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

Ming Xie

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Locating Mobile Agents through Distributed Mechanisms

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

Sagar Naik

Nicola Santoro

Amiya Nayak

Nejib Zaguia

Gary W. Slater

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

Locating Mobile Agents through Distributed Mechanisms

by

Ming Xie

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfilment of the requirements

For the PhD degree of Computer Science

Supervisor: **Dr. Paola Flocchini**

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Your file Votre référence
ISBN: 978-0-494-48664-1
Our file Notre référence
ISBN: 978-0-494-48664-1

Abstract

Mobile agents are processes that may be dispatched from a source computer and be transported to remote servers for execution. In any mobile agent system, the ability to communicate with agents in real-time is essential for retrieving any data or information that they have collected, and for supporting coordination and cooperation among them. Communication with a mobile agent requires the ability to track it. In the thesis, we propose several distributed mechanisms for tracking (or locating) mobile agents.

We first propose a method based on distributed hashing table (DHT) to locate mobile agents efficiently in a large-scale network. The idea is to treat mobile agents and hosts similarly to the way peers are treated in a peer-to-peer system, and to implement a lookup protocol that is based on that principle, while taking care of the mobile nature of the agents. We then introduce a new locating mechanism. Our technique is based on appropriate delays that the mobile agents must perform while moving on the network so to facilitate its tracking, should it be needed. The searching agent computes a particular searching path that will guarantee the tracking within one traversal of the network. We indicate various strategies, and we perform experiments for computing the searching path and for comparing their performances in synchronous and asynchronous settings, in arbitrary and specific topologies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile agents [2, 7, 26, 85] are programs that can be dispatched from one computer and transported to a remote computer for execution. Arriving at the remote computer, they present their credentials and obtain access to local services and data. The remote computer may also serve as a broker by bringing together agents with similar interests and compatible goals, thus providing a meeting place at which agents can interact. The program chooses when and where to migrate. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine.

1.1 The Locating Problem

One of the most important problems when computing in systems of mobile agents is the problem of *locating* (or *tracking*) a mobile agent. By definition, a mobile agent moves in the network, and it might bee desirable to locate it for various reasons, for example to terminate its job, to give further instruction, or to collect information.

In any mobile agent system, the ability to communicate with agents in real-time, as agents move from one network node to another, is essential for retrieving any data or information that they have collected, and for supporting coordination and cooperation among them. Coordination (both among agents and with other entities that the agents can encounter during the execution in the hosting environments) plays a fundamental role in mobile-agent applications. Some motivates for the real-time communication with the mobile agents are the following:

- A mobile agent application often involves a collection of agents working together for a common task. For cooperation among agents to succeed, the inter-agent communication at any time is required. The mobile agents need to communicate with each other for exchanging information and achieving synchronization.

- The user needs to control the sent out mobile agent: startup, terminate, activate, deactivate etc. For example, if a user decides that an agent is no longer used, then the system should allow the user-controlled termination of the agent (i.e., if the results of a mobile agent searching a distributed data base are no longer relevant, or if an agent monitoring an information source is no longer needed, then the agent could be terminated.) A mechanism providing this functionality allows an agent to be stopped and removed from the system. One possibility to implement a mechanism for terminating agents is to use a mechanism to locate the agent and to send the termination command to it subsequently.

- The user exchanges information with the mobile agent: send instructions to and receive reports from the mobile agent. For example, if an agent has been sent out, and the user requests a status report, then the agent has to be located somehow to allow a detailed examination.

- The client may request some services from existing agents somewhere in the system. The idle mobile agents in the system can be reused by the clients and will be located firstly.

In many applications the ability to get information about the computation in progress, i.e. to get status information from the working agents, or to interact with the agents, e.g. to change their behavior, is essential. Communication with a mobile agent subsumes the ability to locate it, which means to find the node and execution environment in which it currently resides. To do this reliably and with acceptable costs, mechanisms have to be used that locate the agent in order to communicate. A mechanism for locating agents returns either the current location of the agent, or the information that the agent does not exist anymore.

1.2 Main Limitations of the Existing Solutions

As we will see in more details in Chapter 2, typical solutions to the locating problem involve: 1) the existence of a central host where all the movements of the agents are constantly updated, or 2) having each agent leave a trace of its movements on its way. Both solutions present obvious disadvantages. In the centralized solution the biggest problems are related to fault tolerance and security (problems that are present every time a centralized solution is employed). In fact, the central database could crash resulting in a complete loss of the information; moreover a third party accessing the central database could immediately determine the positions of all the agents at a given time. The second solution consists of leaving at each node the indication of the node where the agent has moved (*forwarding pointer*). This has to be done for each agent possibly incurring in a

high space complexity to store all the information.

1.3 Thesis Contributions

In the thesis we present several novel distributed mechanisms for locating mobile agents, where there no need for a central homebase nor long forwarding pointers. Two approaches to the locating problem are investigated in two parts, with more emphasis on the second: a) a peer-to-peer approach, where both agents and hosts are treated as peers and mechanisms reminiscent of those employed in peer-to-peer system are used; b) a graph-theoretical approach, where the structure of the network is exploited to construct good locating strategies.

Peer-to-peer Approach. We present for the first time a method based on distributed hashing table to locate mobile agents efficiently in a large-scale network. The idea is to treat mobile agents and hosts similarly to the way peers are treated in a peer-to-peer system, and to implement a lookup protocol that is based on that principle. The locating scheme is adapted from the DHT "Chord" resulting in a scalable, efficient and transparent solution. The adaptation is not straightforward and it is done taking care of the problem due to the mobile nature of the agents. The concept of maintaining an Agent Pool is also introduced to re-use agents instead of creating new ones.

Graph Theoretical Approach. We devise new locating mechanisms based on a completely new semi-cooperative approach: while performing its own prescribed task, a mobile agent moves keeping in mind that a searching agent might be looking for it. Our method is based on appropriate delays that the mobile agents must perform while moving on the network so to facilitate its tracking, should it be needed. The searching agent computes

a particular searching path that will guarantee the tracking within one traversal of the network. The delays to be computed depend on structural properties of the network. We perform several experiments following different strategies for computing the searching path in random and in random regular networks. We compare the various strategies and we find that our approach is particularly suitable for sparse networks.

The method is applied first to Synchronous networks and then extended to asynchronous and bounded delays networks. In the asynchronous case, a mobile agent will take a variable time to traverse an edge and that time may change depending on the network conditions. In this case our strategy does not guarantee the location of the agent within one traversal, however, simulations show that the success rate is good. The delay time for the moving agent, location success rate and location time will be studied through the simulations when the traversing time is randomly chosen for each traversal. In the bounded transmission delay model, the traversing time for an edge is bounded by a value and values are not necessarily all equal. In this setting we provide a solution that guarantees the location within one traversal.

Some special topologies are then considered: the tree network, and the mesh. These studies provide only preliminary observations and can be the basis for further research. For the case of the tree the *saturation* technique is used to collect tree information. Based on the collected information, some "good" traversal are found and the corresponding maximum chord lengths can be obtained. In the mesh network, we concentrate on traversals based on Hamiltonian paths and we derive some results. In the mesh network, for example, it is open to determine whether the best traversal indicated for the mesh is indeed optimal (in our results it is optimal only in order of magnitude). Also, our theoretical studies were concentrated on traversals based on hamiltonian paths, traversals

non necessarily based on Hamiltonian paths are still to be investigated.

1.4 Thesis Outline

The thesis is organized as follows.

Chapter 2 is devoted to mobile agents. We describe some advantages for their employ, some of their applications; we then introduce the locating problem with a survey of the existing solutions and we stress their limitations.; we finally describe related problems (rendezvous, pursuit evasion, and decontamination). In Chapter 3, we present and discuss the new method based on distributed hashing table to locate mobile agents efficiently in a large-scale network. In Chapter 4, we devise locating mechanisms for a synchronous network based on a semi-cooperative approach. We compare several proposals and we conclude that our approach is particularly suitable for sparse networks. In Chapter 5, an extended model for the semi-cooperative approach is presented to support asynchronous networks, where the transmission time depends on traffic. In Chapter 6 we devise a strategy for the bounded transmission delay model, a model which lies in between the two discussed in the previous chapters. In Chapters 7 and 8, we further study the new methodology for locating a mobile agent in Trees and Meshes. Finally, in Chapter 9 we draw some conclusions and indicate open directions.

Chapter 2

Literature Review

In this chapter, we briefly introduce the properties, applications, platforms and interaction mechanisms of mobile agents, and review the existing algorithms for locating mobile agents and other related problems.

2.1 Introduction to Mobile Agents

2.1.1 Properties of Mobile Agents

Traditional distributed applications assign a set of processes to a given execution environment that, acting as local-resource managers, cooperate in a network-unaware fashion. In contrast, the mobile-agent paradigm defines applications as consisting of network-aware entities (agents) which can exhibit mobility by actively changing their execution environment, transferring themselves during execution. Thus, mobile agents offer several advantages over traditional approaches to Internet applications. For example:

- They save significant bandwidth by moving locally to the resources they need. By transferring an agent across the network to the source of data to process it there, the communication bandwidth and communication latency can be reduced.

