability allows mobile software agents to clone other identical mobile software agents, when needed. With the cloning power, a mobile software agent during the cleaning process can have two operations: create and terminate; on the contrary if mobile software agents do not have cloning power, then all mobile software agents are created at the beginning of the cleaning strategy and terminate at the end of the cleaning strategy. Furthermore, a mobile software agent may clone children mobile software agents of itself which are identical in all respects but have a longer identification number (ie: lets assume the parent mobile software agent has an identification number of 123 then the children clones mobile software agents would have identification numbers such as 1231 or 1232).

The Intruder:

The intruder is also a mobile software agent and can *hide* on nodes. In this thesis we consider the worst case scenario for the mobile software agents where the intruder moves as if it can *see* the whereabouts of the team of mobile software agents, thus avoiding them as much as possible. The intruder cannot move through a guarded node.

Complexity Measures: Efficiency will be measured in terms of number of mobile software agents to be deployed and time.
More precisely:

- **Number of mobile software agents**:

  We want to find the maximum number of active mobile software agents required by a strategy at any given time. We consider all mobile software agents, both the ones that are guarding nodes as well as the ones in transition along the edges.

- **Time**

  As mentioned before, in this thesis we look at two time models for the mobile software agents: synchronous and asynchronous.

In the case of synchronous networks, the mobile software agents take exactly one time unit to traverse an edge. All the agents start and move concurrently at each time unit. The time is the total number of time units used to decontaminate the network. We also refer to a time unit as a step.

In case of asynchronous networks the time taken by the mobile software agents movement is unpredictable. Therefore while the mobile software agents are moving concurrently throughout the network we can only monitor the total amount of time elapsed between the start and the end of the decontamination process.

1.3 Related Work

1.3.1 The Graph Search Problem

A variation of the intruder capture problem has been widely studied in the literature as the *graph search problem.* This problem was first introduced by Breish [7] and Parson [30] and studied extensively under different variations (edge search, node search, mixed search) (e.g., see [10, 20, 22, 26, 29, 32]). The main goal of these investigations was the determination of the minimum number of mobile software agents (also referred to as searchers) required to perform the search. Determining such a number in an arbitrary network is a *NP*-complete problem as shown in [26].

What differentiates the graph search variations studied in the literature from the intruder capture problem is that searchers may be placed and removed from any node of the searching graph, i.e., they are allowed to *"jump"* while they perform the searching task.

There are two versions of the graph search problem that depend on the allowed actions of the searcher. For the *node search problem,* an action consists of one the following operations:

1) place a searcher on a vertex;

2) remove a searcher from a vertex. With the two operations, a node is decontaminated by placing a searcher on the node. An edge (x,y) is decontaminated when two searchers are placed on it, one at each node. In the case that one end of the edge is already cleaned, then the edge can be cleaned by placing one searcher on the other contaminated end of the edge.

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

3) move a searcher along an edge. In this problem a contaminated edge (x,y) can be decontaminated by either:

a. placing two searchers on x and moving one of them to y along the edge (x,y) (i.e., traversing the edge), or

b. by moving a searcher from x along the edge (x,y), if all the other edges incident to x are decontaminated.

In both variations of the graph search problem a node is said to be *guarded* if it contains a searcher; a cleaned node x may be re-contaminated unless x is guarded or all its incident links are clean; a clean edge can be re-contaminated if there exists an unguarded path to a contaminated edge due to the removal of a searcher. In both problems the ultimate goal was to find a suitable strategy that allows the complete decontamination of the graph while minimizing the maximum number of searchers used at any one step. This minimum amount of searchers is called the *node-search number* ($ns(G)$) or the *edge-search number* ($es(G)$) respectively in the node or edge search problem. The strategy is *optimal* if it globally requires a *minimum* number of searches.

The graph search problem has been extensively studied. The goal of most investigations is to prove that finding the search number is an NP-complete problem or to find a linear-time algorithm for specific topologies. It has been shown that the node search number problem is NP-complete in bipartite graphs [18], cobipartite graphs [1], bipartite distance hereditary graphs [19], planar graph with maximum degree three [20], and starlike graphs [16]. Furthermore, the edge search number problem is *NP*-complete for general graphs [26] and for planar graphs with maximum degree 3 [29]. For the node-search problem there exist linear-time algorithms in trees [10, 27, 33], one in k-trees for fixed k [4], in cographs [6], and an $O(pm)$-time algorithm in permutation graphs (where

p is the pathwidth) [5], 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 [32]. The edge search problem can be solved in linear time in trees [26].

One of the reasons for this interest in the search number problem is the fact that it is closely related to classical measures of graph complexity such as cutwidth, vertex separator, pathwidth, treewidth, interval-width. For example, it was first observed by Sudborough that the edge search number of a graph cannot exceed its cutwidth. It is showed in [28] by Makedon and Sudborough that the edge search number of G is identical to the cutwidth of G for all graphs of maximum degree 3. Similarly, Kirousis and Papadimitriou showed in [20] that $|es(G) - ns(G)| \leq 1$ and the node search number $ns(G)$ is equal to vertex separator plus 1; they also showed in [21] that the interval-width of a graph is equal to its node search number. Seymour and Thomas [34] showed that the t-search number of a graph G is equal to the tree width of G plus one. Takahashi [38] proved that for any simple graph G, the mixed-search number is equal to the proper pathwidth of G. Instead of solving the graph search problem directly, by proving it is equivalent to some other graph problems (such as cutwidth, pathwidth, vertex separation), we can apply some of these results to the graph search. Other results on the graph search problem are covered in [8, 9, 13, 35, 37].

In spite of the interest for the compatibility of the search number, very little is known in terms of search strategies and on lower bounds for the search number. In fact, only the tree has been deeply investigated in [26] where an optimal strategy that uses $log_3\ n$ is described.

The graph search problem and its variants have many applications, including pursuit-evasion problems in a labyrinth [7, 30, 31], decontamination problems in a system of tunnels contaminated by toxic gas and network security problem [3, 17]. Moreover, the graph search problem appears in the VLSI design because it is equivalent to the gate matrix layout problem [12, 27].

However, when one considers networked environments, allowing agents to jump is not practical. This leads to another variation of the intruder capture problem: the Contiguous Search problem.

1.3.2 The Contiguous Search Problem

In the Contiguous Search, another variation of the intruder capture problem, the mobile software agents move through the networks' edges as described in [2, 3]. In addition to this, in the Contiguous Search Problem, mobile software agents starts from a single location.

The contiguous search problem has the following requirements:

1) the mobile software agents can move only from node to neighboring node.

2) the strategy is monotone.

3) the decontaminated area is connected.

As we have seen earlier, these are similar requirements used in the intruder capture problem. This is why the contiguous search problem and intruder capture problem are equivalent.

In the contiguous search problem, the contiguous assumption considerably changes the nature of the problem and the results of node and edge graph search problem do not generally apply. In addition, the contiguous search problem increases in difficulty with respect to the non-contiguous one. As mentioned in [26] there exist a minimal search strategy for tree networks without removal that require at least $\Omega(n \log n)$ steps, whereas if the removal is allowed there exists a strategy in each graph that requires at most $O(n)$ steps [22].

Unlike the graph search problem, the contiguous search problem is harder as has been shown [3] that there are networks where the contiguous searching number is strictly greater then the non-contiguous searching number. In particular it has been shown in [3] that the contiguouse search problem is an NP-Complete problem for General Graphs. An investigation on the relationship between the number of searchers of different variations of this model and the graph search has been studied in [2]. It is shown in [2] that the number of mobile software agents required in the contiguous search is no smaller than what is required in the graph search. Several other topologies have been studied for the contiguous search problem and the results are described in the following sections.

1.3.2.1 Contiguous Search in Tree networks

The Contiguous Search has been shown [3] to be solvable with a linear time algorithm for computing the contiguous search number in trees where the corresponding minimal contiguous search strategy is given. It is also shown in [3] that the contiguous search number of any tree with n nodes is at most $\lfloor \log_2 n \rfloor$. In contrast, it is known that $\log_3 n$ searchers suffice for the classical search in every tree with n nodes. The strategy described in [3] determines the starting location starts by using a saturation technique. The tree is labeled in such a manner that a node knows how many mobile software agents are respectively required to decontaminate each one of its' subtrees. Once the tree is labeled, a node can determine how many mobile software agents it requires to decontaminate the entire tree. This number is either the *maximum* required to decontaminate any subtree or in the case of several subtrees requiring the same *maximum* it is the *maximum* $+1$, the extra agent is required to guard the root node. Then to find the minimal contiguous search strategy, a homebase that requires the least number of mobile software agents is selected.

After the home base has been selected, in order to decontaminate the tree, the mobile software agents are sent to each subtree of the home base starting with the one(s) requiring the smallest number of mobile software agents. It has been shown in [3] that with this strategy the contiguous search number and the minimal monotone contiguous search strategy can be found $\Theta(n)$ Time.

1.3.2.2 Contiguous Search in Mesh networks

In the mesh topology the contiguous search problem has also been studied in [24]. The main strategy to decontaminate a mesh type network is to clean one row and then proceed to the next one until the entire mesh is clean. It has been shown that to decontaminate an $m \times n$, with $m \geq n$, requires at least $n+1$ mobile software agents. In the general model, if the mobile software agents have Visibility 1, $min(m,n)$ mobile software agents are required.

The paper considers several variations of the mesh topology: the $m \times n$ mesh, the $m \times n$ hexagonal mesh, the $m \times n$ octagonal mesh and the $m \times n$ toroidal mesh. Let $V_{0,0}$ denote the node in row 0 and column 0.

For example in the $m \times n$ (with $m \geq n$) mesh, the strategy is as follows:
- A special agent called coordinator deploys n mobile software agents at one end of the mesh from node $V_{0,0}$ to node $V_{n-1,0}$.
- The coordinator then moves an Agent in node $V_{0,0}$ to node $V_{0,1}$.It then moves back to the previous node and goes down.
- The coordinator moves all the mobile software agents in this manner until they are all in the next row.
- This is repeated for each row until the entire mesh is decontaminated

The coordinator coordinates the mobile software agents. In this way, mobile software agents move sequentially according to the algorithm.
The authors of this paper introduce the model of visibility 1, with which mobile software agents can determine whether or not to go and clean a contaminated node. In this case a mobile software agent would wait until it has only one contaminated neighbor and then proceed. This allows the mobile software agents to act concurrently.A summary of the results, as described in [24], is shown in the tables below.

	$n \times m$ mesh	$n \times m$ hexagonal mesh	$n \times m$ toroidal mesh	$n \times m$ octagonal mesh
coordinator	$m+1$	$m+1$	$m+2$	$2m+1$
visibility 1	m	m	$m+1$	$2m$

Table 1.1: Mobile Software Agents Required

1.3.2.3 Contiguous Search in Hypercube networks

The Hypercube has been studied in [25], where the minimum number of mobile software agents has been determined and a strategy developed. The strategy is based on the *broad-*

cast spanning tree (commonly used to do optimal broadcast in the hypercube), which is employed for the decontamination process. The broadcast spanning tree for a hypercube of dimension d has $d+1$ levels. All the nodes with i bits of their labels set to 1 are regrouped in level i of the broadcast spanning tree. For example, the node labeled 001001, has two bits of its label set to 1, it will be located in level 2 of the broadcast spanning tree. All the nodes at level i are linked to nodes at level $i+1$ (i.e.: level 0 has only the root node labeled 000...000, and level 1 has all its' neighbors 000...001, 000...010, ..., 010...000 and 100...000).

As with the mesh we are going to illustrate the algorithm using a coordinator. The algorithm is described as follows:
- Initially the mobile software agents start in the root node, which in this case is also the homebase, and the coordinator guides them one by one to each node at level 1 in the broadcast spanning tree.
- For level i the coordinator goes to the node with the smallest label, and guides the mobile software agents to the neighboring nodes at level $i+1$ that are still contaminated. This is then done until the coordinator reaches the last node on level i.
- This is repeated until the coordinator reaches node 111...111.

As with the mesh network two variations have been studied, one with local knowledge and the other with visibility 1. In particular, it is shown that in both cases $\Omega(\frac{n}{\sqrt{log\ n}})$ mobile software agents are required. This number of mobile software agents is also sufficient it is achieved using local knowledge. There is however a significant time difference between the two models, while with a coordinator $O(n)$ time is required, with visibility only $O(d)$ (d being the number of levels in the broadcast spanning tree) time is required. Similar to

	$n \times m$ mesh	$n \times m$ hexagonal mesh	$n \times m$ toroidal mesh	$n \times m$ octogonal mesh
coordinator	$O(mxn)$	$O(mxn)$	$O(mxn)$	$O(mxn)$
visibility 1	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Table 1.2: Time Required

what was observed with the mesh, this gain in time comes from the nature of the algorithm: while with a coordinator the algorithm is sequential, with visibility the mobile software agents act concurrently.

1.3.2.4 Contiguous Search in Chordal Ring networks

Another topology in which the contiguous search problem has been studied in [11] is the chordal ring topology. A chordal ring is an augmented ring. Formally it is defined by the pair (n,L) where n is the number of nodes of the ring, and L is the set of chords $L \subseteq d_2, ..., d_k$. Each cord $d \in L$ connects every pair of nodes of the ring that are at distance $|d|$ in the ring. We also define a "window" as a set of $|d_k|$ consecutive nodes on the outer ring. The strategy used to decontaminate the Chordal Ring is based on its structure. The number of mobile software agents required depends on the length $|d_k|$ of the longest chord d_k. The strategy can be described as follows: a coordinator initially deploys two consecutive windows of mobile software agents on the ring. Then the coordinator progressively moves the first window through the entire ring until it is decontaminated.

As with the hypercube and the mesh, the algorithm has been studied in both local knowledge and visibility 1 model.
The local knowledge strategy employs $2|d_k| + 1$ mobile software agents. In [11] it has been shown that this number of mobile software agents is indeed required since it is a lower bound. As with the hypercube, due to the sequential nature of this strategy, it uses O(n) time.

The strategy with visibility 1 employs $2|d_k|$ mobile software agents. Also in this case it is shown in [11] that this is a tight bound since $2|d_k|$ mobile software agents are also necessary. Similar to the hypercube and the mesh a gain is observed in terms of time. The chordal ring with visibility 1, can be decontaminated in $O(\frac{n}{|d_k|-|d_{k-1}|})$ time. It is interesting to note that the number of concurrent movements executed by the algorithm depends on the difference between the length of the longest and second longest chord.

1.3.3 Contiguous Search with Local Immunization

Another interesting variation is studied in [23], where an unguarded clean node with a contaminated neighbor is not immediately re-contaminated. The re-contamination only happens when a majority of the neighboring nodes of a decontaminated node are contaminated. This is particularly useful in environments such as distributed databases, where a local vote is done to elect which local version is the correct one. Two synchronous networks were studied in [23], k-dimensional toroidal meshes and general trees.

For the k-dimensional toroidal mesh the principle comes down to the fact that no node can be re-contaminated unless a majority of its neighbors are contaminated. This new requirement allows to decontaminate a 2-dimensional *nxm* toroidal mesh with only 4 mobile software agents. The general concept of the algorithm in a 2-dimensional *nxm* toroidal mesh is to keep the clean nodes in a rectangular shaped zone and expand this zone until the entire mesh is decontaminated. This works as each node not in the corners of the rectangle has at least 3 clean neighbors, thus satisfying the majority rule.

More precisely the algorithm for a 2-dimensional *nxm* toroidal mesh is described below:

- From the homebase, 3 mobile software agents move to adjacent nodes to form a square of 4 nodes in the mesh.
- Then two mobile software agents (in the same column) move in parallel to decontaminate the two first rows of the mesh. They stop when they are adjacent to the two other mobile software agents. The decontaminated hosts now form a rectangular zone that covers the two first rows of the mesh.
- The two agents in the bottom row go down to the next row. One of these agents then proceeds to decontaminate this row. This is repeated until the entire mesh has been decontaminated.

As proven in [23] because of the local "immunization" effect, agents are not required

to guard all the nodes of the rectangle, just its corners. In particular it is shown that the contiguous search of any k-dimensional toroidal mesh, with $k \geq 1$ requires 2^k mobile software agents, as only the corners need to be guarded.
It is also shown in [23] that for disinfecting a complete k-ary tree T of n vertices, with $k \geq 4$, at least $\lceil log_k n \rceil$ mobile software agents are required.

1.4 Our Contribution

In this thesis we consider a new approach to the intruder capture problem (or decontamination problem) in arbitrary and d-regular arbitrary networks. Contrary to previous investigations, on the intruder capture problem, we do not require the decontaminated hosts to form a connected graph. We study both situations where the decontamination is done in classical manner (i.e.: from a single starting location) and when it is done with multiple starting locations. In addition to this, the agents have different levels of visibility (i.e.: visibility 1 or 2) and the ability to clone themselves to increase their autonomy. Our goal is to allow the agents to decontaminate the network on their own without requiring a specific topology algorithm, a coordinator or a pre-computed set of moves. In this manner we can create a set of more general algorithms for agents that can be used in networks where the topology can be possibly unknown.

Without the requirement of connectivity, the agents can decontaminate the network starting from several different locations. In particular, different agents teams can perform the decontamination concurrently until the decontaminated areas merge and the all the networked hosts are clean. However, we do require that each of the decontaminated areas is expanded in a monotone way, in order to guarantee the monotonicty of the overall strategy.

Our strategy has the following requirements:

1) the agents can move only from a node to a neighboring node along the edge connecting them.

2) the strategy is monotone.

We want to determine the minimum number of agents required to decontaminate an arbitrary networks. The problem of determining the minimum number of agents required to decontaminate an arbitrary networks is an NP-Complete problem. In fact, if there existed a polynomial time algorithm P that could determine which set S of starting locations in graph G uses the least number of agents, then P could also be used to solve the graph search problem. But, as shown in [26], the Graph Search Problem is NP-Complete for arbitrary graphs, therefore the decontamination problem is also NP-Complete.

In this thesis we study how different numbers of starting locations and different visibility models have an impact on the minimum number of agents and the time required to decontaminate a given arbitrary network. In our strategies, the starting agents begin the decontamination process by following a Breadth-first expansion from their home base(s). In doing so, new agents are cloned and they ensure that no re-contamination occurs. This technique can be applied to any arbitrary topology and can be started in an arbitrary number of starting locations. Its efficiency varies on the basis of the number of starting places and their locations. Generally, starting from more locations will initially increase the number of agents while decreasing the time required to decontaminate the network.

An interesting problem is to determine, given an arbitrary network, where to locate the starting locations in order to increase the efficiency. We study this with synchronous d-regular arbitrary networks where the agents have visibility 1. For graphs of given size and degree, we want to find how many starting locations are needed to minimize the number of agents deployed simultaneously. This is done using a genetic algorithm to find good combinations of starting locations.