- They carry the code to manage remote resources and do not need the remote availability of a specific server.

- They proceed without continuous network connections, because interacting entities can be moved to the same site when connections are available, and interact without requiring further network connections.

- They contain the protocols. Protocols often become a legacy problem with the development of the Internet. Mobile agents permit new protocols to be installed automatically, and only as needed for a particular interaction.

- They support asynchronous and autonomous operation, especially in the mobile computing systems. A mobile agent does not need a permanent connection to its owner; it performs its task asynchronously. By employing the mobile agent paradigm, a task can be encapsulated in an agent created on the mobile device while disconnected. This agent is then launched into the network, where it performs its task. The agent now autonomously follows the given task. Clearly, the mobile device can be disconnected as soon as the agent is transferred. Later, the mobile device can reconnect to accept the results of the agent's task.

- They adapt to a changing environment dynamically and flexibly. Mobile agents can examine their execution environment, and adapt dynamically to changes, e.g. by using different services. If a service needed is not accessible, then the agent may choose to use a combination of other services to access the same information, or it

may move to another host in the network. This service might even be the execution environment itself.

- They support fault-tolerant computing. The potential of a mobile agent to react dynamically to unfavourable situations makes it easier to build robust and fault-tolerant distributed systems. An agent can e.g. be instructed how to solve problems such as services that are no longer provided by using alternate services, or failing communication channels e.g. by using other servers providing the needed service.

Most of these properties could be achieved by other means, but not in this combination, and not while offering the flexibility provided by the mobile agent paradigm.

2.1.2 Applications of Mobile Agents

There are many interested applications that can be accomplished by mobile agents. We will indicate here some examples:

- *Distributed information retrieval.* Instead of moving large amounts of data through the network to extract the needed information on the client side, an agent can be sent to remote information sources, where the information can be extracted locally. This is especially valuable if the information to be extracted cannot be anticipated exactly. A mobile agent can incorporate an implementation of a specific search algorithm that extracts the data, and move to the location of the data, thus allowing for semantic information compression locally. In particular, when information spread across multiple sites has to be related, the mobile agent paradigm is beneficial.

- *Information dissemination.* Mobile agents can disseminate information (e.g. news) to a number of customers. They provide access policies, which e.g. ensure that an article can only be read after the customer has paid for it. Additionally they implement the functionality needed to show the article in the way best suited to article and output medium, e.g. computer or PDA screen, or printer.

- *Monitoring.* An agent can be sent out to monitor a given information source or wait for a specific event. As soon as a change occurs, the agent reacts according to its programming; for example, it can send a message or buy shares on a stock market. Another example is the monitoring and management of network devices. The advantages of mobile agents in this application area are the flexibility of the agent's programming and the asynchronous and autonomous execution.

- *Electronic commerce.* Electronic commerce is certainly one of the most attractive application areas: a commercial transaction may require real-time access to remote resources, such as stock quotes. A mobile agent can act and negotiate on behalf of its user, buying, selling, or trading goods, services or information.

- *Parallel processing.* In principle, parallel processes can be distributed by mobile agents very easily. The mobile agents distributing the computation might even redistribute themselves to adapt to changes in the environment. In a framework that allows transparent distribution of computations over a network of mobile agent systems, all computations that can be split into smaller, autonomously computable parts can be distributed automatically, thus employing the full computational capacity of a network of computers.

- *Workflow management systems.* In workflow applications the flow of data between coworkers characterizes the processing of information. Mobile agents are especially

useful here because they provide mobility, behavior, information about the workflow and autonomy to every workflow item. Thus the workflow becomes independent of a particular workflow application. Moreover, the agent can provide all code necessary to access the items of information in a semantically correct manner.

- *Software deployment.* Mobile agents can be used to automate the process of software installation and software updates. Before transporting the software package itself to the target computer, an agent can gather information about the environment in which the software will be installed, e.g. its version and additional packages. It can query the user for installation preferences, uncompress and compile the software, and watch for future software updates.

2.1.3 Interactions among Mobile Agents

In between migrations, a mobile agent may need to interact with other agents to complete its jobs. Cabri et.al. [23] identify four kinds of coordination models for agents to employ in interacting with one another.

- *Direct Communication*: The first model is based on direct coordination and involves the agents sending messages directly to other agents. Agents start a communication by explicitly naming the partners involved. In the case of inter-agent communication, two agents must agree on a communication protocol, typically peer-to-peer. Direct communication usually implies temporal coupling-synchronizing the entities involved.

- *Meeting-oriented Communication*: The second coordination model is referred to as meeting-oriented that does not require agents know the names of other agents, in-

stead agents join the meeting points that they either explicitly or implicitly known; afterward, they can communicate and synchronize with the other agents that participate in such meetings.

- *Blackboard-based Communication*: The third model is the blackboard based model, where agents post and retrieve messages from a shared data spaces. One advantage of the blackboard-based model is that the agents do not need to exist at the same time. A sender can post and terminate before the receiver ever reads the message.

- *Linda-like Communication*: The final model is the Linda-like model, which is based on the associative mechanisms of the Linda model of process creation and coordination. As in the blackboard-based model, a space is used to store and retrieve messages, called tuples; however, unlike the blackboard-based model, the agents retrieve the messages in an associative way.

The mobile agents being considered in this thesis communicate directly with one another and consequently direct coordination mechanisms are required. Since communication is achieved only directly, to transmit information to an agent it is first necessary to locate it.

2.2 Locating Algorithms

Different mechanisms for locating agents can be identified, depending on the assumptions about migrational behavior, on assumptions about the size of the agent system, and on assumptions about the communication costs.

2.2.1 Typical Locating Algorithms

The classical location techniques are based on centralized solutions and on forwarding pointers (e.g., see [4, 11, 40, 64]). Both approaches require a certain host or a registration server to initiate a lookup.

Home-based approach is a basic and widely used mobile agent locating approach. In the approach, each mobile agent reports its current location to its home host (or a Location Database Server) whenever it moves to a new location. The home host always keeps tracks of the agent and records the latest location of the mobile agent, and acts as a permanent forwarding point. A request for the agent will contact the home host to obtain the address firstly, then contact the mobile agent directly. Then, it bypasses the home host for further communication. There are obvious disadvantages for this approach, such as single point of failure and communication cost. Since a mobile agent will communicate with its home host for refreshing its location whenever it arrives a new location, especially when the agent is far away from its home host, the communication cost is significant. The approach is not scalable, not fault tolerant, and not suitable for large scale network.

In home-based approach, the updating cost is significant since the mobile agent will update its location whenever it arrives a new location. One reasonable improvement is that, a mobile agent updates its location once every d movements instead of every one movement. T. Li et al [58, 59] proposed such a basic tracking scheme to accomplish it. A mobile agent updates it location once every d movements and is then searched by Sequential Searching Method. The authors try to set up some optimal algorithms to find suitable d. It is obvious that, although the updating cost is decreased by lowing the updating frequency, the sequential search increases the searching cost. This tracking scheme inherits most disadvantages of home-based approach and is also not suitable for

large-scale network.

The *forwarding pointer* [40, 9, 64] approach provides functionality for locating mobile agents or any other distributed objects, such that tracking is performed with the use of forwarding pointers that are distributed along the path of the traversing mobile agent. Each time the agent changes its location a pointer to the new location is deposited at the old location. In the *forwarding pointer* approach, when an agent moves through the network, the agent leaves its new location on the old place when it migrates, then a path of forwarding pointers is created. A lookup will initiate from the original static host, then follow the forwarding pointers ultimately leading to the place where the agent currently resides. The *forwarding pointer* approach allows the agents to roam on the network without having to contact their home host, which means there is no additional communication cost for refresh its latest location information compared with home-based approach.

In the approach, the forwarding pointer path can be refreshed to shorten the path. Whenever a search for an agent is successful, the information about the location of the agent is sent to the original host, and set the target of the pointer at the original host to the new place. The intermediate pointers are now useless and can be removed. The remove action can be done by send a *remove* request along the old pointer path to clean the expired pointers. Moveover, if there is a circular path, the circular part is removed.

The *forwarding pointer* approach has several disadvantages. Firstly, the initial location of the path to follow needs to be known before lookup. The communication cost to find an agent can be arbitrarily high, depending on the length of the path. This limits the scalability of the algorithm, especially for large scale network. There is no fault tolerance mechanism in place: if any one of the intermediate forwarding pointer is unreachable,

the agent can not be found. Thus, the longer the path, the worse the availability of the system. Another problem of this approach is how to remove the forwarding pointers when the path is no longer valid.

2.2.2 Other Locating Algorithms

Hashing approach. G. Kastidou et al [49, 50] proposed a locating solution based on hashing, which scales well with both the number of agents and their mobility rates, such as the number of moving and querying operations. The mobile system constitutes two tiers. There are some special agents, called IAgents (Information Agents), each of them maintains the current location of a set of mobile agents. The normal mobile agents are organized in sets, and each agent is associated with a certain set which is maintained by a certain IAgent. This can be determined through a system-wide hash function. The IAgents can be mobile or static. The location of a mobile IAgent depends on the distribution of the mobile agents that they serve. To locate a mobile agent A, the system will find the ID of the certain IAgent that maintains the precise current location of A. This is done by applying the hash function on A's id. In the system, there is a central static agent - HAgent (Hash agent) that keeps the current hash function. To allow the system to adapt to the changing number of mobile agents and the varying system workload, such as the mobility rate of the agents and the rate of requests for communication, the number of IAgents changes over time. When an IAgent is over-loaded, it splits its load by creating a new IAgent. On the other hand, under-loaded IAgents are merged by assigning their load to other existing IAgents. Every time that the hash function changes, the copy of the HAgent is immediately updated.

Secure hashing approach. V. Roth et al [76] propose a secure and scalable global

tracking service for mobile agents. It is not only scalable to the Internet, but also suitable to a high rate of location update requests. Registering is used as the basic tracking mechanism here, and a global hash table is proposed. In order to make this tracking service scalable globally, the range of hash function is divided to subranges and a tracking server is set up for each subrange. Each of the slots of the hash table is managed by a tracking server. Agents are assigned to the slots by using a cryptographic hash function to fulfill the security requirements.

Guess approach. When using the traditional tracking algorithms, some exceptions may happen. For example, the location database server is not available or loses contact with the agent, or the forward pointer path is broken. Y. Lien et al [60, 25] proposed some binary search algorithms and an algorithm called "highest probability first search" to handle these cases. These algorithms try to guess the location of an agent by using the knowledge of an agent's movements. They assume a predefined path from which the agent can not deviate, and try to compute the current place of the agent using the time interval since the agent left, assuming a binomial distribution of the execution time on each place. This assumption makes the approach unusable for all those applications in which the migration path of the agent is not known in advance. For instance, in e-commerce or in information retrieval scenarios the migration path of the agent depends on information they obtain while en route. This approach does not guarantee that the agent is found before arriving at its end point, i.e. if the agent leaves every place jut before the mechanism guesses this place. Some of these algorithms try to compute the current place of the agent using the time interval since the agent left with the help of a probability function. If the target agent or the service network can't work normally, all estimation about the service time will be wrong. Moveover, a slip-through problem exists in some of these algorithms because the agent may not be found before arriving at its end

point, e.g. if the agent leaves every place just before the mechanism guesses this place.

Holding list approach. In a large scale agent system, locating an agent by referring to a middle agent, such as the registration server, usually adds a large communication complexity. O. Shehory suggested a scalable agent location approach in [80] with no need of middle agents. This approach requires that each agent i holds a list L_i of other agents it knows. This list includes the information of the agents' names, addresses, functions and other relevant information. When an agent needs to locate another agent which does not exist in its list, it will consult some or all the agents on its list for such information. These agents will in turn perform the same procedure recursively. A unique request id will be used to prevent an agent from handling a request originated from itself to avoid the request cycles. Since in a large scale agent system it is too costly for an agent to keep a large list L_i, an appropriate selection of the size of L_i will allow agent location via a very low exploration depth and few communication operations.

Hierarchical approach. L. Bernardo et al [19] proposed a multi-tier hierarchical structure for locating services in the large scale network. In the network, each agent system is tied to a location server, running locally or on a nearby system. These location servers are connected to others to offer a global locating service, and are structured as a mixture of a meshed and a hierarchical structure. At each hierarchical level, the location servers interact with some of the others at that level and, possibly, with one above. Higher hierarchical levels always have incomplete information about the location of the server that stores the agents' location. The client performs searches on a step by step basis, through a sequence of location servers. The routing information for the path is based on service hints. A location server will forward messages to its location server on the higher layer by using an identical simplification process. The information in servers at higher hierarchical

levels will be composed of incomplete service hints (identified by hash values). They offer a broader but less detailed vision of the locating services and act as a distributed index service.

There are many mobile agent tracking algorithms applied in certain areas, such as tracking mobile agents in Internet network [82], tracking mobile agents in a wide distributed environment which proposed a location finding protocol called the *Search-By-Path-Chase* [81], etc. This *Search-By-Path-Chase* protocol is based on an efficient algorithm which follows a part of the links the agent has left on the registers of the visited sites or regions. In this protocol, the agent is locked from migrating after it is found until the interaction has taken place.

Ooi et al [70] integrate agent and peer to peer system to collect information through the network, but do not discuss the locating problem. In the same vain, we propose a fully distributed strategy by integrating agent and peer to peer system in Chapter 3.

2.2.3 Locating Schemes in Mobile Agent Platforms

There are several mechanisms for locating agents when an agent migrates only along a preordained migration path: searching the migration path sequentially or in parallel or by guessing. With dynamic paths, three different groups of mechanism exist: mechanisms using no logging (brute force methods), mechanisms employing logging (either database or path proxies), and mechanisms applying non-deterministic methods.

- *Brute Force Search*: This technique first identifies every agent system in a region, a set of agent systems owned by the same person or organization. Then each system is checked to find the agent. Searching can be done in parallel or sequentially.

- *Logging*: When an agent leaves an agent system, it leaves on this system a log entry (called mark) that says where it is going. By following the log entries the agent can be located. The log entries are garbage-collected after the agent dies. Logging is the same as using forwarding pointers. The main disadvantage is its low reliability.

- *Agent Registration*: An agent updates its location in a predefined directory server that allows agents to be registered, unregistered or located. Other agents uses the directory to locate the agent. The location information may be stored in one centralized database, such as MASIF [64] or distributed database, such as Aglets [2].

- *Agent Advertisement*: The agent advertises its location whenever it deems it is necessary. To find an agent for which the advertised information is out-of-date, brute force search has to be used.

Combinations of various schemes may be used in a mobile agent platform. Depending on the scenarios a method could be better than another. Brute force search is not feasible for large agent system: either the agent can be found in the own area of authority or the whole distributed system has to be searched. Multiple agents workin parallel on a specific task at different hosts, which try to locate and establish communication with each other. In this case, the trail information necessary to them to locate each other is not available, while registration scheme is a better choice. If requests for naming servers are overloaded or tail information is temporarily unavailable, brute force scheme is used.

Few other mobile agent systems support naming services for locating agents based on limited schemes: Voyager [85] supports only forwarding of messages based on the Logging scheme while Odyssey [69] supports Registration via an interface to a naming server, which is configured with an agent system. In contrast, MOA [66] supports multiple

schemes. In comparison with Aglet, MOA uses special location objects to encapsulate the current location of an agent, as well as the specific locating scheme to locate it. Before an agent is located, its location object needs to be located in naming servers. Finally, as mentioned earlier, MASIF [64, 65] standardizes the locating of agents via Registration for heterogenous type of agents.

2.3 Related Problems

In this section, we introduce several related problems for mobile agents.

- *Rendezvous.* Two or more mobile agents *cooperate* to find each other; in fact, their goal is to meet somewhere in the network and their actions go towards this common goal.
- *Pursuit-evasion.* Two agents are present in the systes: one agent tries to escape, the other agents tries to capture it.
- *Decontamination.* A team of agents cleans a contaminated network. The problem can be seen also as an intruder capture problem, where however, the agents do not want the intruder to move on nodes already traversed (recontamination). It is a non-cooperative problem; the intruder knows everything, which implies that it will be located only at the very end of the cleaning.

2.3.1 Rendezvous

The *rendezvous* problem [6, 36, 55] is a natural and fundamental problem for mobile software agents traveling through a distributed network. It was first proposed by Steven

Alpern [5] and since then has been widely investigated, under different scenarios and different assumption: in a distributed environment, a set of agents are placed in a domain and are required to all meet or rendezvous at the same place and time within the domain. The settings differ mainly in the types and properties of the agents and the types and properties of domains. More precisely, given a particular agent model (e.g., deterministic, anonymous agents with knowledge they are on a ring of size n) and network model (e.g., anonymous, synchronous with tokens) a set of k agents distributed arbitrarily over the nodes of the network are said to rendezvous if after running their programs after some finite time they all occupy the same node of the network at the same time. It is generally assumed that two agents occupying the same node can recognize this fact (though in many instances this fact is not required for rendezvous to occur). As stated, rendezvous is assumed to occur at nodes. In some instances one considers the possibility of rendezvous on an edge, i.e., if both agents use the same edge (in opposite directions) at the same time.

The definition of rendezvous must begin with establishing the properties of the agents that will rendezvous and the domain in which rendezvous will occur. An important property to consider is whether or not the agents are distinguishable, i.e., if they have distinct labels or identities. Agents without identities are referred to as *anonymous* agents. When the mobile agents or the network nodes are uniquely numbered, solving the rendezvous search problem is trivial. When the mobile agents are identical and the network nodes are anonymous, the symmetry can make the problem difficult to solve.