We show that, the amount of time is reduced with an increased number of starting locations. Furthermore, we observe that in most cases the maximal number of agents deployed is higher when a single homebase is employed instead of multiple ones.

The observations made in our first experiment raise a question as to if we can observe similar patterns when using random starting locations. To verify this we run experiments in both arbitrary and d-regular arbitrary synchronous networks considering both levels of

visibility 1 and 2.

The patterns observed previously remain similar with d-regular arbitrary graphs and visibility 1. We have also found that the patterns are present in arbitrary networks, as long as some conditions are satisfied.

We also observed similar results with visibility 2. Both arbitrary and d-regular arbitrary network showed similar results, as long as they have the same number of edges and vertices. However, we discovered that visibility 2 greatly reduces the number of agents required to decontaminate the network.

Finally a last question is raised, as to how the models will behave in asynchronous networks. For this we have generalized our models for asynchronous networks. We have looked at both visibility 1 and 2 and applied the asynchronous algorithm to our models.

While we did not achieve our original goal of improving our models, as both of them where shown to be inadequate to decontaminate arbitrary networks, we made several interesting observations from these failures. This experiment allowed us to achieve a better understanding as to how decontamination happens in asynchronous arbitrary network.

CHAPTER 2

EXPERIMENTAL PARAMETERS AND PRELIMINARY OBSERVATIONS

2.1 Experimental Setup

In this thesis we consider three separate experimental sets. Each set of experiments reflects a different goal in order to better understand the impact that multiple starting locations have on reducing the maximum number of agents used.

In our first set of experiments we look at synchronous d-regular arbitrary networks. In particular we are interested in finding good combinations of starting location that minimize the maximum number of agents required to decontaminate a network. In order to find a suitable combination, while not using a variation of exhaustive search, we used a genetic algorithm, which allowed us to find good solutions (though not necessary optimal).

In our second set of experiments we verified if the results obtained with the genetic algorithm would hold in more general terms (no longer with a good combination of starting locations but also with random combinations of starting locations). Further more we wanted to see if we could improve the results of our first experiment by introducing a visibility 2.

In our final set of experiments, we study how our models work in asynchronous networks. In particular we considered what modifications were required in order to make the models work and what their limitations were.

For statistical significant results we ran each set of experiments over 60'000 individual runs. The entire experiment is divided into 200 different graph groups (5 graph size 512;768;1024;1576;2048, 4 degrees 4;8;16;32, and 10 different amount of home bases

1;2;3;4;6;8;10;12;14;16). We will refer to each graph group as follows: $(2048, 32, 16)$ where 2048 is the number of vertices, 32 the average number of edges per node and 16 the number of starting locations in the network. Each test for a group has 30 series that use 1 particular graph of the group which is then run 10 times by the genetic algorithm. Each run yields two results, the number of mobile agents used and the time used.

Once all the data has been collected for each graph group series, the median is taken for both sets of data. Another median is then taken of all the medians of the series for each graph group giving the resulting typical number of mobile agents used for the specific graph groups and the time used for the specific graph groups. Each graph has four curves, which we also refer as graph families.

2.2 Graph Generation

As we are looking into arbitrary and d-regular arbitrary graphs we define the algorithms to generate them in the following two sections.

2.2.1 Generating an arbitrary graph

An arbitrary graph of size n and degree d is a connected graph with n vertices and $(n \times d)/2$ bidirectional edges. The edges are distributed arbitrarily amongst the vertices. However it should be noted that, while there are loops, there are no parallel edges. The procedure used to generate an arbitrary graph g is as follows:

Given the input of a given size n and degree d, we fix the total number of edges to be $(n \times d)/2$. Then the edges are allocated randomly between all the pairs of vertices (a, b) of G such as there are no two edges between the same pair. A more formal definition the algorithm is given in Figure 2.1. It should also be noted that to further study the impact of the number of edges has on the results obtained with visibility 1 and 2, we created 2 versions of this generator: one which uses a constant number of edges, the other with a varying number of edges. This difference will be pointed out in the experiment of Chapter 4.

Random graph generation
1.Start with n vertices1,2,...,n. Set U = 1,2,...,n. (U denotes the set of available vertices.)
2. Choose two random vertices i and j in U, and if they are suitable (ie: no edge exists between vertices i and j), pair i with j. Repeat this until no suitable pair can be found in U.
3. Create a graph G with an edge from vertex r to vertex s if and only if there is a pair containing these vertices.
4. If G is connected, output it, otherwise return to Step 1.

Figure 2.1: Quick Generation of an arbitrary graph

2.2.2 Generating d-regular arbitrary graph

A d-regular arbitrary graph of size n and degree d is a connected graph with n vertices and $(n \times d)/2$ bidirectional edges. The edges are distributed equally amongst the vertices (i.e.: each vertices has d edges). It should be noted that loops are allowed, parallel edges are not allowed.//

Given the limitations and requirements of generating a random regular graph, modifying the algorithm in figure 2.1 would create an inefficient algorithm. We have found an algorithm that can quickly generate random regular graphs as described in [36]. The procedure used to generate a d-regular arbitrary graph G is as follows:

Given the input of a given size n and degree d, we fix the total number of edges to be $(n \times d)/2$. Then the edges are allocated randomly between all the pairs of vertices (a,b) of G such that there are no two edges between the same pair and that both a and b still have open edge ports. A more formal definition of the algorithm is given in Figure 2.2.

Regular Random graph generation

1.Start with $n \times d$ points(1,1),(1,2),...,(1,d),...,(n,d) ($n \times d$ even) in n groups . Put $U = \{(1,1),(1,2),...,(1,d),...,(n,d)\}$.

(U denotes the set of unpaired points.)

2.Choose two random points i and j in U, and if they are suitable, pair i with j and delete i and j from U. Repeat this until no suitable pair can be found in U.

3. Create a graph G with an edge from vertex r to vertex s if and only if there is a pair containing points in the r'th and s'th groups.

4. If G is d-regular and connected, output it, otherwise return to Step 1.

Figure 2.2: Quick Generation of d-regular arbitrary graph

2.3 Intruder Capture Problem with Local Knowledge

Our strategy allows for several visibility models to be considered; in the following chapters we will look at visibility 1 and 2 for which we have run experiments. However, for the local knowledge model, we did not run any formal experiments. In this section, we look at the local knowledge model from a theoretical point of view. Unlike visibility 1 and 2, the local knowledge model clearly limits the possible actions agents can do. As a matter of fact, it only allows communication directly with other agents on the current node.

2.3.1 Definition and Algorithm

When we initially looked at the decontamination problem, we considered synchronous d-regular arbitrary networks without the restriction that the decontaminated host be connected. What the local knowledge model implies for the is that they can only *see* the local node and its outgoing edges. For practical reasons we assume that the agents are able to distinguish which edges are linked to their parent node (this could be done via edge label-

ing on the nodes' whiteboard); furthermore agents are able to locally select a leader using their unique ids.

Similar to our other experimental sets we use a breadth first search strategy (see figure 2.3) by which agents ensure that no clean node will be re-contaminated. We also observed the strategies' limitations in Figure 2.4.

Protocol DECONTAMINATION WITH LOCAL KNOWLEDGE (for an agent a arriving at node x)

If a is alone:
  Clean x.
  Let N_x be the set of neighbors of x not including the parent node.
  Clone $|N_x|$ agents.
  Send the cloned agents to the neighbors.
If a is not alone:
  Locally choose a leader.
  If a is the leader:
    Clean x.
    Let N_x be the set of neighbors of x not including the parent nodes of all local agents.
    Clone $|N_x|$ agents.
    Send the cloned agents to the neighbors.
  Otherwise
    Terminate.

Figure 2.3: Protocol DECONTAMINATION WITH LOCAL KNOWLEDGE

Theorem 2.1: Protocol DECONTAMINATION WITH LOCAL KNOWLEDGE performs a monotone decontamination of the network.

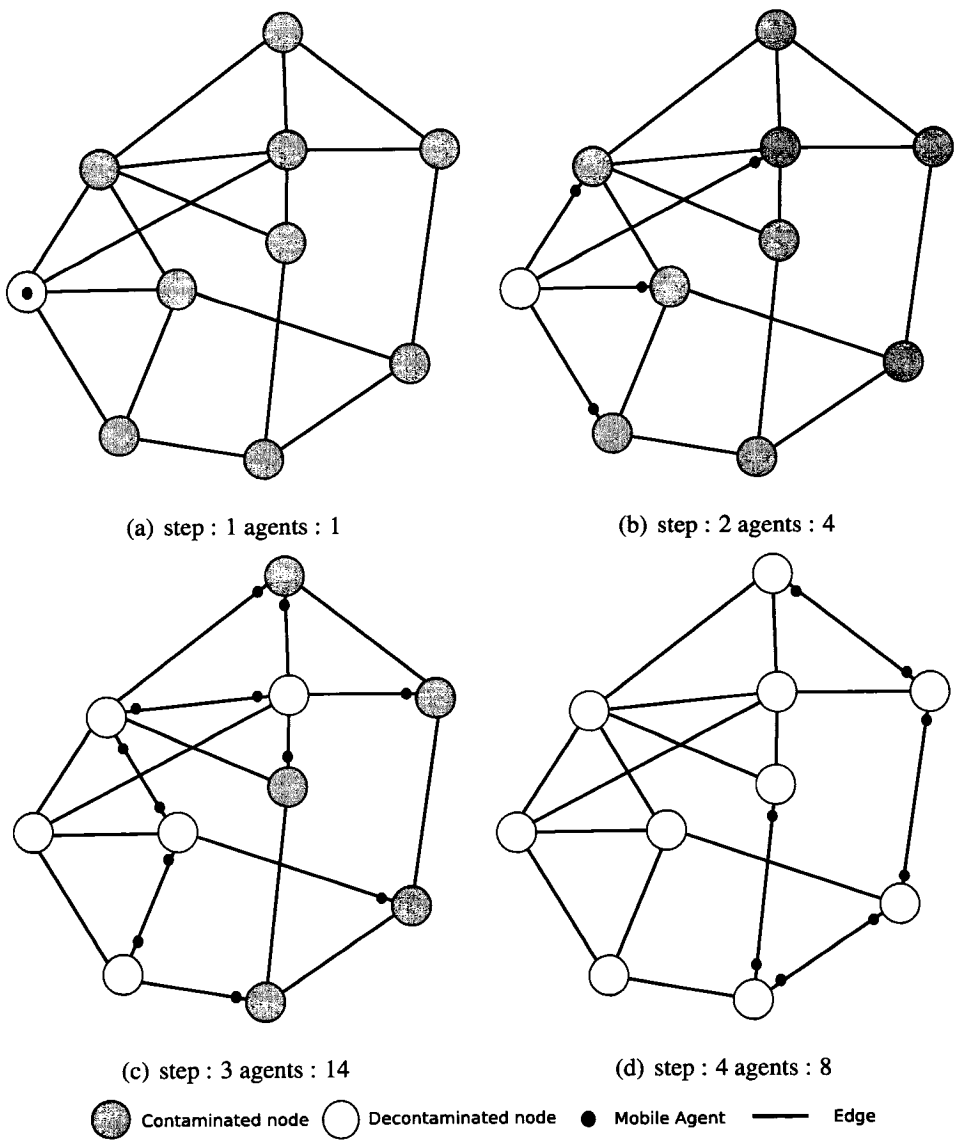
Proof: We want to prove by induction that once a node is clean, it will never be re-contaminated.

Basis. The starting locations of the agents are clean and since, by definition, enough agents are sent simultaneously to all their contaminated neighbors, they will not be re-contaminated in the subsequent step. Moreover, the border nodes are all guarded (in this case border nodes are just single nodes).

Induction. At step k, some nodes are clean and the border nodes are guarded. At step $k+1$ the agents that are on the border nodes will clone themselves, by definition of the algorithm, and then proceed to the neighboring nodes. As all contaminated neighbors receive at least one agent, the old border nodes become internal nodes, and thus cannot be re-contaminated in the subsequent step, while the border node are all guarded.

By induction we have shown that once a node is clean, it will never be re-contaminated. Since the graph is connected, all nodes will eventually be decontaminated. □

As we can see in Figure 2.4while using the local knowledge model is the simplest manner to go through the network, it is also the method that uses the most agents to clean the network as they will traverse any edge that does not belong to a parent node. In this case the results for synchronous or asynchronous networks are similar, as an agent will send his clones to all neighboring nodes except its' parent edge(s) in both cases. In particular, we can observe that, the complexity of *Flood protocol* is $O(m)$ and in the worst case it is $O(n^2)$.



(a) step : 1 agents : 1

(b) step : 2 agents : 4

(c) step : 3 agents : 14

(d) step : 4 agents : 8

(e) Legend

Figure 2.4: Flood protocol in d-regular arbitrary Graph with local knowledge

CHAPTER 3

SOLVING THE DECONTAMINATION PROBLEM IN SYNCHRONOUS NETWORKS USING A GENETIC ALGORITHM

In our first set of experiments, we consider the contiguous decontamination problem in a synchronous arbitrary d-regular network without the restriction that the decontaminated nodes must be connected. We first describe a general strategy in which the agents perform the decontamination by moving in a breadth first search manner, making sure that no re-contamination will occur. This technique can be applied to any arbitrary topology. This general strategy can be initiated by an arbitrary number of starting locations, and its efficiency depends on the number of starting places and their location.

Generally, starting from more locations will increase the number of agents but will decrease the time for decontamination. An interesting question is: given a network and a number of starting locations what is the optimal placement of the agents? Even in symmetric networks, by increasing the number of starting locations, the problem becomes quite complex; thus, in order to obtain the minimum number of mobile software agents required, we resorted to simulations using a genetic algorithm. In fact, to choose good starting locations, we designed a genetic algorithm that will find a solution which uses the least number of agents in a single step, for a given graph with a fixed number of starting locations. Through experiments, we show that as the number of home bases increases, the number of agents required decreases in most network topologies considered.

3.1 The Strategy

The cleaning strategy (protocol CLEAN illustrated in Figure 3.1) is straight forward. Initially, the agents are placed in arbitrary starting location. Each starting agent will try to move its clones on a breadth first search tree of the network rooted at its starting position. More precisely, at each step, if an agent arrives to a node alone, it cleans the node, clones itself as many times as the number of contaminated neighbors, and sends them on the corresponding links. If, however, more than one agent arrives at a node simultaneously, only one of them survives, cleans the node, clones itself as many times as the number of contaminated neighbors, and sends them on the corresponding links; the other mobile software agents terminate at the node.

The procedure *"locally choose a leader"* in Figure 3.1 consists in selecting one of the agents to continue the cleaning operation, the leader is the agent with the smallest id.

In a clean area, internal nodes are the nodes whose neighbors are all clean, border nodes have some clean neighbors and some contaminated neighbors.

Theorem 3.1: Protocol CLEAN performs a monotone decontamination of the network.

Proof: We want to prove by induction that once a node is clean, it will never be re-contaminated.

Basis. The starting locations of the agents are clean and since, by definition, enough agents are sent simultaneously to all their contaminated neighbors, they will not be re-contaminated in the subsequent step. Moreover, the border nodes are all guarded (in this case border nodes are just single nodes).

Induction. We assume that at step k, some nodes are clean and the border nodes are guarded. At step $k+1$ the agents that are on the border nodes will clone themselves, by definition of the algorithm. Then the clones proceed to the contaminated nodes. As all contaminated neighbors receive at least one agent, the old border nodes become internal nodes. Therefore, these nodes will not be re-contaminated in the subsequent step, while

Protocol CLEAN (for an agent a arriving at node x)

If a is alone:

  Clean x.

  Check the state of neighbors.

  Let $N_{C(x)}$ be the set of contaminated neighbors of x.

  Clone $|N_{C(x)}|$ agents.

  Send the cloned agents to the contaminated neighbors.

If a is not alone:

  Locally choose a leader.

  If a is the leader:

    Clean x.

    Check the state of neighbors.

    Let $N_{C(x)}$ be the set of contaminated neighbors of x.

    Clone $|N_{C(x)}|$ agents.

    Send the cloned agents to the contaminated neighbors.

  Otherwise

    Terminate.

Figure 3.1: Protocol CLEAN

the border node are all guarded.

By induction we have shown that once a node is clean, it will never be re-contaminated. Since the graph is connected, all nodes will eventually be decontaminated. □

Notice that, for some specific topologies, it can be easy to compute the number of agents needed for decontamination, when considering a single starting location ("home base"). In this case, the number of steps is always equal to the diameter of the network, while the number of agents clearly depends on the topology. For example, in the case of

the hypercube, the maximum number of agents simultaneously active would be equal to the maximum number of edges between levels of the broadcast spanning tree as seen in the introduction.

However, when we have more than one home base for the agents, the nature of the problem becomes more complex with the added difficulty of finding the optimal configuration. Even in symmetric networks, adding multiple starting locations increases the complexity; thus, in order to obtain minimum number of agents we resorted to simulations using a genetic algorithm.

3.2 The Genetic Algorithm

3.2.1 Genetic Algorithm Background

Genetic Algorithm:

For 52 generations do
    Set total fitness TS to 0.
    While there exists a candidate string out of 1024 candidates
        s = next candidate string.
        Evaluate s with the fitness function $F(s)$.
        Add $F(s)$ to TS.
    End While
    Create a biased wheel of size TS where for each s,
        a space corresponding to the $F(s)$ fitness is allocated.
    Pick randomly 1024 candidate strings from the wheel.
End For

Figure 3.2: Genetic Algorithm

The genetic algorithm as described by Goldberg [14] is a process composed of two

elements: a population of strings and a fitness function. As can be seen in Figure 3.2, at each round of the genetic algorithm, the fitness function allows to pick the fittest individuals (the strings that achieve a better score according to the fitness function) and then uses them as a basis to generate a new generation. As each generation brings a new population generated from the fittest individuals from the previous generation, over the course of several generations, the individuals tend to get better and better fit, thus providing a good solution in a relatively short period of time as the least fit individuals are no longer considered. However, evolving the optimal solution is more difficult and will take more time as in the later generations. When there are many individuals with equal fitness, there is no bias towards any particular individual.

3.2.2 Our Implementation

For our experiment, we use individuals where each individual represents the starting positions of the agents within the graph. The graphs are represented by a Boolean array where each entry of the array corresponds to a node in the graph. If the Boolean value corresponding to a node is true it means that the node is going to be a starting location for an agent. These arrays are stored in matrix composed of 1024 individuals (the population size used by the genetic algorithm).