Among deterministic solutions to rendezvous, Yu and Yung [86] proposed an algorithm for gathering multiple agents in synchronous graphs assuming that the topology is known to the agents. Dessmark et al. [31] showed how two agents can meet in synchronous rings

or arbitrary graphs, when they have distinct identities. This result was later improved in [52] to obtain a polynomial solution. A solution to rendezvous in the asynchronous graphs was provided by De Marco et al. [61]. All the above results are for agents having distinct labels and but having no ability to mark the nodes of the graph.

The idea of performing rendezvous search using marks on the starting places was explored first by Baston and Gal [16]. Notice that if marking of nodes is permitted then it could be possible to achieve rendezvous of anonymous agents in unlabelled graphs. Kranakis et al. [56] and Flocchini et al. [36] gave solutions for the rendezvous of two (resp. multiple) agents, on a ring networks using unmovable pebbles (called tokens) to mark the starting nodes. Gasieniec et al. [45] gave an optimal memory solution for the same setting. Kranakis et al. [54] studied the problem for a synchronous torus using both fixed and movable tokens.

Using whiteboards for marking the nodes, Barrière et al. achieved rendezvous of multiple agents on arbitrary unlabelled graphs with sense of direction [13]. In presence of whiteboards, the rendezvous problem is equivalent to the agent election problem. In [14], Barrière et al. consider solutions to the agent election problem in presence of distinct but incomparable agent labels. This result was improved by Chalopin [24] who gave an effective solution for this model. Among fault-tolerant solutions, Flocchini et al. [38] considered the case when tokens fail, i.e. when the tokens suddenly disappear. Dobrev et al. [34] solved the rendezvous and near-gathering problems in the presence of a dangerous node (black hole). Both these results are for the ring networks only.

Recently the theoretical computer science community has taken up the challenge of the problem of rendezvous for autonomous software agents moving through a distributed network. Requiring such agents to meet in order to synchronize, share information, divide

up duties, etc. would seem to be a natural fundamental operation useful as a subroutine in more complicated applications such as web-crawling, peer-to-peer lookup, meeting scheduling, etc.

2.3.2 Pursuit-evasion

Pursuit-evasion [72, 28, 1] is a very problem that has been extensively studied in deterministic and especially in randomized environments. In pursuit evasion there is a *competitive* setting, where one agent tries to escape, while the others are chasing it. [28] presents a pursuer-evader problem where a single faster agent tries to track a single slower adversarial intruder.

Pursuit-evasion games are problems of fundamental interest in many diverse fields such as computer-science, operations-research, game theory and control theory. The goal of a pursuit-evasion game is to find a strategy for a pursuer trying to catch an evader who, in turn, tries to avoid capture indefinitely.

- *Environment where the game is played*: Examples include plane, grid, graph, etc.
- *Information available to the the players*: Do they know each others' positions all the time? Does the pursuer know evader's strategy?
- *Controllability of the players' motion*: Is there a bound on their speed? Can they turn with arbitrary angles?
- *Meaning of capture*: In some games, the pursuer captures the evader if the distance between them is less than a threshold. In other games, the pursuers must see or surround the evader in order to capture it.

- *Graph Traversal*

Earlier studies of *pursuit-evasion* were motivated by control tasks such as intercepting missiles. The problem is addressed in the robotics community for its applications in collision avoidance, search and rescue, and air-traffic control. In these models typically the motion of the evader is modeled by a stochastic process. However, recently there has been increasing interest in modeling games where the evader is more "intelligent" and has certain sensing capabilities. *Pursuit-evasion* games on graphs have been studied not only for their applications in network security and protocol design but also for their relations to fundamental properties of graphs such as vertex separation. A remark about the terminology: In the literature, the names *pursuer-evader*, *cop-robber*, *monster-princess*, *hunter-rabbit*, *sheriff-thief* have been used somewhat synonymously.

2.3.3 Decontamination

In networked environments supporting mobile agents, a particularly important security concern is to protect a network from unwanted, and possibly dangerous intrusions. At an abstract level, an intruder is an alien process that moves on the network to sites unoccupied by the system's agents "contaminating" the nodes it passes by. A team of agent is initially located at a node (the *homebase*) and agents can move from node to neighbouring node. At any point in time each node of the network can be in one of three possible states: *clean*, *contaminated*, *guarded*. Initially all nodes are contaminated except for the homebase (which is guarded). A node is guarded when it contains at least one agent. We say that a node is clean when an agent passes by it and all its neighbouring nodes are clean or guarded, contaminated otherwise. The solution of the problem is given by devising a strategy for the agents to move in the network in such a way that at the

end all the nodes are clean.

It is easy to see that the *decontamination* problem can be equivalently formulated in terms of an *intruder capture* problem [12], where an intruder moves arbitrarily fast in a network and a team of searching agents is deployed to capture it. The intruder capturing problem has been extensively studied in the literature under the name of *graph search* in a model where the searchers may be placed and removed from any node of the graph, i.e., they are allowed to *"jump"* while they perform the searching task. This problem was first introduced by Breish [22] and Parson [72], and after that several variations [8] of the problem have been studied: among them, node search and edge search, where the aim is to find a strategy that minimizes the number of searchers and leads the graph to a state in which all nodes (or nodes and edges) are simultaneously decontaminated.

Megiddo et al. [62] showed that finding the searcher number of a graph is NP-hard. Bienstock and Seymour [20] studied the monotone version of the problem (when recontamination is not allowed), while LaPaugh [57] was the first to show that recontamination does not help in decreasing the number of searchers required for searching.

Graph search, *intruder detection*, and *decontamination* are equivalent problems. Some papers [15, 12] studied the setting that the agents *cannot be removed from the network*: they can only move from a node to a *neighbouring node*; this assumption is obviously motivated by the fact that the software agents that are able to move only on the edges of the network. They consider the *contiguous, monotone, decontamination*: 1) the removal of agents is not allowed, 2) at any time of the search strategy, the set of clean nodes forms a connected subnetwork, and 3) a clean node cannot be recontaminated. The contiguous assumption considerably changes the nature of the problem and the classical results on node and edge search do not generally apply.

2.3.4 Graph Traversal and Exploration

When the nodes of the graph have distinct identifiers, then the problems of traversal and exploration are equivalent, (i.e. if the agent succeeds in traversing every edge of the graph, it can obtain a map of the graph). Previous studies on the exploration of such graphs (or digraphs), have emphasized on minimizing the cost of exploration in terms of the total number of edge traversals (See for example [3, 29, 30, 71]. In this case, a simple depth-first traversal is optimal in terms of number of moves (i.e. making $\Theta(m)$ edge traversals) while a modified version given by Panaite and Pelc [71] takes $m + O(n)$ moves. Awerbuch et al. [10] have studied the problem of piece-meal exploration where the robot has to periodically return to its homebase during the exploration.

A more challenging problem is to traverse an anonymous graph (i.e. where the nodes are unlabelled). In this case, traversing the graph is not equivalent to mapping. The knowledge of the size n (or the diameter D) of the graph is a necessary and sufficient condition for achieving traversal. But solving the mapping problem requires that the robots have some means of marking the nodes (except when the graph is a acyclic, i.e. it is a tree). Different models for marking the nodes have been used by different authors. Bender et al. [17] used the method of dropping a pebble on a node to mark it and showed that any strongly connected directed graph can be explored using just one pebble, if the size of the graph is known and using $O(\log \log n)$ pebbles, otherwise. Dudek et al. [35] used a set of distinct markers to explore unlabelled undirected graphs. Yet another approach, used by Bender and Slonim [18] was to employ two cooperating agents, one of which would stand on a node, thus marking it, while the other explores new edges. Another much stronger model is the whiteboard model (used in [41, 42] where the robot can write on public whiteboards available at each node. With unbounded whiteboard memory,

single agent exploration(and mapping) is again reduced to performing a traversal.

Most of the known results on exploration are for single agent exploration with the exception of [18] (which uses two agents as mentioned above) and [41] where multiple agents are used to explore a tree. Notice that when the nodes have to be marked in order to map the graph, the distributed exploration by multiple agents is more difficult than single agent exploration ([27]). However the distributed exploration can be reduced to single agent exploration if we can gather the robots together in one node (Rendezvous) or elect a leader among the agents (Leader Election).

There has been many studies on graph traversals when the memory available to an agent(i.e. its ability to remember) is limited. The agent is often modelled as a finite automaton and the emphasis is on minimizing the number of states of the automaton. Fraigniaud et al. [43] show that $\Theta(D \log \Delta)$ memory is sufficient for traversing graphs of diameter D and maximum degree Δ. They also show that at least $\log(n)$ bits of memory is necessary for traversing a graph of size n. In [44], the authors show that k non-cooperating agents each having $\log(Q)$ memory cannot traverse all graphs of size kQ. There are many previous results on exploration of mazes by one or more finite automata (see [53, 67, 74, 79]). However, exploring mazes is much more easier in general, than exploring graphs, due to the presence of sense of direction, as was shown in [21].