In order to have a starting point for the genetic algorithm, the initial population is generated in a random manner. Each individual, having the same number of agents, will place these agents in random positions of the array (if an agent is already there, another random position is selected until an empty position is found). Once the population has been generated, it is then passed on to the genetic algorithm which then runs 52 times, each time generating a new population (from the previous generation).

Each population that is passed on to the genetic algorithm is evaluated according to a "fitness" function. The fitness function used is the maximum number of agents that were used at any one step during the decontamination of the network. The fitness obtained is then stored in order to determine the maximum fitness and the best individual of each gen-

eration later on.

After all the fitnesses have been collected, the algorithm sets up a biased evolutionary wheel, that gives to each individual a percentage of the wheel proportional to its fitness compared to the total fitness of the population (note that the wheel represents the total fitness of the population). This favors the fittest individuals as they will represent a greater percentage. Thus, while creating a new generation, as the selection is done in a random manner, the fittest individuals are most likely to be selected. The new generation is then created using one of two techniques, *mutation* and *cross-over*, described below.

The *mutation* consists of selecting an individual randomly in the evolutionary wheel. We then change the position of one of the agent's starting home base within the Boolean array which represents an individual (note that it is still possible for a selected individual not to get changed during the mutation).

The *cross-over* consists of selecting two individuals randomly in the evolutionary wheel and then creating two new individuals by switching the second half of each of the selected individuals. For example, if the two parts of the first selected individual is (x1, x2) and the second selected individual is (y1, y2), the cross-over process creates two new individuals with two parts as (x1, y2) and (y1, x2).

After the new generation has been bred, the best of this generation is then compared with the best of all previous generations; if the fitness is higher, the new champion individual is preserved. Finally the next generation is passed to the genetic algorithm, and the process is repeated until we reach the 52nd generation.

3.2.3 The Benefits and Restrictions of Genetic Algorithms

The advantage of using the genetic algorithm is that it allows us to narrow down to an acceptable solution for a given topology. Most importantly, it gives us a good approxima-

tion (if not the maximum number of agents required) for a particular network. We are also able to use it instead of doing an exhaustive search to evaluate each and every possible configuration of the mobile agents starting home bases for a given network topology.

There are two limitations with this approach. First, without resorting to an exhaustive search, it is not possible to confirm whether the results obtained by the genetic algorithm truly represent the optimal solution for a particular graph. Second, there is the possibility of premature convergence giving a local minima as a result instead of finding a lower minimum. To minimize the effects of these limitations, we run each configurations 300 times.

3.3 Experimental Results

The goal of this set of experiments is to find a minimum number of mobiles software agents used for decontaminating d-regular arbitrary networks with one or more home bases. In the following section, we present our observations and analysis on the results obtained.

3.3.1 Observations and Analysis

We observed that within a graph group, the variation of results for the number of agents between individual experiments and between series of experiments is less than 3%. Occasional abnormalities in the results due to pre-mature convergence in the genetic algorithm are less than .5% and are ignored. For the time units, we also observed less than .5% abnormality (considered when the variation is not within ± 1 time unit).

3.3.1.1 Number of Agents Used

Figure 3.3 shows the results obtained by the genetic algorithm for different graph groups for agents.

For a given graph group, fewer agents are required for certain numbers of home bases. For example, the graph group (2048, 32, 2) uses fewer agents than the graph group (2048, 32, 1). The results seem to follow certain patterns based on their graph degree (eg: all graphs of degree 32 start with a higher number of agents which then drops to increase later). The variations of the number of agents used depends on the degree of the graph groups. The variation is larger for graph groups of higher degrees.

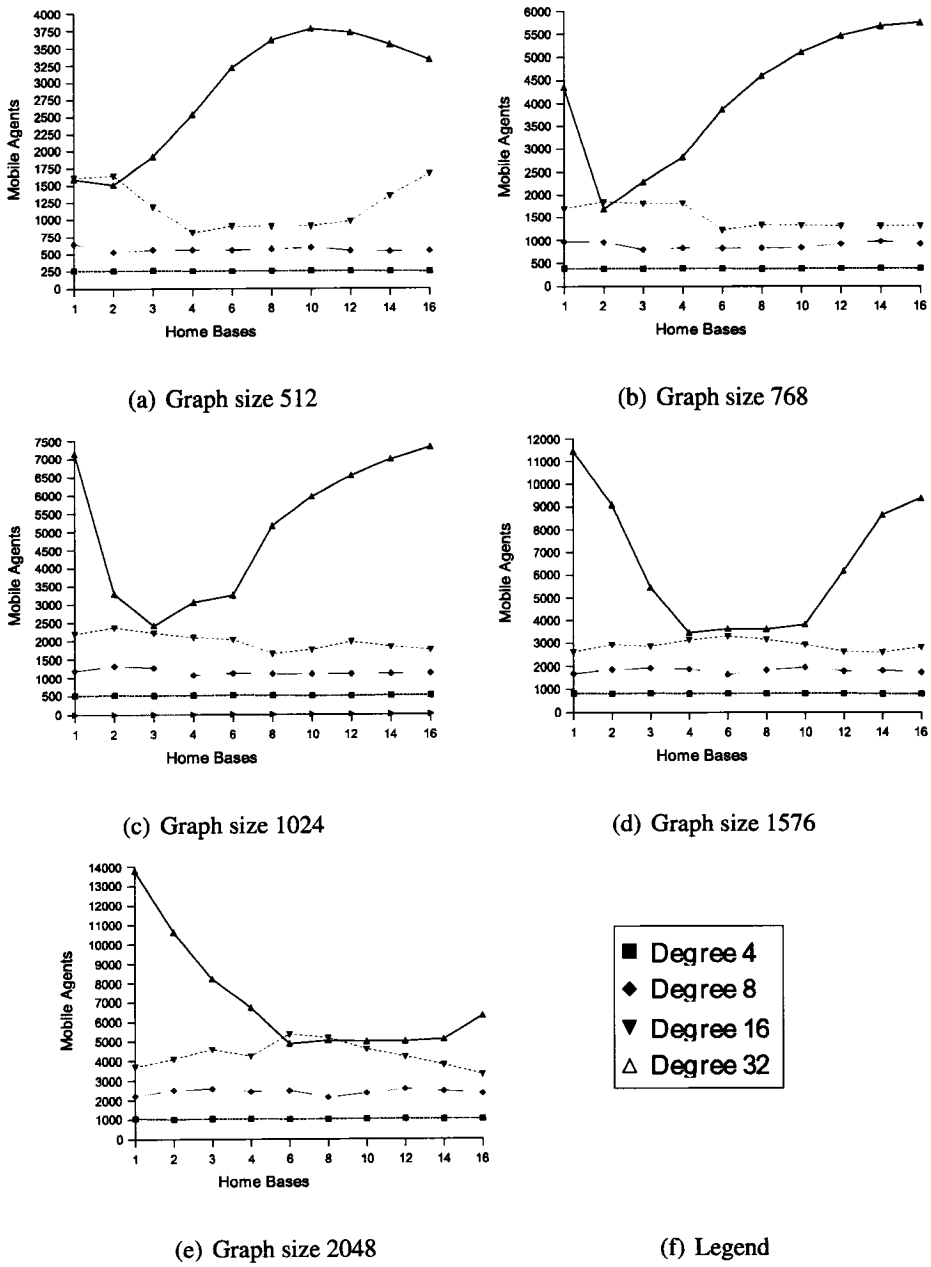
These results behave as we expected we clearly see there is an overuse of the number of agents such as in graph group (2048, 32, 1) where the algorithm uses 14000 agents when there are only 2048 nodes. This comes from the nature of the decontamination strategy; since our algorithm is a variation of the breadth first search. With visibility 1 a contaminated node will always receive an agent from all its decontaminated neighbors. therefore the overuse of agents is proportional to the ratio of decontaminated and contaminated nodes. Initially, the overuse is low as there are fewer decontaminated neighbors for each contaminated node. The overuse is highest in the last step of the decontamination process as most of the neighbors of contaminated nodes have been decontaminated.

Figure 3.5 illustrates the propagation of the agents, in the given graph of degree 4 with 7 nodes and 1 home base, through the network using the BFS technique. In the last step, while there are only 2 nodes left to be decontaminated, 6 agents are used by the algorithm. This example clearly shows the overuse of agents when using the BFS technique in a synchronous network.

3.3.1.2 Time Used

In Figure 3.4 we can see the average time required during the experiments. These results behave as we expected; we clearly see that with an increased number of starting home bases the time decreases steadily. Also it is interesting that overall in our set of experiments Figure 3.4 shows that the time varies between 2 and 9 time units.

Both of these observations were expected due to the nature of our algorithm. In fact we



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

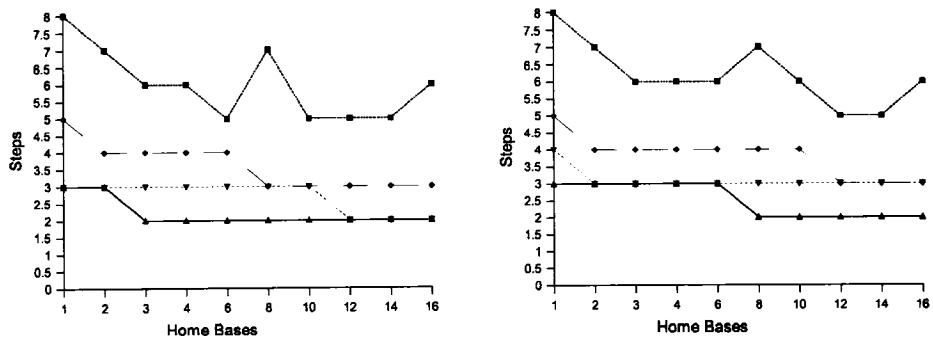
Figure 3.3: Genetic Algorith Minimum Agents Used in a Random Regular Graph with Visibility 1

can at each step, in this strategy, it is possible to estimate what that the number of agents is going to be at the following step. For example, in a network of degree 4 with 512 nodes and one starting base, the number of agents would roughly increase 4 fold at every step. One can see that once we are at a step where the number of agents exceeds the number of nodes in the graph it would be ideally the maximum of steps used for that graph.

3.4 Summary

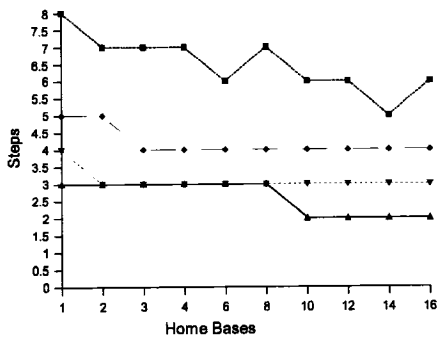
In this chapter, we have proposed a new approach to solving the decontamination problem with multiple starting locations. As expected we have observed that the time required to decontaminate the entire network decreases when the number of home bases and/or the degree of nodes increases. For agents, despite using the genetic algorithm in an attempt to find good solutions, the results obtained are disappointing. But, these same results led us to several unexpected observations, on the patterns observed and the overuse of mobile software agents.

These observations bring several interesting questions. In the following chapters we look into some of these question. In particular, in Chapter 4, we will study if the patterns are a results of the selection of good combinations of starting locations or if they can be repeated with random combinations of starting locations, and how we can eliminate the overuse of mobile software agents. In Chapter 5, we will study how our algorithms work and what modifications need to be done to make them work in asynchronous networks.

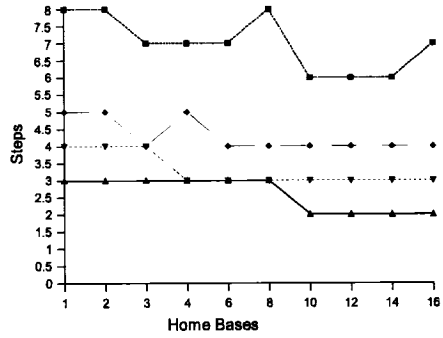


(a) Graph size 512

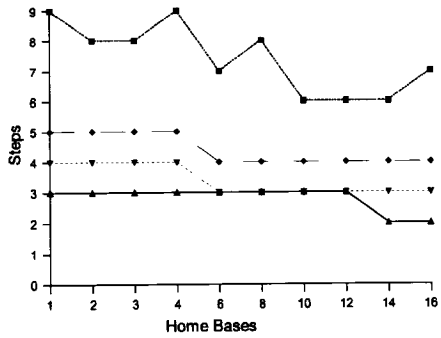
(b) Graph size 768



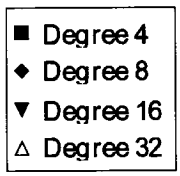
(c) Graph size 1024



(d) Graph size 1576

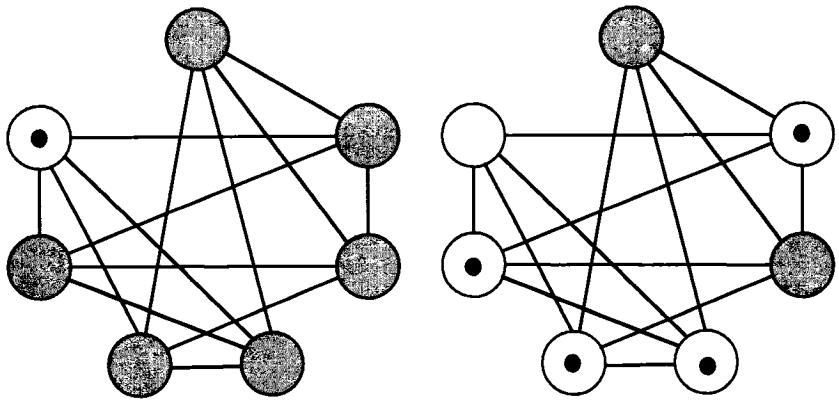


(e) Graph size 2048



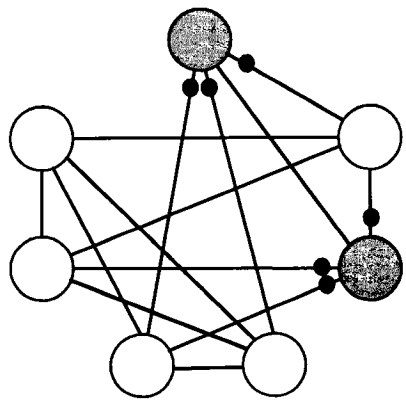
(f) Legend

Figure 3.4: Genetic Algorith Minimum Steps Used in a Random Regular Graph with Visibility 1



(a) Step 0

(b) Step 1



(c) Step 2

(d) Legend

Figure 3.5: Overuse of agents

CHAPTER 4

DECONTAMINATION OF SYNCHRONOUS NETWORKS

For our second set of experiments, let us consider the decontamination problem in synchronous networks. We want to verify if the observations made in Chapter 3 hold for random combinations of starting locations. We will look at both visibility 1 and 2 to see if we can overcome the limitations described in Chapter 3.
We describe the general strategies in which the agents perform the decontamination using visibility 1 and 2. As in Chapter 3, these strategies can be applied to any topology. This general strategy can be initiated by an arbitrary number of starting locations, and its efficiency depends on the number of starting places and their locations.

As we have seen in Chapter 3, generally in most cases for d-regular synchronous networks, having more starting locations not only decreases the time but also decreases the number of mobile software agents. An interesting problem is to see if we can generalize our algorithm and see if we still observe the same behavior. Through experiments, for both visibility 1 and 2, we show that as the number of home bases increases, the number of agents required decreases in both arbitrary and d-regular arbitrary networks with random starting locations.

4.1 Visibility 1

The cleaning strategy (protocol CLEAN illustrated in Figure 4.1) is the same algorithm we have previously seen in Chapter 3. Initially, the agents are placed in arbitrary starting locations. Each starting agent sends its clones in a BFS manner. More precisely, when the agent is the only agent to arrive at a node, it cleans the node, clones itself as many

times as the number of contaminated neighbors, and sends them on the corresponding links. If, however, more than one mobile software agent arrives at a node simultaneously, a leader is selected who then cleans the node, clones itself as many times as the number of contaminated neighbors, and sends them on the corresponding links; the other agents terminate here.

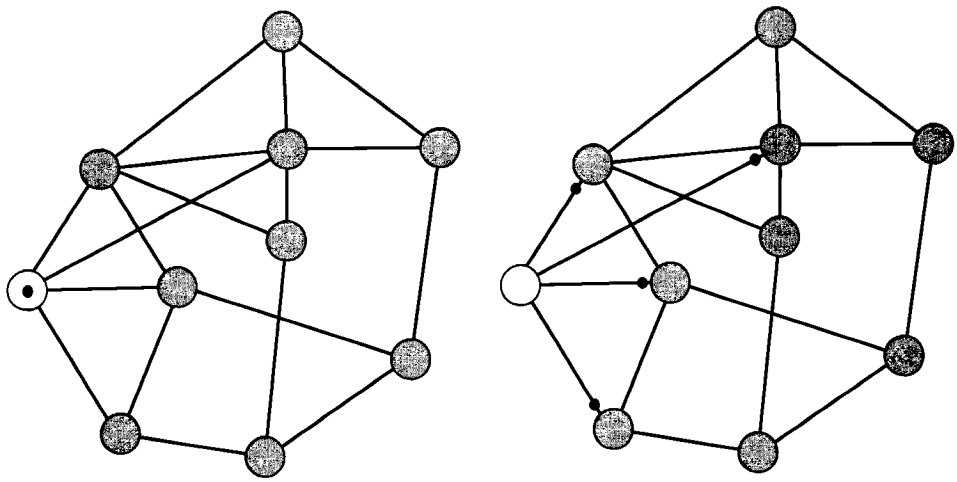
Protocol CLEAN (for an agent a arriving at node x)

If a is alone:
  Clean x.
  Check the state of neighbors.
  Let $N_{C(x)}$ be the set of contaminated neighbors of x.
  Clone $|N_{C(x)}|$ agents.
  Send the cloned agents to the contaminated neighbors.
If a is not alone:
  Locally choose a leader.
  If a is the leader:
    Clean x.
    Check the state of neighbors.
    Let $N_{C(x)}$ be the set of contaminated neighbors of x.
    Clone $|N_{C(x)}|$ agents.
    Send the cloned agents to the contaminated neighbors.
  Otherwise
    Terminate.

Figure 4.1: Protocol CLEAN

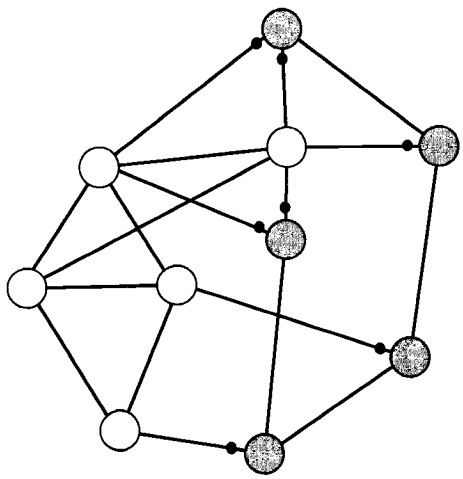
As has been proven in Chapter 3, Protocol CLEAN performs a monotone decontamination of the network.