Another related problem in that of perpetual traversal, where the agent is not required to halt after it has visited all the nodes. A perpetual traversal algorithm must ensure that every node is visited within a finite amount of time. It was shown in [33] that $\log(\triangle)$ bits of memory are sufficient for perpetual traversal of trees of maximum degree $\triangle$.

Chapter 3

Locating Mobile Agents through DHT

In this chapter, we propose a method based on distributed hashing [83, 75] to locate mobile agents in a system. The strategy is fully distributed, and transparent to the client that is performing the locating request; moreover, it is scalable and efficient. The idea is to treat mobile agents and hosts similarly to the way peers are treated in a peer-to-peer system, and to implement a lookup protocol that is based on that principle, while taking care of the mobile nature of the agents. We also introduce the notion of Agent Pool, which allows us to reuse mobile agents that might be idle in the system instead of creating new ones at each request. Some of the results of this Chapter appeared in [37].

3.1 Peer-to-peer systems: Chord

Before describing our locating technique we briefly introduce Peer-to-peer (P2P) systems and, in particular the one based on *Chord* [83].

Peer-to-peer systems are fully distributed dynamic systems of hosts (peers) that contain information to be shared. Chord is one of the most common P2P systems and it provides a distributed lookup protocol designed to support fast data locating and node joining/leaving.

Chord uses a m-bit identifier ring, $[0, 2^m - 1]$, for routing and locating objects. Both the objects and the peers in the system are assigned m-bit keys through a consistent hashing function and mapped to the identifier ring. An object is stored at the peer following it on the ring, i.e., its successor.

Each peer maintains a finger table for efficient routing. The finger table of a peer contains the IP address and identifiers of $O(\log N)$ other peers that are at exponentially increasing distances from the peer on the ring, where N is the number of peers in the system. At each hop during a lookup, the request is forwarded to a peer from the finger table whose identifier immediately precedes the destination point, such that the distance between the destination and the current peer will decrease by half after each hop. Thus, the routing performance is $O(\log N)$ hops.

A peer can join or leave the system at any time. When a new peer joins the system, it is assigned an identifier and sends a join request toward this identifier through an existing Chord peer. The affected finger tables are updated accordingly and the keys are assigned to the new peers transferred from its successor. Similarly, upon departure of a peer, its keys are assigned to its successor and the affected finger tables are updated.

Some systems integrate Chord to lookup distributed data, such as [77], where a content-based similarity search strategy is combined with Chord. It employs Information Retrieval techniques to extract feature vectors from documents and use them to determine the locations of the document indices in a DHT (Chord in the case). Through the technique, it offers a strategy for distributing and retrieving the index information for documents over a Chord ring using the feature vectors. In this thesis, we utilize Chord to accomplish our locating scheme too.

3.2 Overcoming some Limitations of Existing Thecniques

Our method avoids the two main problems of the existing techniques: need of a central authority, or use of long traces. In fact, it is totally decentralized while fully transparent to the client; moreover, it does not make use of forwarding pointers. We propose to associate appropriate keys to the mobile agents and to the hosts, and to employ a distributed hashing mechanism to locate them, similarly to what is commonly done for Peer-to-peer systems. Each agent is associated to a coordinator to whom it periodically reports its location. When a client needs to locate an agent, it contacts an arbitrary host with the key corresponding to the agent. The host will initiate the search and will find the coordinator of the agents, which can then contact the agent directly. The Distributed Hash mechanism is designed in such a way that the retrieval of the agent is quite efficient.

In this chapter we also introduce the concept of *Agent Pool*. We assume there are mainly two reasons for locating an agent: one is when the agent must be terminated, the other is when there is need for the agent to perform another task. In the first case it

is necessary to locate precisely a specific agent and to stop its work; in the second case, there is the need to retrieve any mobile agent that is capable to perform a certain task. It is for this second purpose that we introduce the notion of Agent Pool. The agents are created with different characteristics and different capabilities. They are then sent on the network to perform some tasks. Some agents might successfully terminate the task and they will still be present in the network in an *idle* state. If a client requests an agent with certain characteristics, we would like the system to retrieve any idle agent in the system with those characteristics. If none is present, it will have to be created. In other words, the set of agents present in the system at any given time (some busy, some idle) represents an Agent Pool, from where clients can retrieve the agent they need.

3.3 The Location Scheme

3.3.1 General Overview

The agents present in the system are considered to belong to an "*Agent Pool*" distributively managed by the hosts of the system. Each agent has its characteristics and several agents might share the same characteristics.

When a client needs an agent with certain characteristics, we would like the client to be able to request it from any host, and the host to be able to retrieve any one with those characteristics efficiently.

Similarly to a Peer-to-peer system (e.g., see [75, 83]), we propose to see the hosts in the mobile agent system as peers the and mobile agents as objects with associated keys. The two systems (P2P and mobile agents) obviously present different challenges because P2P

objects are static, while agents do move in the network; the general concept is however similar.

We organize the hosts (coordinators) like peers in a distributed hash table (DHT) structured as Chord [83]. The keys associated to coordinators and agents are drawn from the same m-bit name space. The space can be viewed as a circle, in which the highest identifier is followed by zero. Each coordinator is assigned an m-bit node identifier, which is obtained by hashing its IP address. All possible $N = 2^m$ nodes are ordered in the one-dimensional circle; the coordinators are mapped to this virtual circle according to their identifiers. The system applies a consistent hashing function to map each key onto the coordinator whose identifier most closely follows the key on the identifier circle. Therefore, each node is responsible for the key range of (predecessor, its identifier], which means each coordinator in the system maintains the information of agents whose keys belong to the space range between the preceding existed coordinator's identifier and its identifier.

When a client needs a mobile agent to fulfill a particular task, it contacts any host existing in the DHT, and sends the request with required type of service. The lookup is done in the DHT and the host will return the result, the location of a matching agent, to the client. Then the client contacts that agent directly and asks it to perform the task. Figure 3.1 shows these operation stages which can be divided into two phases, one for lookup through DHT to gain the location of the agent, another for communicating with the agent to obtain the service.

In the system, when a mobile agent moves, it reports its new location to the coordinator that is in charge of its key depending on the distributed hash table. Note that the coordinator in the DHT may not be the creator of the agent. When a host generates a new agent, it transfers the new agent's information to the host that is in charge of its

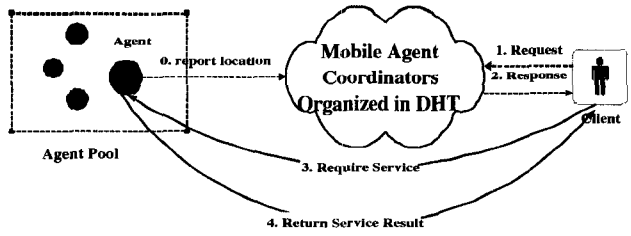

Figure 3.1: Locating Scheme

key. Subsequently, the creator is irrelevant to the location management of that new agent. Nodes that join or leave the system are treated in a similar way as in peer-to-peer systems, transferring or receiving part of the hashing table. An extra operation (compared to peer-to-peer) is that a leaving coordinator sends a change notification to the controlled agents, in such a way that those agents can communicate with the new coordinator directly.

The locating performance is $O(\log N)$ on the overlay coordinator network, where N is the number of the hosts. The experiments of Chord [83] show that the number of lookup hops is actually close to $\frac{logN}{2}$.

3.3.2 Key Management

Since our purpose is to find any agent that can offer a required service, we do not apply hashing only to the agent's ID to obtain its key but also to its characteristic information.

If the client knows the key of its expected mobile agent, it can look for that particular agent. However, if a client wishes to obtain a service, it will not care which agent offers it. So, our objective is that the client does not need to know the exact key of the agent, but, of course, it has to know what kind of services (i.e., looking for a movie with a name) it needs, and includes this information in the request.

Each coordinator's identifier is directly obtained by hashing its IP or name. Each

mobile agent owns an identifier (ID) and a characteristic that shows what kind of service it can offer (more than one agent could be performing the same service). The key of an agent is obtained by hashing the pair $(characteristic, ID)$, which concatenates the hashing value of both *characteristic* and *ID*, $key = f(hash(characteristic), hash(ID))$. For example, suppose the size of name space of DHT is 2^8, which means the length of the full key is 8 bits. We consider the first 3 bits are obtained by hashing the characteristic, and the last 5 bits are obtained by hashing its ID. Consider, for example, $agent1$ with $(characteristic1, ID1)$: let $hashing(characteristic1) = 100$ (binary), and $hashing(ID1) = 10000$ (binary), the full key of $agent1$ is then $10010000(binary) = 144(decimal)$. When a client wishes to find a agent that can offer the service of $characteristic1$. Any agent with full key between $10000000(binary) \sim 10011111 = 128 \sim 159(decimal)$ can offer the service to match the client's request. Obviously, more than one host could be in charge of those agents. If this is the case, those hosts must be consecutive neighbours in the DHT and each of them knows if its immediate predecessor or successor, which is also in charge of the matching agents. In this way, agents with the same characteristics are distributed in consecutive peers. This type of distribution helps achieving not only efficient lookup, but also load balance and fault-tolerance.

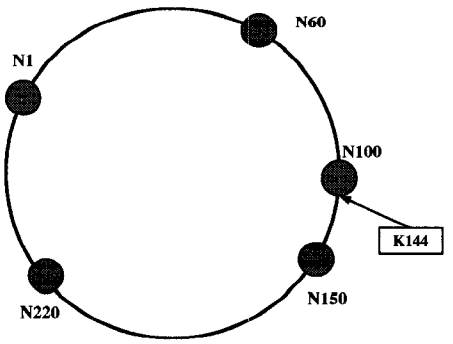

Figure 3.2: Distribution of Agents' Keys in DHT

In the example of Figure 3.2 the hosts with identifiers 100 and 150 are shown, which

may store the information of agents who can offer the service of *characteristic*1. Host 100 is in charge of *agent*1. The location of *agent*1 will be returned to the client who requested the service of *characteristic*1.

3.3.3 Lookup

There are two kinds of lookup requests. One occurs when the client wants to locate a particular mobile agent, and it knows the full key of that agent. The lookup may start from any host in the system, and follows the same lookup strategy as in Chord. A host that is in charge of the key of the required mobile agent is found. Once the location information is obtained and a reply sent to the client, the lookup is done. The other type of request occurs when the client requests a type of service to a host in the system and wants any agent able to satisfy its request to be returned. In both cases, we want the mobile agent system to be totally transparent to the client.

Let us consider this second type of request. In the beginning, a host receives the service request from the client. It first hashes the service name to obtain the first part of the full key. Suppose the length of the full key is 8, and the first 3 bits represents the service type, $Hash(characteristic) = b_1b_2b_3$. Thus, any mobile agent with the key in the range $[b_1b_2b_300000, b_1b_2b_311111]$ can offer the required service, and is the qualified candidate for the client. The system looks for the host or hosts that are in charge of those keys. Then, the lookup will be forwarded firstly to the host that manages key $b_1b_2b_300000$.

Based on the key management strategy, one or more consecutive hosts will manage those keys. Similarly to Chord, the system applies a logarithmic search strategy to lookup the host in charge of the key $b_1b_2b_300000$. If there is no available matching agent in that host, a linear search begins. It will search the key in the successors of the first found host

to lookup for an agent that qualifies. The linear search begins from the host node that theoretically manages the key $b_1b_2b_300000$ and proceeds with its successors, until the host in charge of $b_1b_2b_311111$ is reached. During the search, if there exists a qualified mobile agent, its location information is returned to the client; if not, the lookup fails or another strategy is adopted. (See the next two subsections concerning this case.)

3.3.4 Agent Status Transition

In this section we describe how to deal with different states of the agents (some are idle, some are busy).

In the following we consider clients requesting a service from a mobile agents, rather than requests for the termination of a specific agent, since this second case corresponds exactly to retrieving the agent with a logarithmic search as in Chord, while the first case requires more care.

If the searched agent is found in the idle state, it is always assigned to the client. If it is found in an active state, the service will be delayed. The status strategy is used to avoid this delay and also to face the two points raised below. First observe that there may be several agents that can offer a particular service; in this case, we want to avoid that the system always return the same first found agent based on our lookup algorithm (this would lead to an unbalanced load for the agents). The second point is about the location refreshing frequency. Some agents may be required to move only a little and perform static tasks (e.g., monitoring stock exchange). On the other hand, some agents may frequently change their location (e.g., when performing searches of information). If these fast moving agents report their latest location whenever they move, the communication overhead would be significant.

We propose to associate a *busy* state to an agent while it is performing a task and an *idle* state when being inactive. Moreover, we propose to update the agents' location only when the task is performed and the agent becomes idle again.

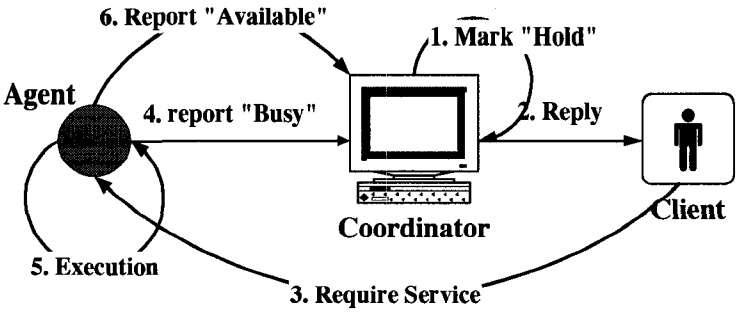

Figure 3.3: Status Transition in the System

If there is an idle agent which can offer the service, then the peer will return its location to the client. If there is no idle agent that can offer the requested service, the host returns a waiting response and put the request in a waiting list. When a matching agent is available, the coordinator will notify the client about the need to wait. If the delay becomes too high a new agent could be generated.

More precisely, an agent can be in one of the following states:

- *Available*: the agent is ready to accept a new task
- *Busy*: the agent is busy and cannot accept a new task
- *Hold*: the agent has been requested and its location has been returned to the client.

When the system finds an idle matching agent (with status "available" in the information table on the peer that controls it), it changes the status as "hold", and returns the location information to the client. The client contacts that agent for services, the agent returns "busy" to its coordinator and the coordinator marks the agent's status as "busy"

in its information table. When the agent finishes the task, it reports "available" to its coordinator and the coordinator marks that agent's status as "available" in its information table. If a "hold" status exceeds a predefined "timeout", the system considers that the client has already given up the request for service, and mark the status as "available".

There are three kinds of responses:

- Found an idle matching agent, and return the location of that agent to the client. The client can now contact that agent right.
- Found no idle matching agent, and return a waiting message to the client. The client may wait for a idle one or give up this service request or a new agent is created by the host.
- Found no matching agent. The client has to give up its request or a new agent is created by the host.

A waiting list is set up for each kind of agent to queue the requests that find only busy agents. The waiting list for a particular kind of agent can be maintained in the last host among those who manage that kind of agent, because if a matching agent is found at other peers, the request will be returned and not reach the last peer, of course no need to add that request to the waiting list. When a coordinator receives an "available" report, it will check if there exists a waiting request in the waiting list. If that is the case, it removes the item from the list and sends the agent's location to the waiting client setting the "hold" status of that agent.

3.3.5 Creation and Destruction of Agents

The control of an agent includes the management of its location and life cycle. Here, we discuss when and how agents are created or destroyed.

When an agent remains idle for too long in the agent pool (i.e., for a predefined timeout), it is considered useless and needs to be destroyed from the system to save system resource. In this case, the agent will self-destroy and will report its termination to its current coordinator.

When a particular type of agent cannot be found or when several clients are waiting for a certain type of agent, the system needs to find the coordinator who can generate this type of agents and request its creation. Some systems employ special super nodes to manage the life cycle of agents and thus determine their creation. In our fully distributed system, however, there is no need to employ super nodes to control the life cycle of agents since each coordinator manages the location refreshment of the agents that fall under its range of keys, and thus can take care also of it life cycle.

We assume that the coordinators in the network are heterogeneous, and different coordinators may create different kind of agents. When a host finds there is a need to add a new particular type of agent to the agent pool, but it does not have the capability of generating it, it needs to find another host that can accomplish the task. In the following, we describe how to find a coordinator that can create the agents with such kind of services. Both agents and coordinators can be considered as objects with keys $(hashing(characteristic, ID))$. The agent's characteristics correspond to the type of services the agent can offer, while the coordinator's key corresponds to the type of agents it can create. A coordinator plays two roles then: one is the peer in the DHT (i.e., a coordinator), the other is the object (a possible creator with a type associated to it).

When it is considered as a coordinator, its identifier is obtained by hashing its IP; when it is considered as an object, its key is obtain by hashing its $(characteristic, IP)$ pair. Obviously, the location information of both the agent and the coordinator with the same characteristic will be stored in nearby locations. Notice that, if more than one coordinator are in charge of the range of the characteristics, they may not be immediate neighbors, but they will be very close; in other words, either the information are all stored at the same coordinator, or in a group of coordinators. The advantage of this nearby storing is that, if the coordinator that in charge of the agent finds there is a need to create a new agent of the same type, it (or its neighbors) also knows the location of the host that can create such type of agent. There is no need to initiate a new lookup request, the manager of the waiting list can just check its predecessors to find the host that can create that type of agent, and obtain its IP address and send a creation request directly. Further notice that the manager of the waiting list must be the last host that is in charge of such type of agents. For example, the coordinator that theoretically manages key $b_1b_2b_3$11111 will take care of the waiting list. This choice is motivated by the fact that such a host (being the last in charge for a given type of agent) will know if a matching agent has been found or not.

3.4 Summary

3.4.1 Advantages of our Method

In this chapter we have proposed an agent locating scheme based on distributed hashing, as well as a mechanism for reusing mobile agents that are idle in the system. The main advantages of our scheme are the following.

The system is entirely transparent to the client that requests a service. In fact, the client may not know any information about the coordinator or the agent's ID in advance. It just needs to send the request about what kind of services it needs, and it will be assigned a matching agent to accomplish the task. The management of mobile agents and the services offered by agents are separated: the DHT organized by coordinator hosts manages the agents and the agents offer the services.