As illustrated in Figure 4.2, by using visibility 1 for the agents, we reduce the maxi-



(a) step : 1 agents : 1

(b) step : 2 agents : 4



(c) step : 3 agents : 7

(d) Legend

Figure 4.2: Example of Agents Used in a Random Regular Graph with Visibility 1

mum number of agents required to decontaminate the network. All our examples in this chapter use synchronous networks, as we will study asynchronous networks in Chapter 5.

4.2 Visibility 2

The protocol CLEAN VISIBILITY 2, which uses visibility 2 as illustrated in Figure 4.3), is a variation of Protocol CLEAN. Initially, the mobile software agents are placed in arbitrary starting locations. From each starting location, the agents proceed in a BFS manner so as to ensure that no further re-contamination occurs. More precisely, when an agent arrives at a node, it cleans the node, clones itself as many times as the number of contaminated neighbors, and sends them on the corresponding links. However, every time an agent comes to clean node x it first checks to see if it is the only agent or one amongst many that can clean node x (in this manner we avoid the overuse of agents observed in Chapter 3). While in the first case the mobile agent would behave much like it did with visibility 1; in the second case it will first check if it is the leader of all the mobile software agents that can clean node x. If it is the leader, the agent then proceeds normally, otherwise it guards the current node for one time unit until node x is decontaminated. After one time unit has elapsed, all adjacent nodes being clean, the agents that were guarding nodes can then terminate.
We also note that, with visibility 2, since only one agent is sent to a node we do not need to locally select a leader in our algorithm.

Theorem 4.2: Protocol CLEAN with Visibility 2 performs a monotone decontamination of the network.

Proof: We want to prove by induction that once a node is clean, it will never be re-contaminated.

Basis. The starting locations of the agents are clean and since, by definition, enough agents are sent simultaneously to all their contaminated neighbors, they will not be re-

Protocol CLEAN VISIBILITY 2 (for an agent a arriving at node x)

If a is alone:

Clean x.

Check the state of neighbors.

Let $N_{C(x)}$ be the set of contaminated neighbors of x.

For each node y from $N_{C(x)}$

Let $N_{G(y)}$ be the set of guarded neighbors of y.

Select leader from $N_{G(y)} + x$

If a is the leader

Send clone to node y.

Else

Guard x.

Otherwise

Terminate.

Figure 4.3: Protocol CLEAN with Visibility 2

contaminated in the subsequent step. Moreover, the border nodes are all guarded (in this case border nodes are just single nodes).

Induction. We assume that at step k, some nodes are clean and the border nodes are guarded. At step $k+1$ the agents that are on the border nodes elected to be the leaders for specific adjacent contaminated nodes will clone themselves, by definition of the algorithm. Then the leaders clone themselves and send their clones to their assigned contaminated nodes. As all contaminated neighbors receive one agent and the other edges leading to decontaminated nodes are all guarded, the old border nodes become internal nodes. Therefore, these nodes will not be re-contaminated in a subsequent step, while the border node are all guarded.

By induction we have shown that once a node is clean, it will never be re-contaminated. Since the graph is connected, all nodes will eventually be decontaminated. □

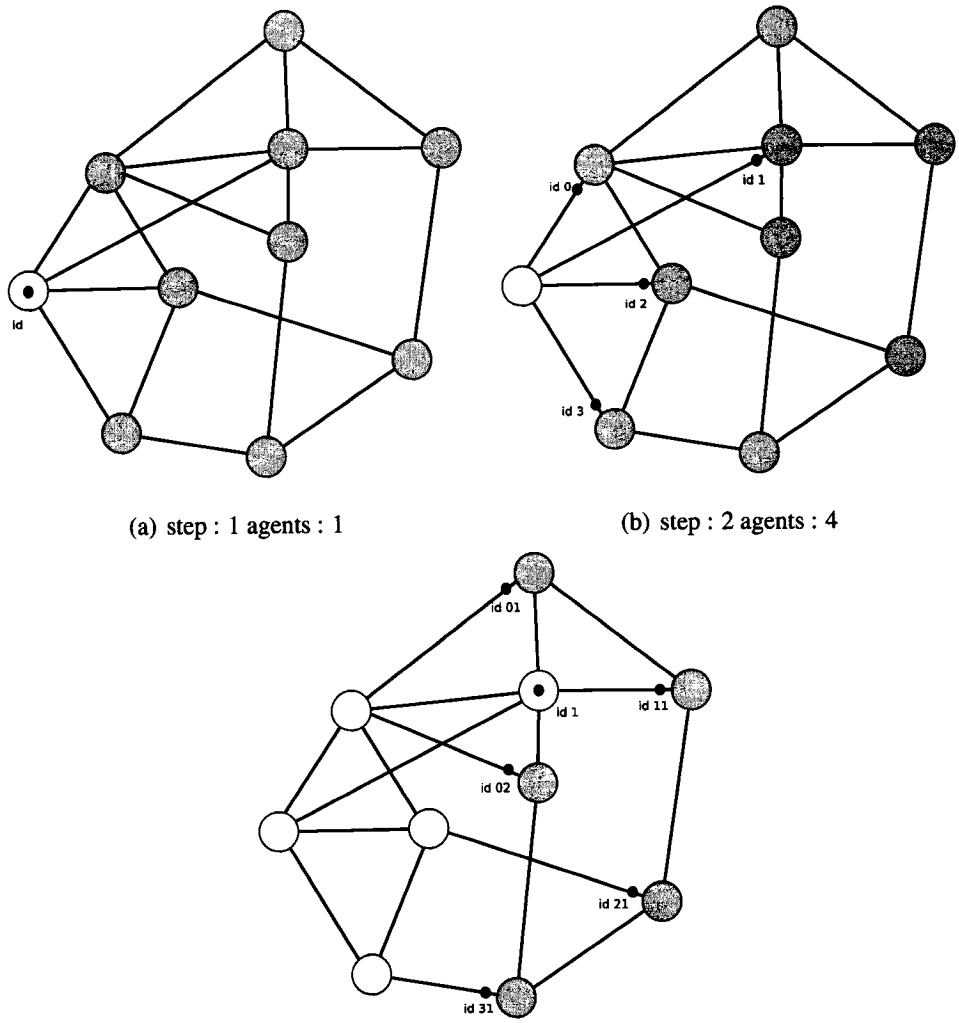
As illustrated in Figure 4.4, when the agents use visibility 2 (versus visibility 1) the maximum number of agents required to decontaminate the network is reduced. Further more when compared to visibility 1 we avoid the situation where multiple agents would reach the same node thus eliminating the "overuse" of agents described in Chapter 3.

4.3 Experimental Results

4.3.1 Observations and Analysis for d-regular Arbitrary Graphs with Visibility 1

As illustrated in Figure 4.6, when we increased the number of home bases, the total time required to do the decontamination is decreased. We can also observe the same behavior when increasing the degree of the nodes in the graph.

Figure 4.5 shows that the number of agents used, for d-regular arbitrary graphs with random starting locations, do in fact corroborate the results observed in Chapter 3. This



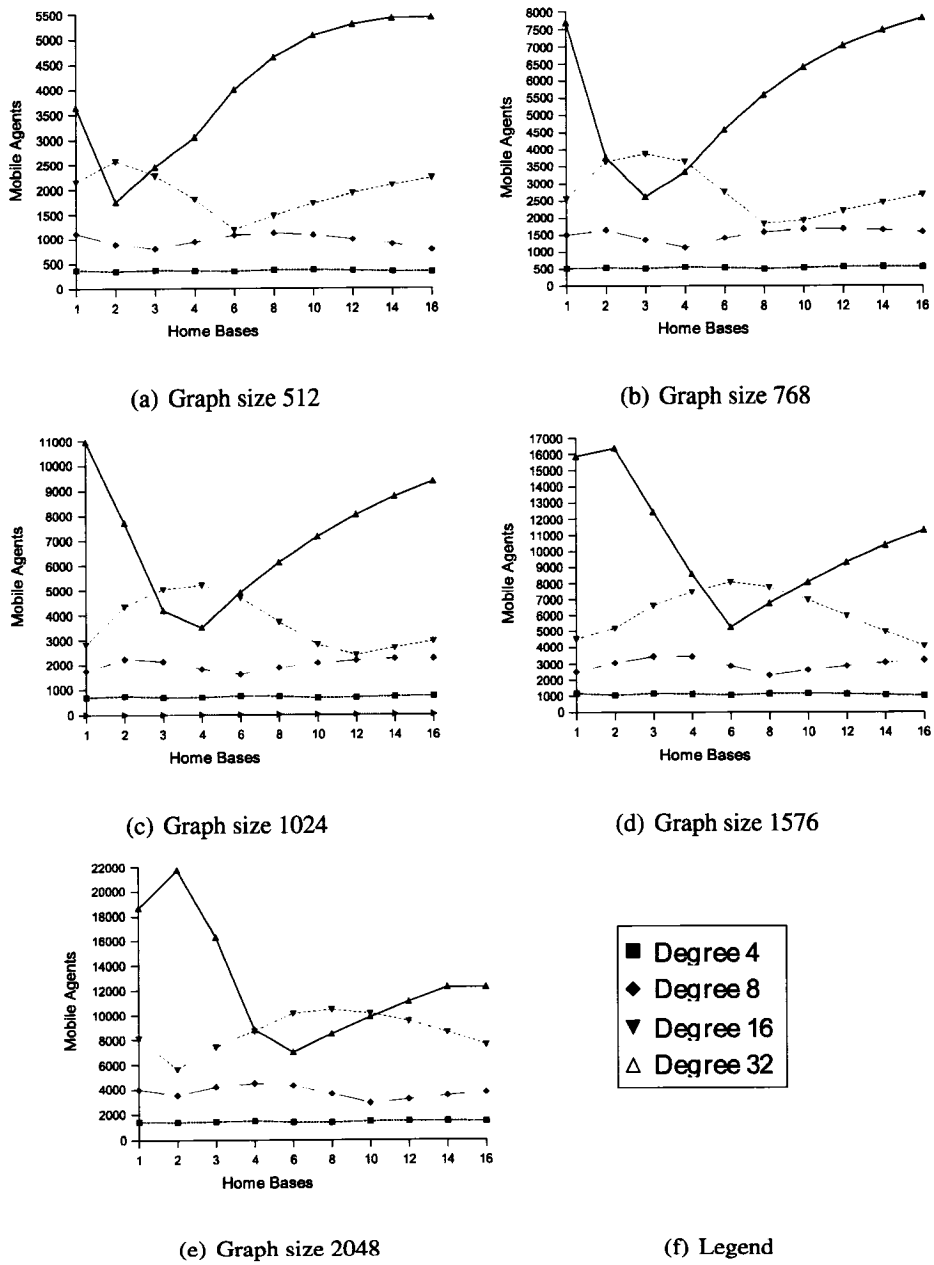
(a) step : 1 agents : 1

(b) step : 2 agents : 4

(c) step : 3 agents : 6

(d) Legend

Figure 4.4: Example of Agents Used in a Random Regular Graph with Visibility 2



(a) Graph size 512

(b) Graph size 768

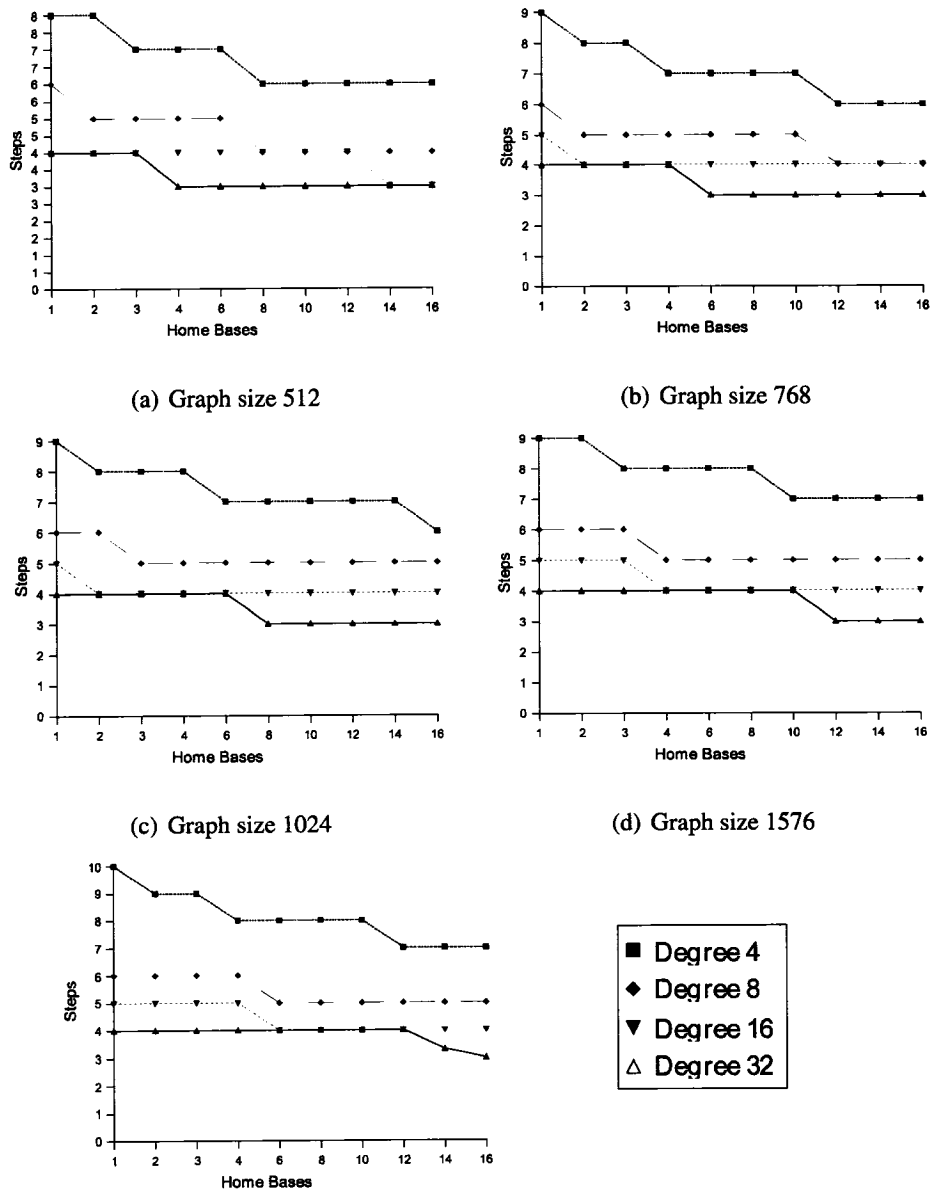
(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.5: Agents Used in d-regular Arbitrary Graphs with Visibility 1



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.6: Time Used in d-regular Arbitrary Graphs with Visibility 1

implies that the impact of multiple starting locations is, in fact, not dependant on the selection of a good or optimal combination of starting location. It should be noted though, that while we observe similar patterns in both experiments, the results obtained by the Genetic Algorithm are better. For example, the graph group $(768,32,2)$ in Figure 4.5 uses roughly 3500 agents with random starting locations, whereas for the Genetic Algorithm that number is around 1600 (as shown in Figure 3.3).

Similarly to what was observed in Chapter 3, the variation in the results of the number of agents used for the same graph group stays within a $\pm 10\%$ range. We can also see what that for certain number of home bases the number of agents required seems to be minimized. We also observe the same agents overuse with random starting locations. As explained in chapter 3, this is again due to the combination of the algorithm and the limited visibility.

4.3.2 Observations and Analysis for Arbitrary Graphs with Visibility 1

As illustrated in Figures 4.8 and 4.7, we observe the expected behavior for both the agents and the time used. In particular, we note that the average results are similar (except for a few rare minor differences), to what the results obtained for d-regular arbitrary graphs. For example, the graph group (512,16,3)in Figure 4.7 uses around 2500 agents whereas the same graph group in Figure 4.5 requires 100 less agents. To make sure that both sets of results were indeed correct, both experiments were run three separate times which produced similar sets of results each time. The variation of regular agents results stays within a $\pm 15\%$ range. Similar to to previous experiments, we observed that for certain number of home bases the number of agents used is minimized.