The system is dynamic. A coordinator peer can leave the system at any time and does not need to call back or destroy the agents it controlled, which are automatically assigned to another coordinator. The agents pool permits the reuse of agents. In fact, idle agents can be reused until they stay idle for a long period of time and terminate themselves. They can be reused even if their former coordinators leaves, and their ongoing execution will not be interrupted.

The system is fully distributed. There is no central control, the coordination of the mobile agents' activities is fully distributed as it occurs in P2P systems.

3.4.2 Future Work

Our idea integrates the concepts of DHT in a mobile agent system. Depending on the characteristics of the network (expected size of the network, memory and performance requirement), other choices could be made in terms of DHT and locating strategy. In this chapter, we adopted Chord (as DHT strategy) and location registration (one of the classic locating strategies) to accomplish the locating. We consider that this combination is efficient especially for Internet like large-scale system. In general, it would be interesting to design the most suitable combination of DHT and location strategies for different scenarios. Such a choice may be affected by the expected size of the network, memory

and performance requirement. For example, our solution is suitable for any size of host network, and is particularly good for large-scale network. If the number of hosts is actually not very large (which is sometimes the case in a agent system) and when other local information is available, there may be some more efficient DHT algorithms to employ [68].

In this chapter, we assume that all the service are equal. The full key only contains a constant number of bits for storing a "flat" characteristic. An interesting direction would be to extended this idea to a multiple service hierarchy where services are organized hierarchically in a tree. We leave this as material for a future investigation.

Another interesting factor to take into account would be the size of the keys. In our examples, we assume the key length is 8, (3 bits for the characteristics and 5 for the ID). Obviously, depending on the features of the systems, on the type of agents available, and on their number, a better choice of this value should be made.

Chapter 4

Locating Mobile Agents through Network Traversal

In this chapter we propose a new strategy for tracking mobile agents in a synchronous network. Our strategy is based on a semi-cooperative approach: while performing its own prescribed task, a mobile agent moves keeping in mind that a searching agent might be looking for it. In doing so we want a fully distributed solution that does not rely on a central server, and we also want to avoid the use of long forwarding pointers. The strategy is based on appropriate delays that the mobile agents must perform while moving on the network so to facilitate its tracking, should it be needed. The searching agent computes a particular searching path that will guarantee the tracking within one traversal of the network. The delays to be computed depend on structural properties of the network. We perform several experiments following different strategies for computing the searching path and we compare our results. Some of the results of this Chapter appeared in [39].

4.1 Introduction

In the classic mobile agents *tracking* (or *locating*) problem: an agent, or a group of agents, is sent on a network to locate a particular agent that is instead moving to perform some tasks. Sometimes the tracking is necessary to communicate with the agent or to terminate its task (e.g., see [11, 60]). In particular, we will focus on two agents only: one is moving to perform its prescribed task (the *moving agent*), the other (the *searching agent*) might be sent on the network to locate the moving agent.

Other problems involving two mobile agents are somehow related to this one: *pursuit evasion*, and *rendezvous*. In pursuit evasion there is a *competitive* setting, where one agent tries to escape, while the other is chasing it. The problem has been extensively studied in deterministic and especially in randomized environments (e.g., see [1, 28, 72]). In rendezvous the two agents *cooperate* to find each other; in fact, their goal is to meet somewhere in the network and their actions go towards this common goal. Also rendezvous has been widely investigated, under different scenarios and different assumption (e.g., see [6] and, for a recent survey [55]). In our problem there is no competition, since the moving agent does not try to escape. On the contrary, there is cooperation, since the moving agent is willing to facilitate the task of the locating agent. However, the degree of cooperation is much weaker than in the rendezvous problem. In fact, the moving agent has other tasks to perform, and it does not even know if some agent has been sent to locate it. While performing its primary tasks it can also perform some actions in order to help the tracking, in case it has to be performed. The agent is willing to do so at the expenses of its own performances, up to a certain degree.

We propose a totally different approach from the typical locating solutions. The idea is to pre-compute a particular *searching walk* that will be followed by the searching

agent whenever the tracking is required. The moving agent moves autonomously and independently to perform its task, without reporting its location and without leaving long traces. The "speed" of its movement, however, is appropriately controlled in such a way that, should the searching agent look for the moving agent, it would locate it within one searching walk. By "controlling the speed" we mean that when the moving agent needs to move over a link, it cumulates a delay (proportional to some network parameter appropriately chosen) before performing the movement. In this chapter the network is assumed to be synchronous; preliminary studies suggest that this assumption could be relaxed for a more realistic environment.

We describe how to compute this appropriate delay, we show that the amount is related to a network parameter called *MinMax chord*. The choice of the optimal search walk is conjectured to be an NP-complete problem. We describe several heuristics to compute various searching walks. We then run some experiments to see what are the performances of the heuristic algorithms in random graphs of various size and degree. The performances measure we consider are the maximum and average delay incurred by the moving, as well as the location time. The results are interesting and motivate further study.

4.2 Model and Terminology

Although the problem might involve several moving agents and several searching agents, without loss of generality we focus on the behavior of a single pair, so we have a *searching agent* SA and a *moving agent* MA. The algorithm we describe would apply to the case of more moving agents.

We assume the system is synchronous, that is, it takes one unit of time for an agent

to traverse a link. The searching agent move at the maximum possible speed (1 time unit per link), while the moving agent "slows down" its movement by waiting an appropriate amount of time at the nodes it encounters on its way.

More precisely, a moving agent will possibly perform certain tasks at a node spending there B time units; when it needs to proceed to the next node, it will immediately proceed if the time already spent at the node is higher than a predefined timeout T, or it will wait $T - B$ time units before moving to the next node. In the next section we will specify how to determine the appropriate timeout T; moreover, we will assume a worst case scenario where the computing time B of the moving agent is zero. Obviously in practice B will be higher thus making MA incurring in a much lower "forced" delay.

We assume that the moving agent is followed by a single forwarding pointer; in other words, a trace of the moving agent arrived at node y from link (x, y) is present at node x as long as the moving agent is in x. This fact guarantees that either the moving agent or its trace are always present on a node (even when the moving agent is in transit on a link). The locating problem is considered solved when SA resides on the same node as MA, or when it finds its forwarding pointer.

4.3 The Searching Walk

The general idea is to determine a traversal of the graph (called *searching walk*) for the searching agent. While the moving agent moves arbitrarily, the searching agent follows this predefined walk. Initially the searching agent is at the "beginning" of the searching walk and the moving agent is obviously "ahead". We will describe how to determine the appropriate delays at each node in such a way that, at each point in time, either the

moving agent (or its forwarding pointer) is caught, or the moving agent is still "ahead".

The searching walk is a particular traversal of the graph, such that all nodes of the graph are visited. Consider any traversal of a graph $G = (V, E)$, let $T = [y_1, y_2, \ldots, y_k]$ be a traversal node sequence of G, where $k \geq n$ (notice that different y_i, y_j could refer to the same node), and the other edges can be considered "chords" (or shortcut of the traversal). An extended weighted graph $G' = (V', E')$, based on the traversal T of G, can be constructed by aligning the traversal nodes (each y_i connected to y_{i+1}) on a horizontal line, mapping each node to its position on T, and adding the other edges of E as chords, as shown in Figure 4.1. If a node is visited more than once, there is a chord in G' only between any two consecutive occurrences of the same node in T, as shown in Figure 4.1, 4.2, 4.3. Moreover, if an edge in E connecting to several occurrences of node u that is visited more than once, there is a chord between that node and the closest occurrence of u, as shown in Figure 4.3, where there is a chord between y_4, and y_6 but no chord between y_2 and y_6. The *length* of a chord $C_{i,j}(y_i y_j, G') = |j - i|$. Given an edge $(y_i, y_j) \in E'$, let $weight(y_i, y_j)$ denote its weight .