These results imply that the impact of multiple random starting locations, is not only independent of the choice of good or optimal starting locations, but also on the regularity

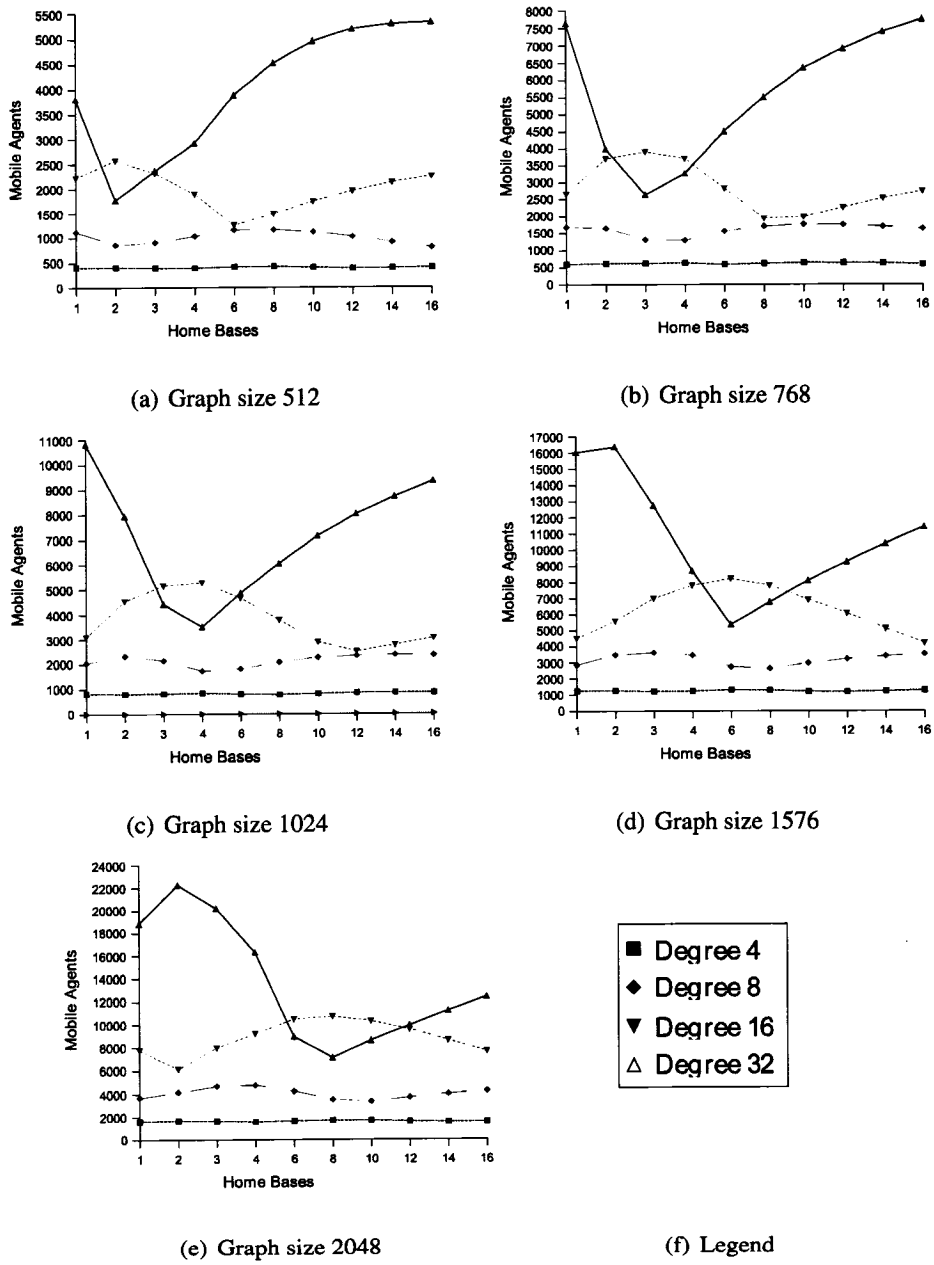
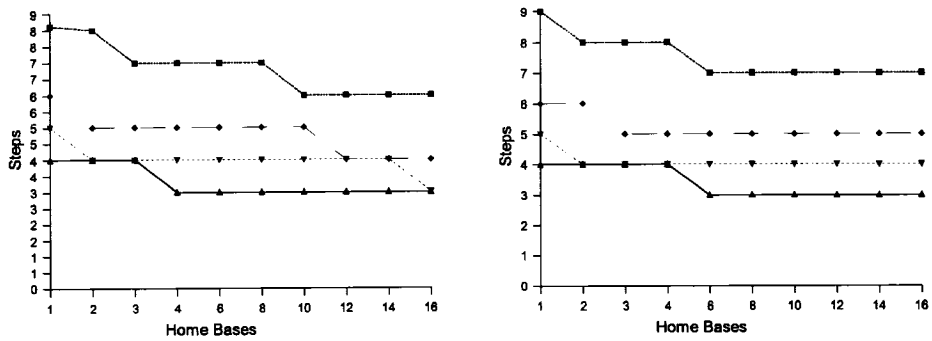
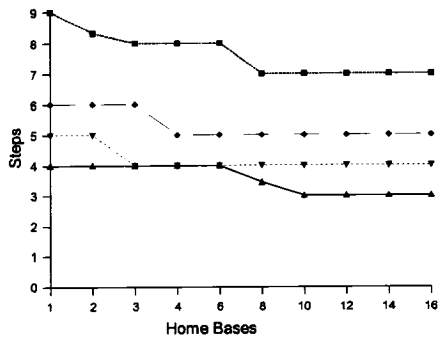


Figure 4.7: Agents Used in Arbitrary Graphs with Visibility 1

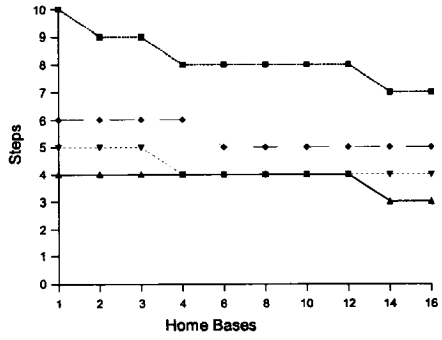


(a) Graph size 512

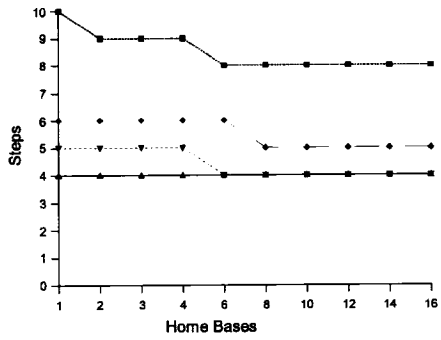
(b) Graph size 768



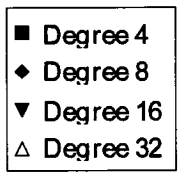
(c) Graph size 1024



(d) Graph size 1576



(e) Graph size 2048



(f) Legend

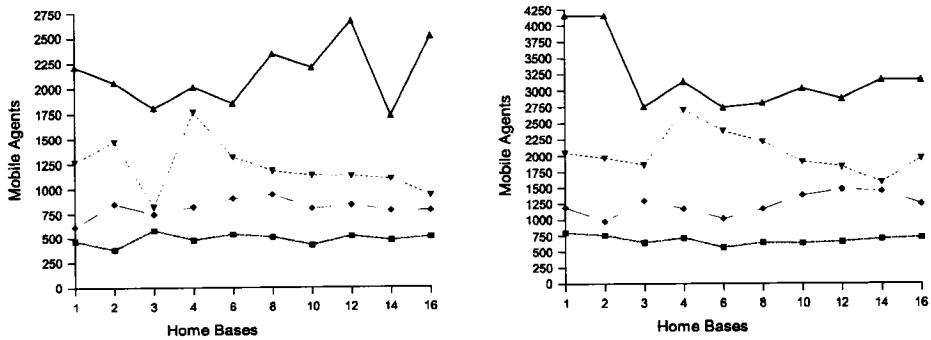
Figure 4.8: Steps Used in Arbitrary Graphs with Visibility 1

of the graphs. In fact, it seems that all that needs to be known is the size of the graph and the number of edges.

This raises an interesting question, as to how this *regularity* in the results is linked to the number of edges. We, therefore, devised another set of experiments whose results are presented in Figures 4.10 and 4.9. As we mentioned in Chapter 2, we made the following modification to the arbitrary graph generator: instead of using a constant number of edges as done previously, we use a fluctuating number of edges. To be more precise, the number of edges can be $\pm 20\%$ of $(n \times d)/2$, the number of edges in a regular graph of degree d and size n.

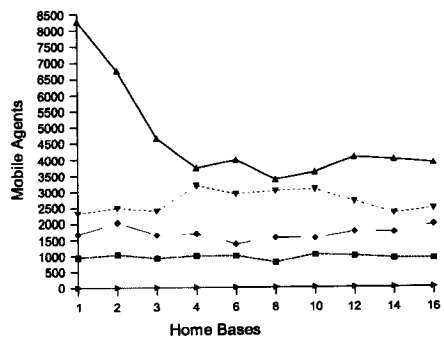
As expected, the time in Figure 4.10 is reduced by increasing the number of home bases. The number of time units used also decreases when higher number edges are used. However, we can observe some irregularities with the time (e.g., the graph group (512,16,3) uses one extra time unit on average in comparison with the graph group (512,16,2)). This is due to the number of time units that may vary as much as $\pm 56\%$ during a run (e.g., the graph group (512,4,8) had a maximum of 125 and a minimum 65 of time units used in a single series of runs, but the average time for the same graph group was around 5 time units).

Figure 4.9 shows that the number of agents used do not corroborate the results observed previously and in Chapter 3. This is due to large variations up to $\pm 90\%$ in the results for the same graph groups (for example, for the graph group (2048,32,16) results vary from 986 to 13014 with an average of 8000). This leads us to hypothesize that "the patterns in the results for multiple starting locations are due to the density of the edges in the graph". We cannot conclusively tell whether or not with certain number of starting locations the number of agents required is minimized.

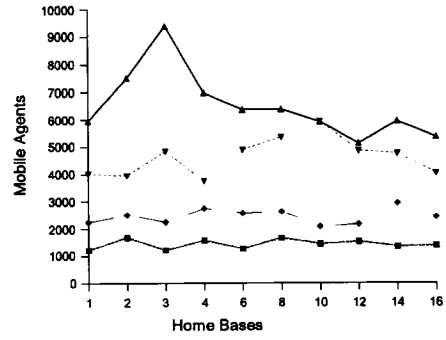


(a) Graph size 512

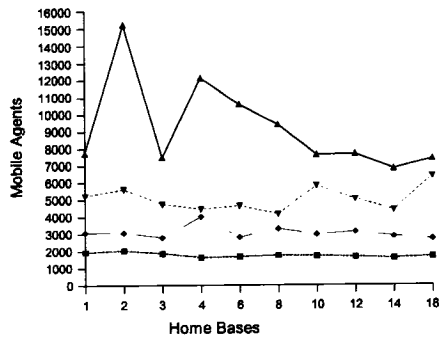
(b) Graph size 768



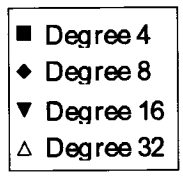
(c) Graph size 1024



(d) Graph size 1576

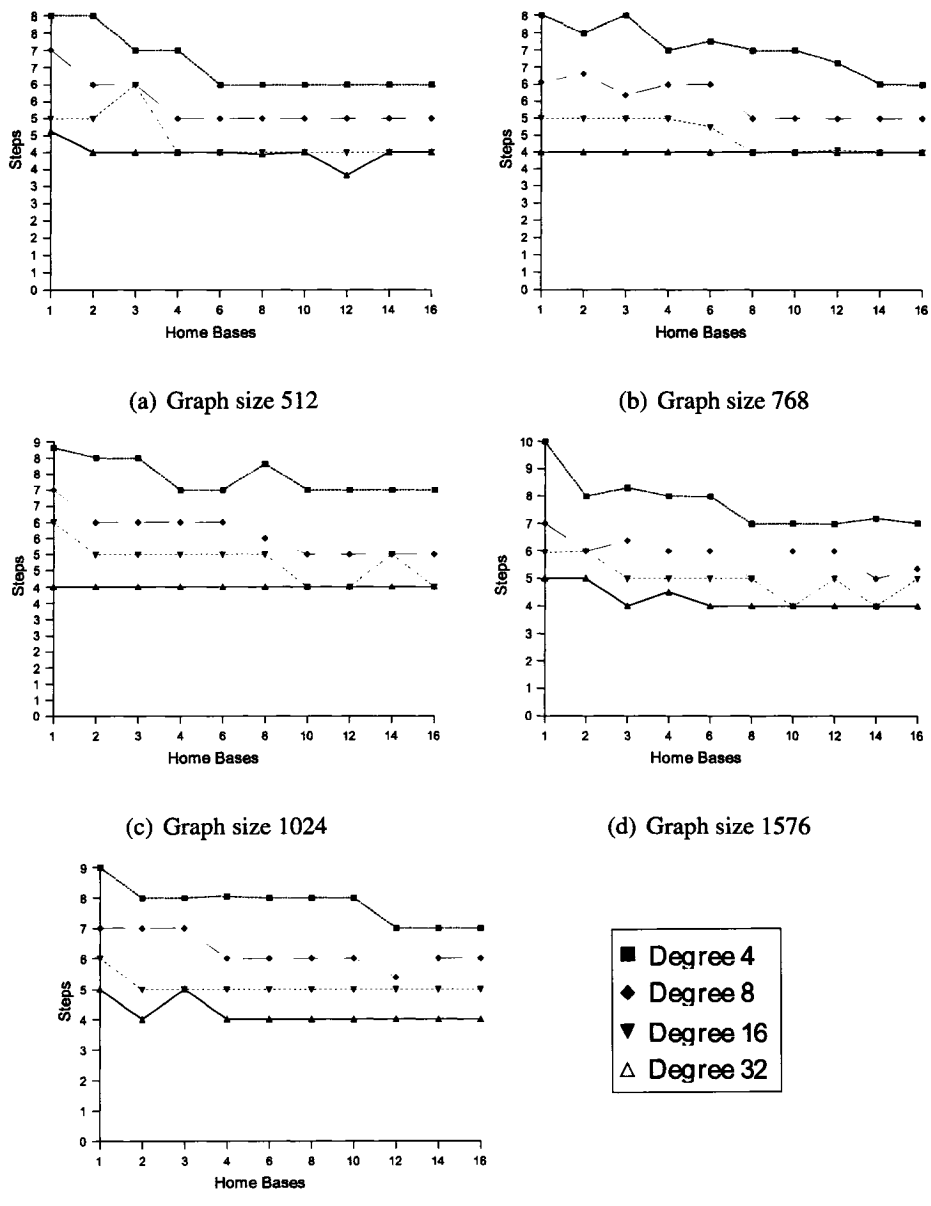


(e) Graph size 2048



(f) Legend

Figure 4.9: Agents Used in Arbitrary Graphs with Visibility 1 and fluctuating number of edges



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.10: Steps Used in Arbitrary Graphs with Visibility 1 and fluctuating number of edges

4.3.3 Observations and Analysis for d-regular Arbitrary Graphs with Visibility 2

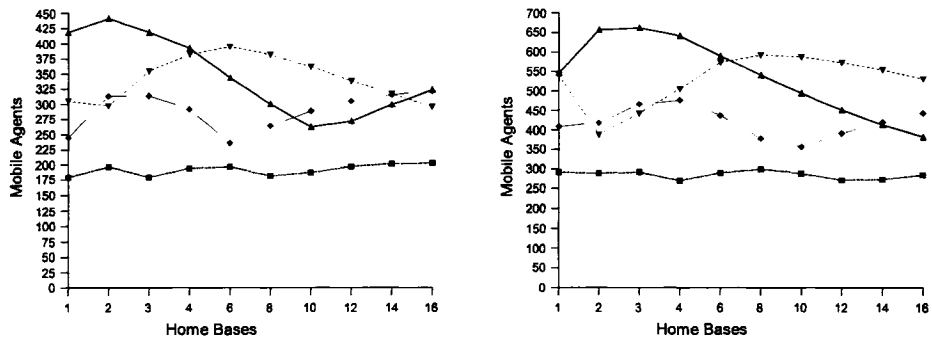
One of our main goals is to determine how visibility 2 impacts the results in this case of d-regular arbitrary networks. We can see in Figure 4.12 that the time decreases with the increase in the number of home bases. Also, the same is observed when increasing the degree for each node in the graphs.

As stated earlier, visibility 2 greatly increases the autonomy of the agents. In Figure 4.11, we observe that we achieved our main goal to avoid the overuse of agents with visibility 2, as we use fewer agents than the number of nodes in all cases when compared to the results we obtained with visibility 1 in Figure 4.5. The overuse of agents that we previously observed was the result of the combination of visibility 1 with a BFS strategy. As expected with visibility 2, the overuse of agents is virtually eliminated.

The variation of random regular agents is within a $\pm 10\%$ range. We observed that, similar to visibility 1, the results seem to indicate there exists certain number of starting locations where the number of agents used at any time is minimized for each graph family.

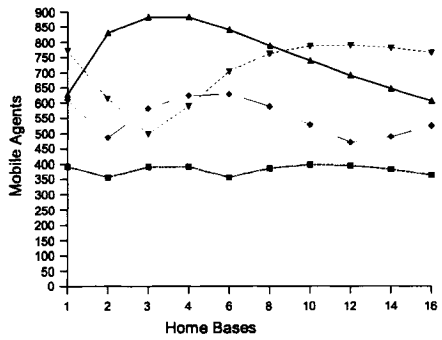
4.3.4 Observations and Analysis for Arbitrary Graphs with Visibility 2

In these experiments, we also address visibility 2 in arbitrary networks. The number of time units required in Figure 4.14 decline as expected with the increase in the number of home bases. The number of time units decreases as expected with the increase in the number of edges. As seen in Figure 4.13, once again we can observe that we have managed to avoid the overuse of agents with the visibility 2, as we use fewer agents than the number of nodes in all runs. It was also observed that with visibility 2, there is a certain number of starting locations for each graph family where the number of agents used is minimal. Hence, much like visibility 1 we can again hypothesize that this observed regularity in

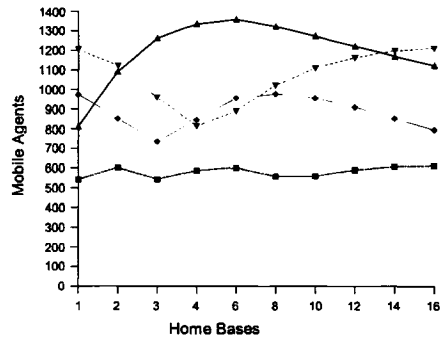


(a) Graph size 512

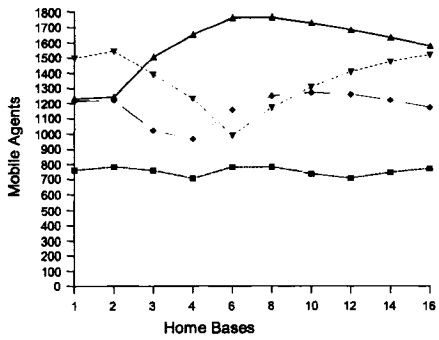
(b) Graph size 768



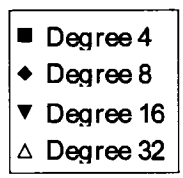
(c) Graph size 1024



(d) Graph size 1576

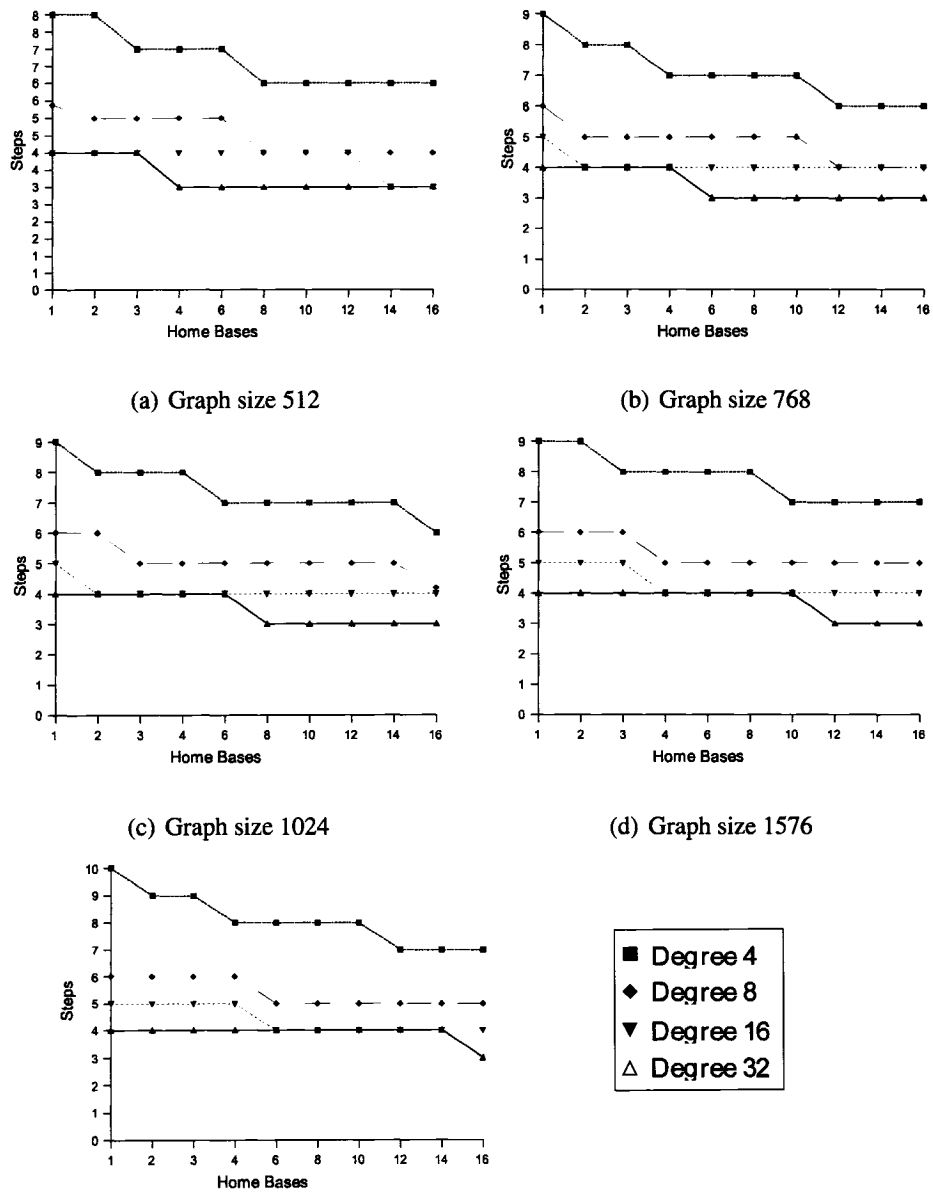


(e) Graph size 2048



(f) Legend

Figure 4.11: Agents Used in d-regular Arbitrary Graph with Visibility 2



(a) Graph size 512

(b) Graph size 768

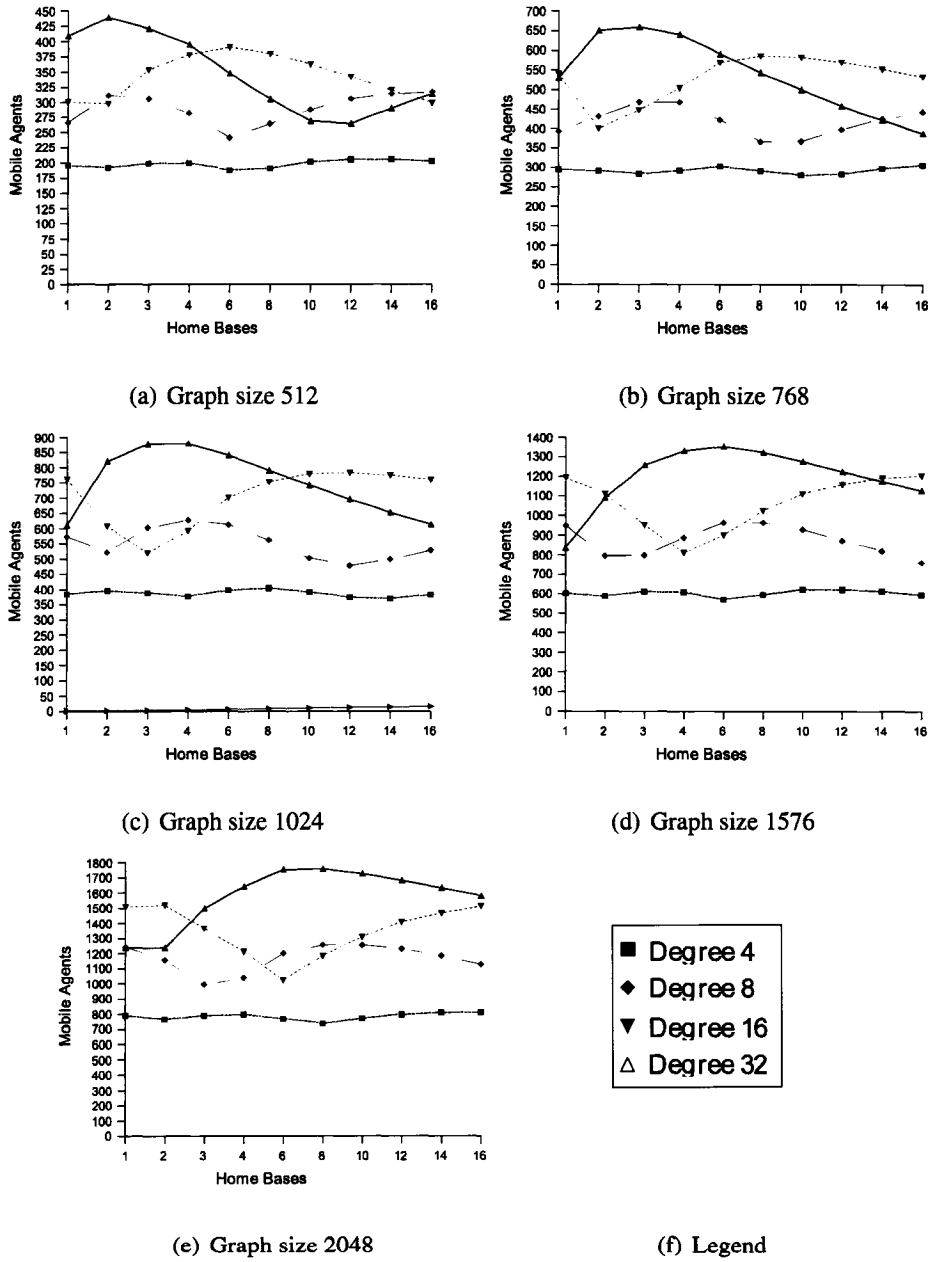
(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.12: Steps Used in d-regular Arbitrary Graphs with Visibility 2



(a) Graph size 512

(b) Graph size 768

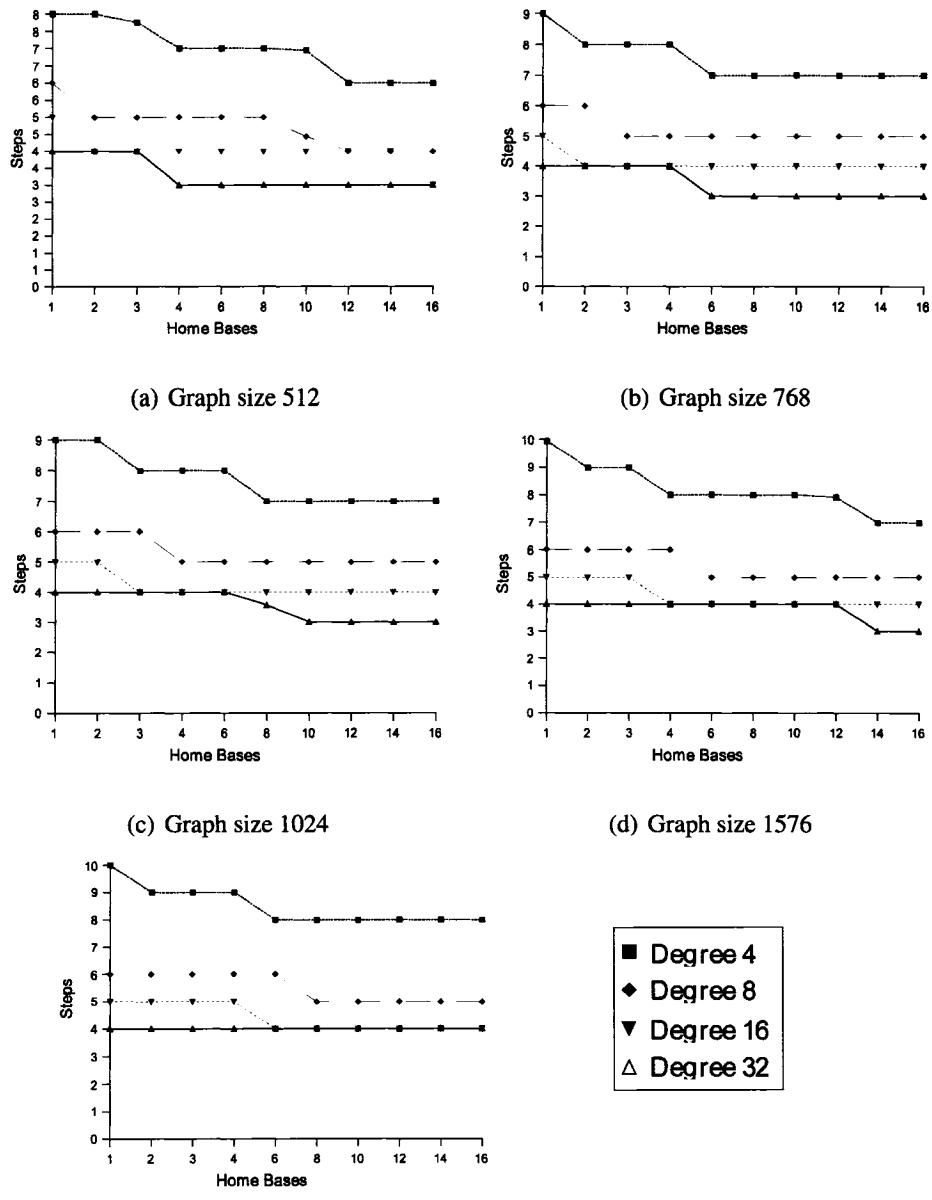
(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.13: Agents Used in Arbitrary Graphs with Visibility 2



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.14: Time Used in Arbitrary Graphs with Visibility 2

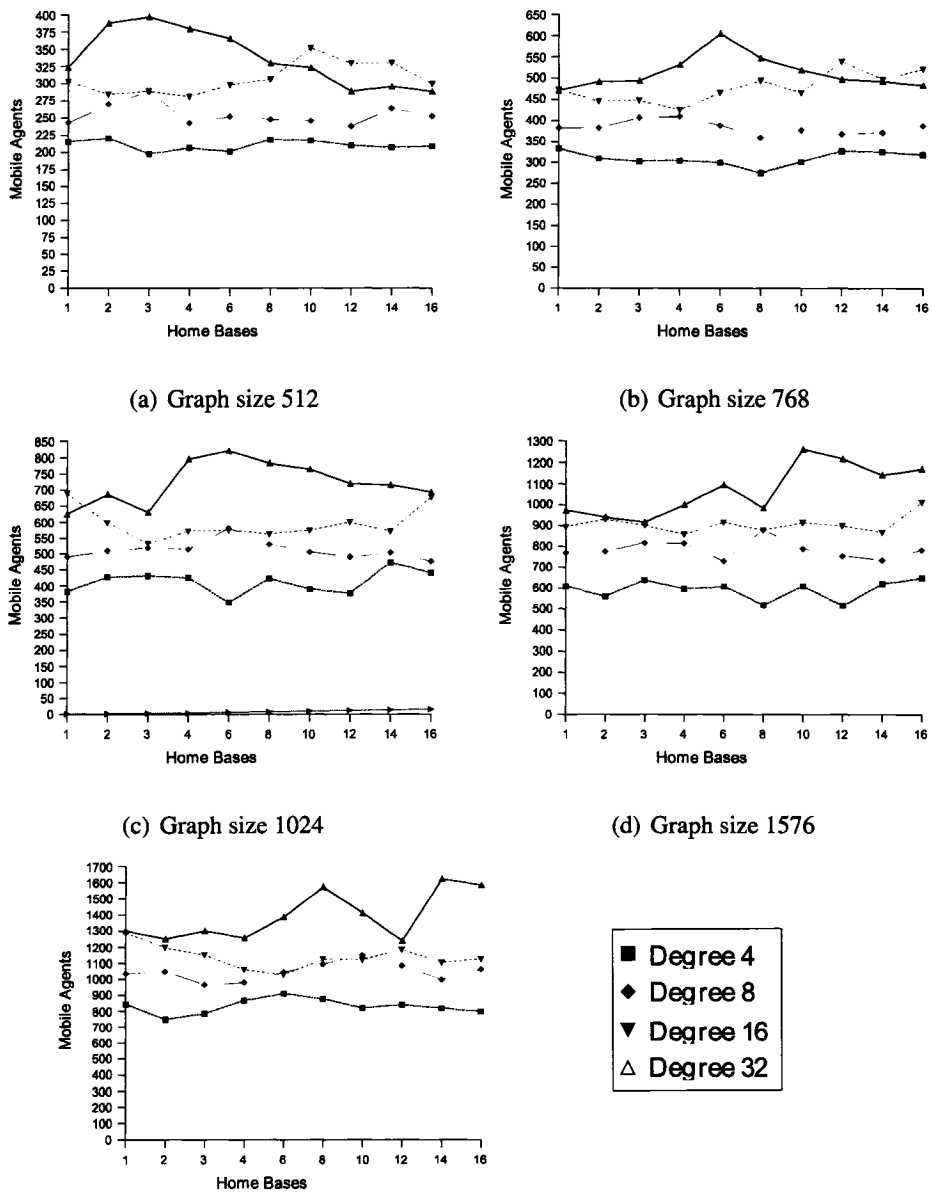
the results does not come from the regular degree but rather from the constant number of edges in the graph. The variation in the results for the agents, in each graph group, is within a $\pm 20\%$ range of the average result.

We also want to verify if the regularity in the results we observed with visibility 2 is due to the constant number of the edges in the graph. For this, we modified our graph generation algorithm (as described in Chapter 2) to use a fluctuating number of edges instead of a constant number. In order to see the impact of the change obtained, the fluctuations in the number of edges was limited to $\pm 20\%$ of $(n \times d)/2$ (n being the number of vertices and d the number of edges in the graph).

As expected, time units in Figure 4.16 decreased with an increasing number of home bases. The number of time units also decreased with an increasing number of edges. Similarly, to what we observed with visibility 1, there were still some irregularities in the time used. These were due to the difference in the time used for each run of a particular graph group. Figure 4.15 shows that while the agents results do not corroborate the results previously observed for visibility 2 due to major anomalies. The variation in the results for agents stays within a $\pm 50\%$ range (for example, for the graph group (2048,32,16) the results vary from 1000 to 1600). In addition to this, we are not able to conclusively tell if the number of agents required is minimized for certain number of starting locations.

4.4 Summary

In this chapter, we have studied arbitrary and d-regular arbitrary synchronous networks with both visibility models. In particular, we have verified that the patterns we initially observed in Chapter 3 with visibility 1 are, in fact, not linked to the selection of good starting locations nor to the regularity of the graphs. We can hypothesize that these patterns we have observed require only that the number of edges and vertices be constant, and depend mostly on the density of the edges in the graph. In addition to this, visibility 2 not only solves the overuse of agents but also reduces the maximum number of agents required in all cases. Finally, as previously observed we have again seen that the time decreases when



(a) Graph size 512

(b) Graph size 768

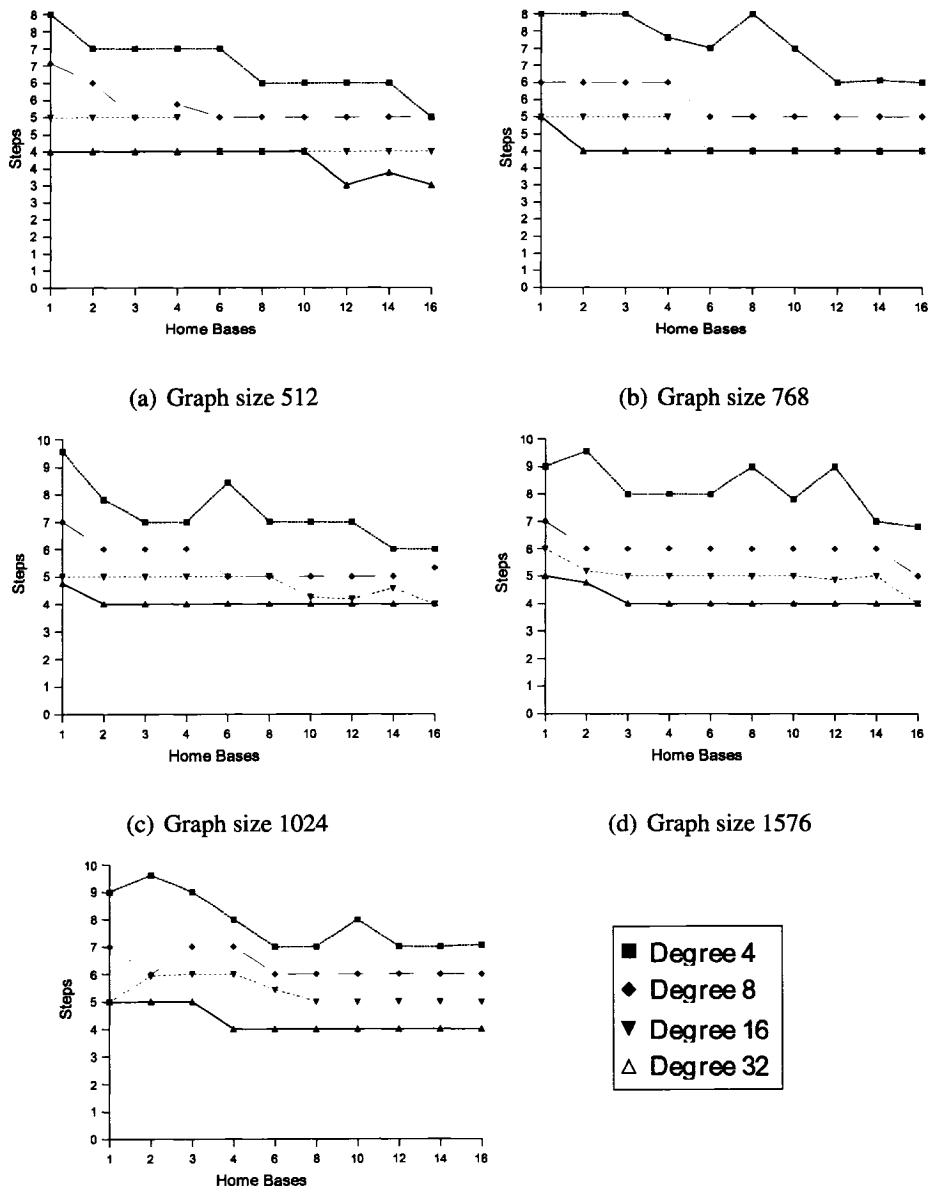
(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.15: Agents Used in Arbitrary Graphs with Visibility 2 and fluctuating number of edges



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 4.16: Time units Used in Arbitrary Graphs with Visibility 2 and fluctuating number of edges

more home bases are used.

In Chapters 3 and 4, we have only considered synchronous networks. This leaves an interesting question, as to how our algorithms and visibility models will behave in asynchronous networks. In Chapter 5 we will see the limitations of our synchronous algorithms and how they can be modified to work in asynchronous networks.

CHAPTER 5

DECONTAMINATION OF ASYNCHRONOUS NETWORKS

In our final set of experiments, we consider the decontamination problem in asynchronous networks with both visibility 1 and 2. We will first describe the limitations in asynchronous networks for both visibility models. We will then consider modifications to our model in an attempt to improve the performance of our algorithms. Since our algorithms are based on the BFS, they can be applied to any arbitrary topology with an arbitrary number of starting locations. Their efficiency depends on the number of starting places and their locations.

An interesting question is: given a network and a number of starting locations, are we going to observe similar patterns in asynchronous networks as we have in Chapters 3 and 4? In order to study this question, we resorted to simulations in asynchronous arbitrary networks. Through these experiments, we show that when we increase the number of home bases the number of agents required decreases in all cases. But we also observe that there are significant drawbacks when our algorithms are used in asynchronous networks.

5.1 Limitations in Asynchronous Networks

While initially testing asynchronous networks we quickly came to observe the following limitations with the models used for the agents.

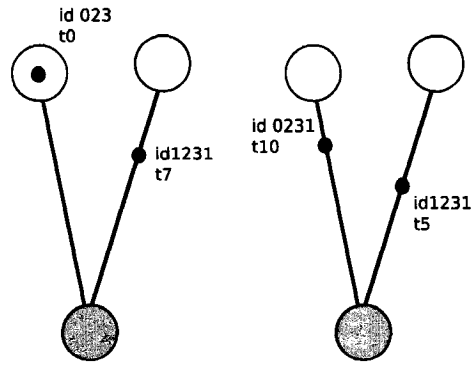
5.1.1 Limitation of Visibility 1

In observing the algorithm for visibility 1 in case of asynchronous networks, we noticed that there would be an occurrence of multiple agent collision similar to what we have seen

with the local knowledge model in Chapter 2.

For example, consider three contaminated nodes x, y and z in a synchronous network where the neighbors of z are x and y. If node x is decontaminated a step before node y, then node z will be decontaminated at the same time as node y. But as seen in Chapter 3, we still end up with a situation where multiple agents are being sent to a single node. And as shown in Chapter 3, this happens more often towards the final steps of the algorithm in synchronous networks.

In asynchronous networks, there is a major limitation with this algorithm, as illustrated in Figure 5.1. In this example of an asynchronous network, two nodes x and y have both been cleaned, and both have the same neighbor z which is still contaminated. When x has been cleaned an agent is sent from x to z. When y has been cleaned, because z is still contaminated, an agent will also be sent from y to z. Thus, we have more situations like this where several agents are sent to decontaminate the same node. This increases the maximum number of agents used to decontaminate the network. Contrary to visibility 2, there is no way to detect whether another agent has been sent to clean a node or not.



(a) step 1 (b) step 2

Figure 5.1: Example of the limitation with visibility 1

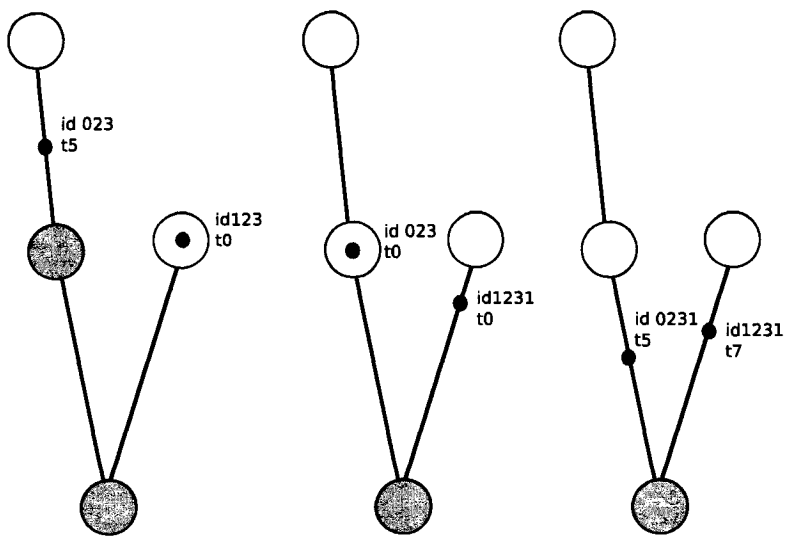
To be more precise, in Figure 5.1 we have 3 nodes and two agents 023 and 1231. Agent

1231 is in transition between nodes y and z, and agent 023 has just cleaned node x. As agent 023 arrives at node x it verifies if z is clean. Since agent 1231 is in transition between nodes y and z, agent 023 sees that node z is still contaminated and has no way of knowing that the node is going to be cleaned. Therefore, agent 023 clones itself and sends its clone 0231 to node z.

This situation is what we are attempting to avoid. It is provoked by a shortcoming in the synchronous algorithm, where it is assumed that all agents take exactly 1 time unit to cross an edge between two nodes. Because of the limitations of visibility 1, there is no possibility to modify the algorithm to make it suitable for asynchronous networks. As a result, we did not conduct any experiment using visibility 1 in asynchronous networks.

5.1.2 Limitation of Visibility 2

The algorithm used for visibility 2 in synchronous networks fails to work in the asynchronous networks as illustrated by Figure 5.2).



(a) step 1 (b) step 2 (c) step 3

Figure 5.2: Example of the limitation with visibility 2

This failure is due to a limitation in the algorithm for visibility 2. For example, let us assume, as seen in Figure 5.2, that we have 4 nodes w,x,y,z and two agents 023 and 123. Agent 023 is in transition between nodes w and x; agent 123 has just cleaned node y, and the bottom most node is node z. At this point, agent 123 realizing that it is the only agent neighboring node z, clones itself and sends its clone 1231 to node z.
When 023 arrives at node x, it verifies if z is clean and as it is the leader of all the agents neighboring node z (since 123 has terminated itself and 1231 is in transition between nodes y and z), it clones itself and sends its clone 0231 to node z.

The latter situation is what we are attempting to avoid. It is provoked by a shortcoming in the synchronous algorithm where all agents take exactly 1 time unit to cross an edge between two nodes. Thus preventing this kind of agent overuse, since either a node is already cleaned if an agent was sent earlier or only one agent is sent from all the neighboring guarded nodes.

In order to avoid this pitfall, we devised a modification to the algorithm that would check if an agent has already been sent to a given node. This modification also required a further change to ensure that an agent is required to guard the node until all its neighbors have been cleaned. In the synchronous network, an agent assigned to guard a node only needs to wait for one time unit. This modification is presented in the following section.

5.2 Modified Algorithm for Asynchronous Networks

The cleaning strategy Protocol CLEAN VISIBILITY 2 as illustrated in Figure 5.3, is a minor modification of Protocol CLEAN. Initially, the agents are placed in arbitrary starting locations. Each starting agent will try to move its clones along the breadth-first-search (BFS) tree of the network rooted at its starting position. More precisely, at each step, when an agent arrives to a node, it cleans the node, clones itself as many times as the number of contaminated neighbors, and sends them on the corresponding links. However,

every time an agent comes to clean node x, it first checks to see if it is the only agent or one amongst many that can clean node x. If it is the only agent who can clean the node, it would behave much like it did with visibility 1; otherwise, it will check if it is the leader of all the agents that can clean node x. If it is the leader, it then proceeds normally, otherwise it guards the current node until all neighboring contaminated nodes are cleaned.

Theorem 5.1: Protocol CLEAN with visibility 2 for asynchronous networks performs a monotone decontamination of the network.

Proof: We want to prove by induction that once a node is clean, it will never be re-contaminated.

Basis. The starting locations of the agents are clean. Since, by definition of the algorithm, enough agents are sent simultaneously to all their contaminated neighbors they will not be re-contaminated in the subsequent step. Moreover, the border nodes are all guarded (in this case, border nodes are just single nodes).

Induction. We assume that at step k, some nodes are clean and the border nodes are guarded. At step $k+1$, the agents that are on the border nodes elected to be leaders for specific adjacent contaminated nodes will clone themselves, by definition of the algorithm. Then, the clones proceed to their assigned contaminated nodes. As all contaminated neighbors receive one agent and the other edges leading to decontaminated nodes are all guarded, the old border nodes become internal nodes. Therefore, these nodes will not be re-contaminated in a subsequent step, while the border node are all guarded.

By induction we have shown that once a node is clean, it will never be re-contaminated. Since the graph is connected, all nodes will eventually be decontaminated in subsequent steps. □

Protocol CLEAN VISIBILITY 2 FOR ASYNCHRONOUS (for a agent a arriving at node x)

If a is alone:

  Clean x.

  Check the state of neighbors.

  Let $N_{C(x)}$ be the set of contaminated neighbors of x.

  For each node y from $N_{C(x)}$

    Let $N_{D(y)}$ be the set of decontaminated neighbors of y.

    Let $N_{G(y)}$ be the set of guarded neighbors of y.

    If there exists a node z belonging to $N_{D(y)}$ but not $N_{G(y)}$:

      Set status to Guard.

    Else

      Select leader from $N_{G(y)} + x$

      If a is the leader

        Send clone to node y.

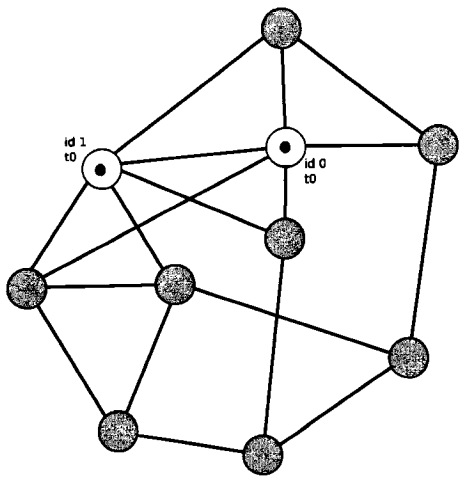
      Else

        Set status to Guard.

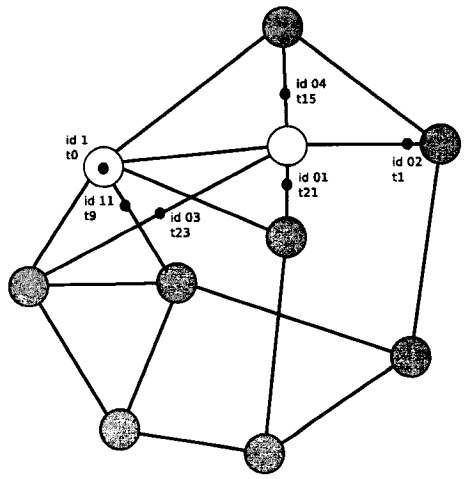
Otherwise

  Terminate.

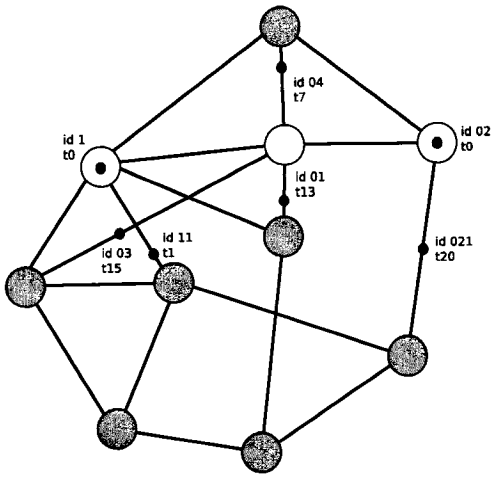
Figure 5.3: Protocol CLEAN with Visibility 2 for asynchronous networks



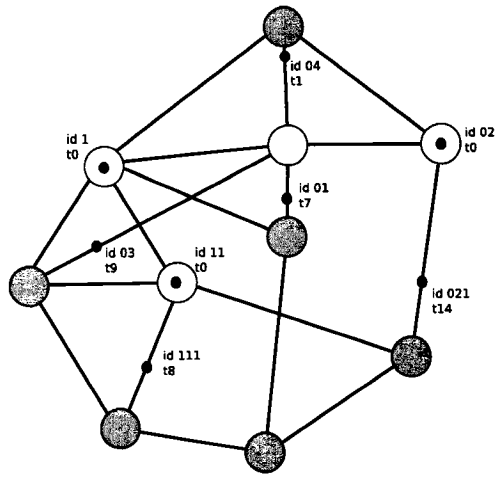
(a) time : 0 agents : 2



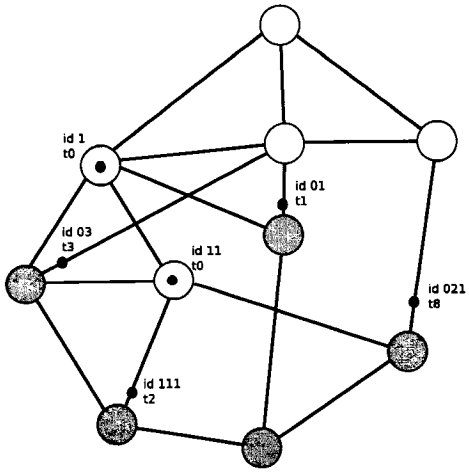
(b) time : 6 agents : 6



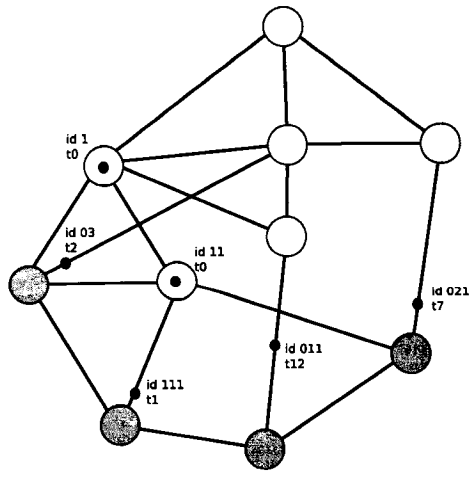
(c) time : 14 agents : 7



(d) time : 20 agents : 8

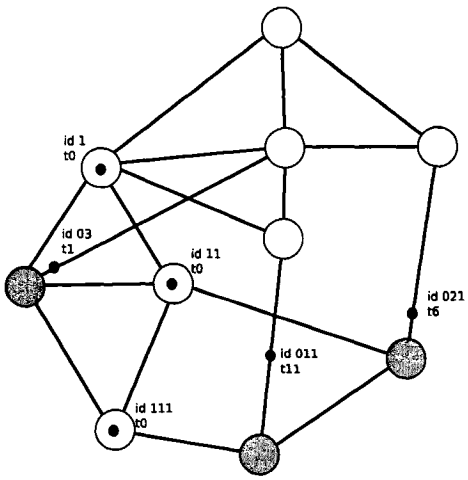


(e) time : 26 agents : 6

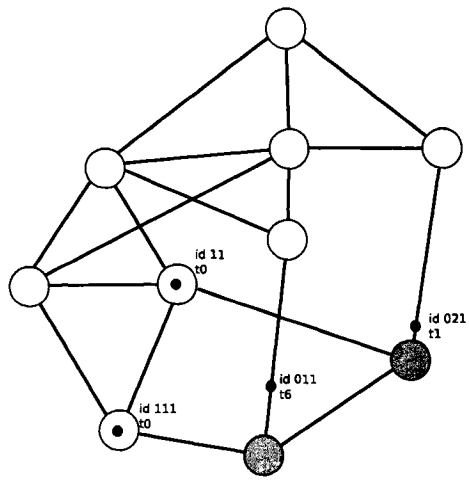


(f) time : 27 agents : 6

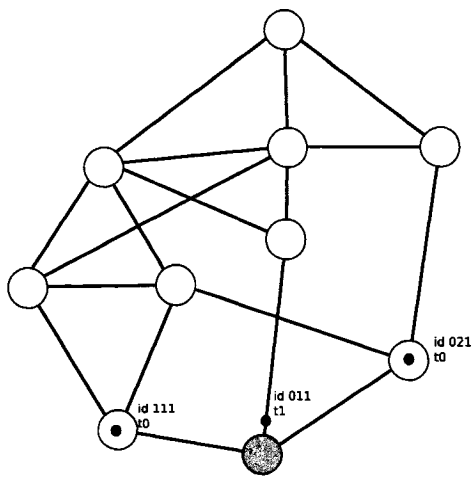
Figure 5.4: Example of Agents Us60 in d-regular Arbitrary Graphs with Visibility 2



(a) time : 28 agents : 6



(b) time : 33 agents : 4



(c) time : 38 agents : 3

(d) Legend

Figure 5.5: Example of Agents Used in a d-regular Arbitrary Graphs with Visibility 2 (Cont.)

5.3 Experimental Results

5.3.1 Observation and Analysis for Asynchronous d-regular Arbitrary Graph with Visibility 2

Time decreases, as expected, with an increasing number of home bases. Also the time decreases with an increasing degree for each node. The variation of agents stays within a $\pm 10\%$ range. Unlike with synchronous networks, we cannot tell if there exists a certain number of home bases for which the number of agents is minimized, as the results obtained form a plateau.

In asynchronous networks, we observe undesirable saturation of the number of agents used. This is due to the fact that many of the agents wait until all of their neighbors are decontaminated.

As we can observe in Figure 5.8, the number of agents waiting increases steadily as the number of agents in transition increases. This can also be observed in Figures 5.4 and 5.5. The agents spread in multiple directions in an arbitrary manner due to a lack of coordination. Because of this lack of coordination, we end up with a large number of agents that guard their nodes, while the neighbors remain contaminated. As there is no balance between new agents being cloned and old ones being able to terminate, the total number of agents used will increase until no new contaminated nodes are discovered. As we prohibit the algorithm of sending more than one agent per node, when no new contaminated nodes are discovered, no new agents will be sent out. When no new contaminated nodes are discovered, the number of agents in transition declines as they clean their respective nodes, this in turn causes the number of guarding agents to decline since they can terminate. The combination of both these declines produces the drop we observe in the Figure 5.8.

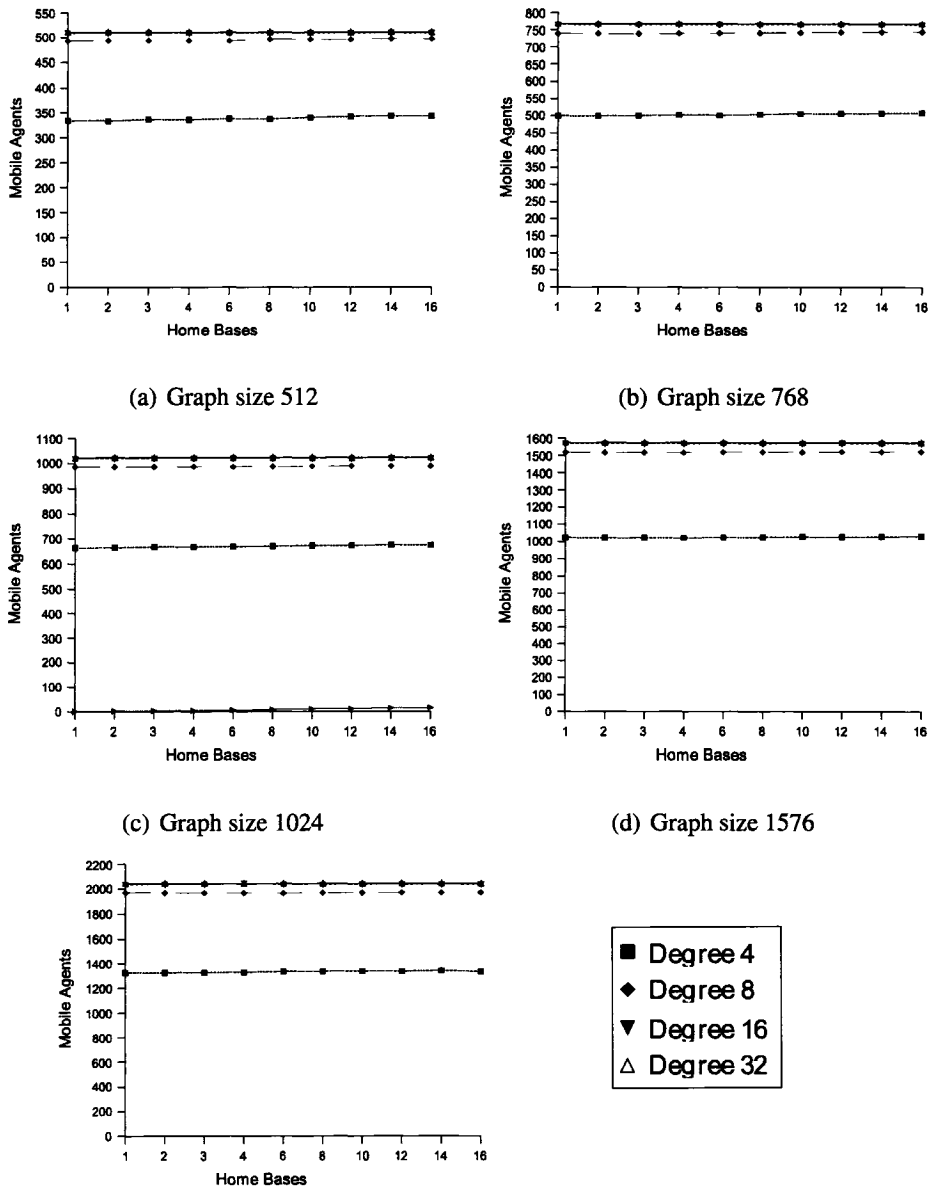
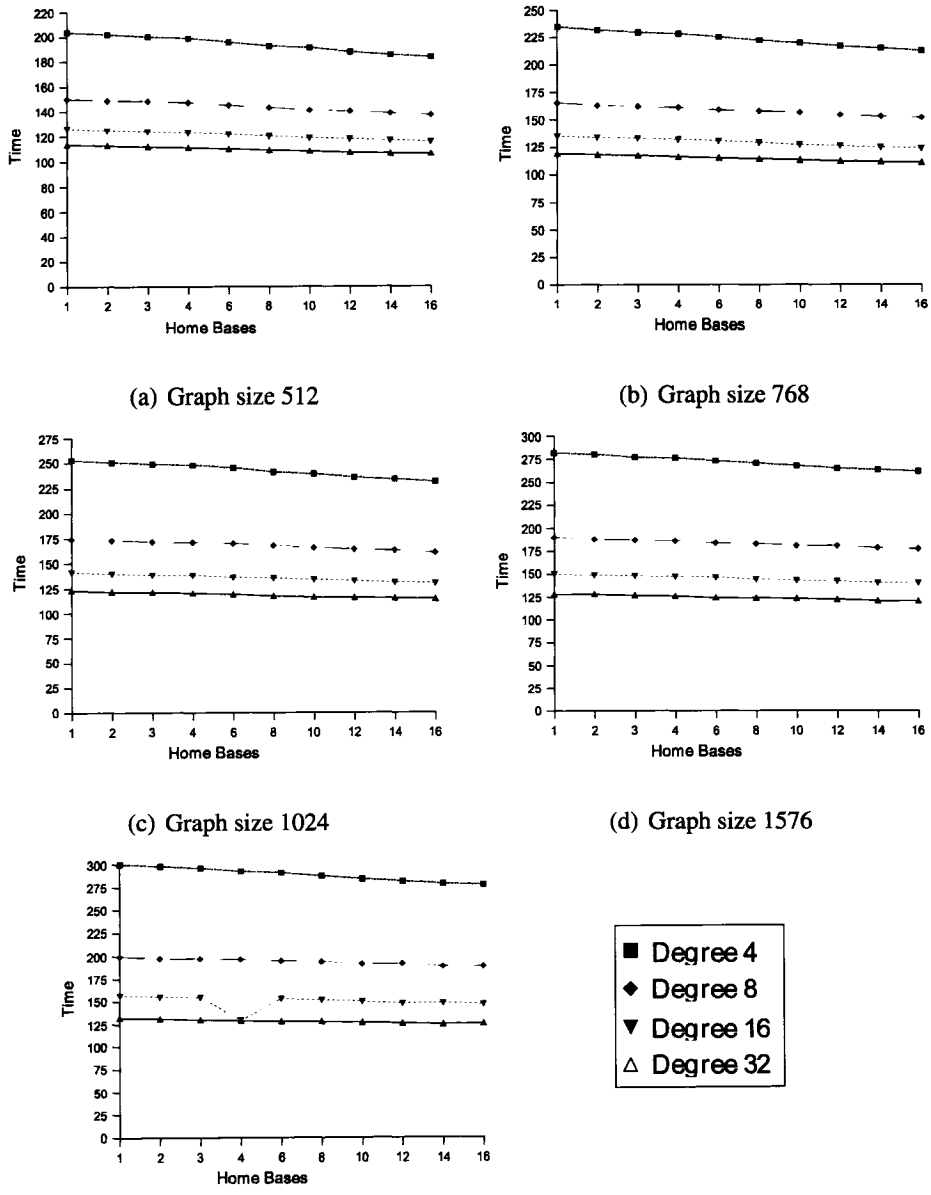


Figure 5.6: Agents Used in d-regular Asynchronous Arbitrary Graphs with Visibility 2



(a) Graph size 512

(b) Graph size 768

(c) Graph size 1024

(d) Graph size 1576

(e) Graph size 2048

(f) Legend

Figure 5.7: Time Used in d-regular Asynchronous Arbitrary Graphs with Visibility 2

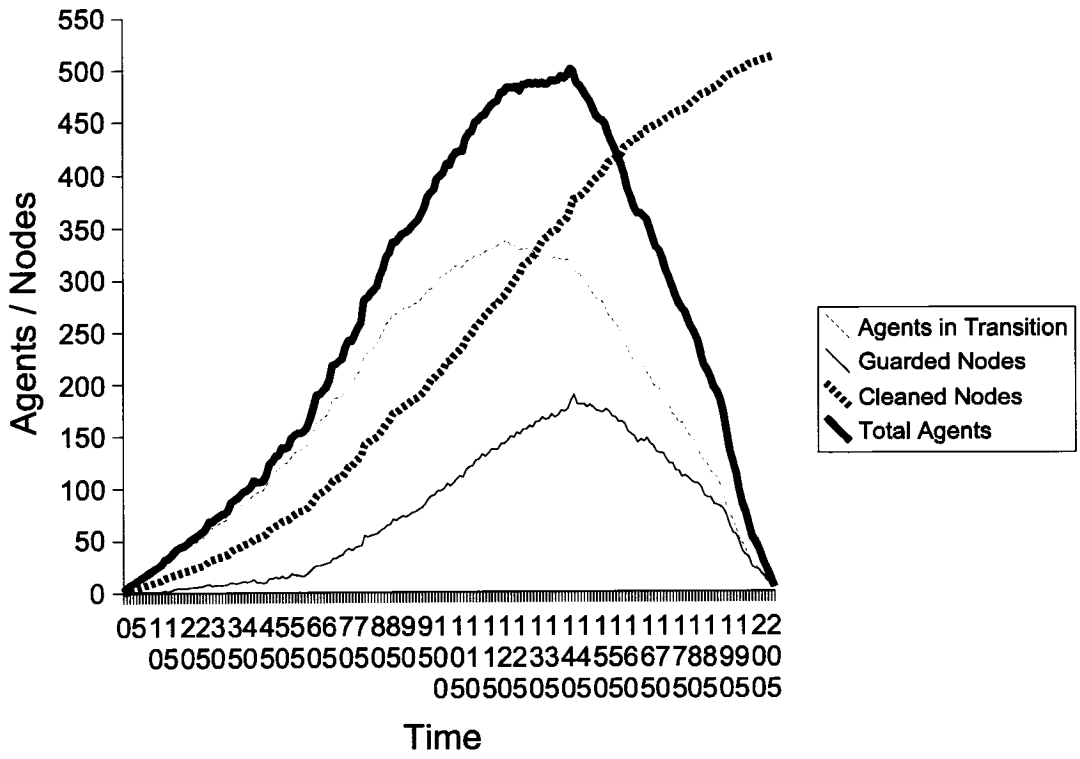


Figure 5.8: Use of agents in an Asynchronous network for graph size 512 homebases 1 and average degree 8

5.4 Summary

In this chapter, we have considered both arbitrary and d-regular arbitrary asynchronous networks with both visibility models. In particular, we have studied the limitations of our algorithms for both visibility models in asynchronous networks. While nothing could be done for visibility 1, we proposed a straight forward modification for our visibility 2 algorithm. As we had previously observed, the increase in the number of starting locations decreases the time. However, the overuse of agents observed in the results obtained revealed the limitations of our algorithm. In a practical application such as wireless sensor networks or distributed databases, this overuse of agents is undesirable as it might disable a considerable segment of the network.

The results observed in this chapter gave us a better understanding of how visibility models and cloning can be used in the Decontamination Problem for asynchronous arbitrary networks. At the same time, we are left with new interesting problems to figure out if there exists a better algorithm that can be used in arbitrary topologies and how to provide a form of coordination in asynchronous networks.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis, we have studied strategies for the decontamination problem in both synchronous and asynchronous arbitrary networks. These strategies have proven to be efficient in terms of time, and have yielded interesting results for agents. Through experiments, we have demonstrated that the combination of multiple starting locations and different levels of visibility can in certain cases further reduce the number of agents and time required to decontaminate the network than with a single home base.

In our first set of experiments in Chapter 3, we studied synchronous d-regular arbitrary networks with visibility 1. For graphs of given size and degree we wanted to find a certain number of starting points that can minimize the number of agents deployed. To do this, we used a genetic algorithm to find good combinations of starting locations. As expected, we observed that the time required to decontaminate the entire network decreased when the number of home bases and/or the degree of nodes increased. Despite using the genetic algorithm in an attempt to find good solutions, we had an overuse of mobile software agents. We also made several observations on the patterns present in the results for the number of agents used and the overuse of agents.

In Chapter 4, we studied arbitrary and d-regular arbitrary synchronous networks with both visibility models. In particular, we verified that the patterns we initially observed in Chapter 3 with visibility 1, are in fact not linked to the good combinations of starting locations nor to the degree of the graph. We strongly believe that these observed patterns

occur only when the number of edges and vertices are constant. In addition to this, visibility 2 not only solves the overuse of agents but also reduces the maximum number of agents required in all cases.

In our last set of experiments in Chapter 5, we considered both arbitrary and d-regular arbitrary asynchronous networks with both visibility models. In particular, we studied the limitations of our algorithms for both visibility models. While visibility 1 could not be adapted, we proposed a straight forward modification for visibility 2. As we had previously observed, when there was an increase in the number of starting locations, the algorithm used less time. However, the overuse of agents observed in the results shed some light on the limitations of our algorithm. In practical applications, such as wireless sensor networks or distributed databases, this overuse of agents is undesirable.

In conclusion, these experiments have given us a better understanding on how to solve the decontamination problem in arbitrary networks. It seems that for our algorithms, arbitrary graphs have common behaviors that depends on how dense the network is, the network size, and starting locations. In addition to this, using multiple starting locations in arbitrary synchronous networks yields improvements both in time and the number of agents required. In particular, we observed that for each graph group a certain number of home bases always seems to require the least number of agents.

Furthermore, the visibility models when combined with cloning not only provides greater autonomy to agents but also results in more efficient algorithms. Our algorithm with visibility 1 did not perform as well as expected, mostly due to the BFS nature of our algorithm. Visibility 2, however, provided the most efficient algorithm for arbitrary networks, and could adapt to asynchronous networks.

6.2 Future Work

Since, in this thesis, we did not use topology specific algorithms unlike most of the related work in this area for mobile agents, our algorithm performed below expectations. To overcome this issue, it would be interesting to look further into other potential algorithms that might be suited for asynchronous arbitrary networks, or some form of coordination.

One of the things we did not have the opportunity to pursue is the use of the genetic algorithm to find the number of home bases which minimizes the number of agents required for specific graph families.

Furthermore, it would also be interesting to study how our visibility models will perform in some commonly used graphs such as the hypercube, the torus, and chordal rings.

REFERENCES

[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] L. Barrière and P. Fraignaud and N. Santoro and D.M. Thilikos. Searching is not jumping. 29th Workshop on Graph TheoreticConcepts in Computer Science (WG), Elspeet, the Netherlands, Springer Verlag, LNCS **2880**, 34-45, 2003.

[3] 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.

[4] H.L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and Treewidth of Graphs. *Journal of Algorithms*, **21**, 358-402, 1996.

[5] 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.

[6] H.L. Bodlaender and R.H. Moehring. The pathwidth and treewidth of cographs. *SIAM Journal of Discrete Mathematics*, **6** (2), 181-188, 1993.

[7] R. Breish. An intuitive approach to speleotopology. *Southwestern cavers*, **VI** (5), 72-28, 1967.

[8] R. Chang. Single step graph search problem. *Information Processing Letters*, 40(2)107-111, 1991.

[9] N. Dendris, L. Kirousis, and D. Thilikos. Fugitive-search games on graphs and related parameters. *Theoretical Computer Science*, 172(1-2):233-254, 1997.

[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] Paola Flocchini, Miao Jun Huang and Flaminia L. Luccio. Decontamination of Chordal Rings and Tori. APDCM 108.

[12] M. Fellows and M.langston. On search, decision and the efficiency of polynomial time algorithm. In 21st ACM *Symp. on Theory of Computing* (STOC '89), pp. 501-512, 1989.

[13] F. Fomin and P. Golovach. Graph searching and interval completion. *SIAM Journal on Discrete mathematics* 13(4), 454-464, 2000.

[14] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning. *Addison-Wesley.* , 1989.

[15] R.L. Graham and D.E. Knuth and O. Oren Patashnik. Concrete Mathematics, Second Edition, Reading, Massachusetts, Addison-Wesley, 1994.

[16] J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, **45** (3), 233-248, 1993.

[17] S. Hansen and M. Eldredge. Intruder isolation and monitoring. In 1st USENIX Security Workshop, pages 63-64, 1988.

[18] T. Kloks. Treewidth. PhD thesis, Utrecht University, Utrecht, The Netherlands, 1993.

[19] 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.

[20] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, **47**, 205-218, 1986.

[21] L. M. Kirousis and C. H. Papadimitriou. Interval graphs and searching. Discrete Math. 55 (1985), 181-184.

[22] A. Lapaugh. Recontamination does not help to search a graph. *Journal of the ACM*, **40** (2), 224-245, 1993.

[23] Fabrizio Luccio, Linda Pagli1 and Nicola Santoro. Network Decontamination with Local Immunization, APDCM, 110-118, 2006.

[24] P. Flocchini, F.L. Luccio, L. Song. Size optimal strategies for capturing an intruder in mesh networks, International Conference on Communications in Computing (CIC 2005).

[25] P. Flocchini, M. J. Huang, F.L. Luccio. Contiguous search in the hypercube for capturing an intruder, Proc. of 18th IEEE International Parallel and Distributed Processing (IPDPS 2005).

[26] 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.

[27] 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.

[28] F. Makedon and H. Sudborough. Minimizing width in linear layout. In 10th *Int. Colloquium on Automata, Languages, and Programming* (ICALP '83), LNCS 154, Springer-Verlag, 478-490, 1983.

[29] B. Monien and I.H. Sudborough. Min cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, **58**, 209-229, 1988.

[30] T. Parson. Pursuit-evasion problem on a graph. *Theory and applications in graphs*, Lecture Notes in Mathematics, Springer-Verlag, 426-441, 1976.

[31] 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.

[32] S. Peng and M. Ko and C. Ho and T. Hsu and C. Tang. Graph searching on chordal graphs. *Algorithmica*, **27**, 395-426, 2000.

[33] 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.

[34] P.D. Seymour and R. Thomas, Graph searching, and a minmax theorem for tree width. *Journal of Combinatoral Theory*, Series B 58, 22-53, 1993.

[35] J. Smith. Minimal trees of given search number. *Discrete mathematics* 66(1987)191-202.

[36] A. Steger and NC. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing (1999)* 8, 377-396.

[37] Y. Stamatiou and D. Thilikos. Monotonicity and inert fugitive search games. In 6th *Twente Workshop on Graphs and Comb. Opt.*, Elsevier, 1999.

[38] A. Takahashi, S. Ueno, and Y. Kajitani. Mixed searching and proper path width. *Theoretical Computer Science*, 137(2):253-268, 1996.