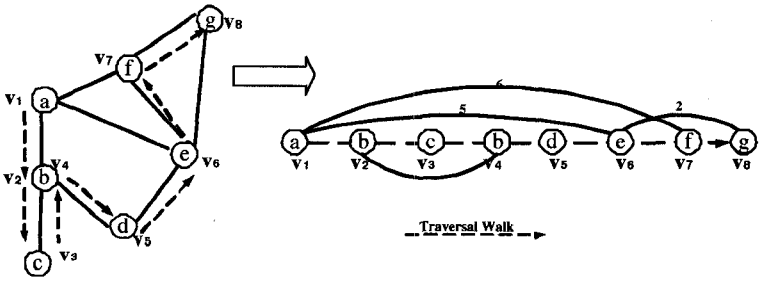

Figure 4.1: Traversal Walk with Chords

More precisely, let $G = (V, E)$ be a graph with n nodes, and let $T = [y_1, y_2, \ldots, y_k]$ be a traversal walk of G (with $k \geq n$). Let $f : V' \Rightarrow V$ be a (non injective) function that, for each element y_i of the traversal, returns the corresponding node $f(y_i)$ of V. Let $E(x)$

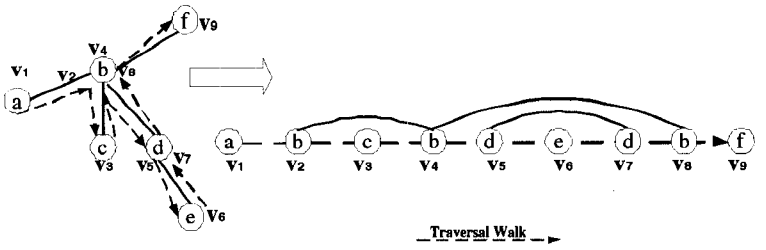

Figure 4.2: b. Traversal Walk with Chords

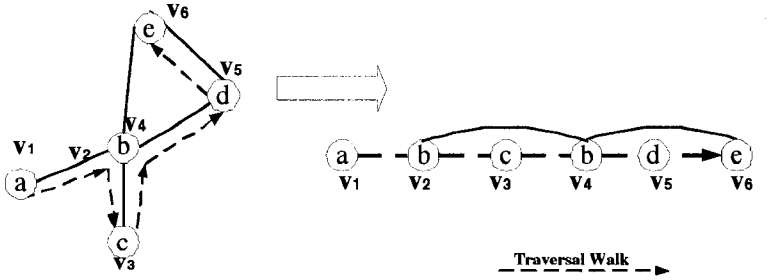

Figure 4.3: c. Traversal Walk with Chords

denote the edges incident to node x in G. We now define a weighted graph $G' = (V', E')$ as follows:

- *Vertices:* $V' = \{y_1, \ldots, y_k\}$ contains one vertex per element of the sequence T;
- *Edges :* $(y_i, y_j) \in E'$ if:

  1) $((f(y_i), f(y_j)) \in E)$ AND (there exists no k $(i < k < j)$ such that $f(y_i) = f(y_k)$ or $f(y_k) = f(y_j)$).

  2) $f(y_i) = f(y_j)$ with $i \neq j$ AND there exists no k $(i < k < j)$ such that $f(y_i) = f(y_k)$.

- *Weight:* Let $y_i, y_j \in T$: $weight(y_i, y_j) = i - j$

Let us call *physical chord* a chord in G' that corresponds to a link in G, and *virtual chord* a chord in G' that connect two occurrences of the same node of G (thus it does

not correspond to any link of G). A virtual chord, in fact, corresponds to a cycle in the traversal walk. For each node $x \in V$, $F^{-1}(x) = \{f^{-1}(x)\}$ contains the corresponding occurrences of x in T.

We now define a node and an edge labeling for graph G. Let $\beta : V \rightarrow Z$ be a node labeling function that associate an integer to each node of the network, and $\lambda_x : E(x) \rightarrow Z$ be an edge labeling function that associate an integer to each edge incident to node x. A global edge labeling function is defined as $\lambda = \{\lambda_x : x \in V\}$. Given a traversal T of G, we label the nodes and the edges of G as follows:

- *Vertices:*

  Let $x \in V$. If $|F^{-1}(x)| = 1$, then $\beta(x) = 0$. If $F^{-1}(x) = \{v_{i,1}, v_{i,2}, \ldots, v_{i,m}\}$ $m > 1$, then $\beta(x) = Max_j\{weight(v_{i,j+1}, v_{i,j})\}$ for $j = 1 \ldots m-1$.

- *Edges :*

  Let $(x, y) \in E$. We define $\lambda_x(x, y) = Max_{a,b,i,j}\{weight(v_{i,a}, v_{j,b})\}$ for $v_{i,a} \in F^{-1}(x)$, $v_{j,d} \in F^{-1}(y))$, and $(v_{i,a}, v_{j,b}) \in E'$.

We compute the delays that the moving agent has to introduce in such a way that the searching agent (SA) is always "behind" the moving agent (MA) along the searching walk, except when it locates it or its forward pointer (FP). With the delays we define below, in fact, it could happen that SA overpasses MA; in this case, however MA is behind the forwarding pointer and it is guaranteed to reach it before it expires.

Suppose the moving agent arrives at node x from link (w, x) and has to move to node y through link (x, y). If $\lambda_x(x, y) < 0$ the agent can move directly without adding any delay because it is moving "away" from the searching agent; if $\lambda_x(x, y) > 0$, it waits the following amount of time. An appropriate amount of time so to be sure that during this

time, either the searching agent finds he moving agent (this will happen if the searching agent is not too far), or the searching agent is still behind the moving agent along the searching walk, or the moving agent is behind, but the forwarding pointer is within reach. The correct delay depends on the label $\lambda_x(x,y)$ of the link the agent has to move on, on the label $\lambda_w(w,y)$ of the link the agent just traversed, and on the label of the node $\beta(x)$.

DELAY

Agent reaching x from (w,x), moving to y through (x,y) in G.

If $\lambda_x(x,y) < 0$

  move-to-y

If $\lambda_x(x,y) > 0$

  WAIT $Max\{\beta(x), \lambda_x(x,y)-1, \lambda_w(w,x)-1\}$

  move-to-y

In other words:

- No delay is added if the movement leads the agent "forward" in the searching path.

- if all the adjacent links of the node in G where the agent resides correspond to physical chords in G', the delay corresponds to the maximum between the weight of the chord the agent wants to traverse (minus 1) and the weight of the chord the agent just traversed (minus 1). Intuitively, the moving agent has to give enough time to the searching agent to reach itself in case the searching agent were traversing y just before the arrival of the moving agent, or to reach its tail in case the searching agent were in x just before the departure of the moving agent.

- if some adjacent links of the node x in G where the agent resides correspond to physical and some to virtual chords in G', the delay corresponds to the maximum between the weight of the virtual chords, the weight of the chord the agent wants to traverse (minus 1) and the weight of the chord the agent just traversed (minus 1). Intuitively, here we

also have to consider that the searching agent is moving along one of the cycles in the searching walk departing from x (because of the existence of virtual links). Before moving away from x, the moving agent has to give the searching agent enough time to finish its cycle and to return and find the moving agent in x.

In the following, let SA denote the searching agent, MA the moving agent, and FP the forwarding pointer. Let $T = [y_1, y_2, \ldots, y_k]$ be the searching walk with SA initially in y_1 when the process starts at time $t = 1$. We want to show that at time t before SA locates MA, either SA is "before" MA or it is between MA and FP. In the latter case, however, SA will reach FP before it expires.

Lemma 4.3.1. *Let MA arrives at node x at time t from node w. Let del be the time MA has to wait before moving to some node y. If by the time $t + del$ MA has not been located, then after the movement of MA towards y, SA is "before" the first occurrence of $\{f^{-1}(y)\}$ in T (i.e., $t + del + 1 < i \ \forall v_i \in F^{-1}(y)$).*

Proof. By induction on the movements of MA. It is true at time 1, when MA moves for the first time. Let the lemma be true when MA arrives to node w and let us prove it is true when MA reaches the next node x. The lemma is trivially true if link (x, y) corresponds to a "forward link" in the traversal path, i.e., if $\lambda_x(x, y) < 0$. Let us then consider only "backward" links.

Consider time t when MA arrives in x. At this time, SA is in y_t and, by induction hypothesis, it is either ahead of MA of it will catch it before MA can move. Now MA waits for

$$Max\{\beta(v), \lambda_w(w, x) - 1, \lambda_x(x, y) - 1\}$$

time units.

Let m and M be smallest and the largest indices such that $y_m, y_M \in F^{-1}(x)$.

We now consider three cases:

- $t < m$.

  If $m - t < \lambda_x(x, y)$, by definition of delay, SA will locate MA before it moves to y (because $del > \lambda_x(x, y)$). Otherwise, SA will be "before" or on y at time $t + del + 1$.

- $m < t < M$.

  Let $y_a, y_b \in F^{-1}(x)$ be the closest occurrences of node x to y_t. SA will reach node x before MA moves to y because, by definition of delay, $del > \beta(x)$ and β is the largest cycle containing x (which is greater than or equal to $b - a$).

- $t > M$.

  SA will reach FA in the next del time units because, by definition of delay, MA stays in x for at least $\lambda_w(w, x) - 1$ time units and thus, FA is in w when SA reaches it at time $t + \lambda_w(w, x) - 1$.

□

It follows from the Lemma that:

Theorem 4.3.1. *The searching agent locates the moving agents by the end of its traversal.*

Clearly, one would like to minimize both the location time (which depends on the length of the searching walk) and the delay incurred by the moving agent. In the following we are interested in searching walks of length $O(n)$ and we would like to minimize the maximum delay as well as the average delay incurred by the moving agent.

The maximum delay corresponds to the longest (virtual or physical) chord; thus, we would need to find the traversal that minimizes such a chord. More precisely, we define

the *MinMax* Traversal $\mathcal{T}_G$ of G as the traversal that minimizes the maximum weight in G':

$$\mathcal{T}_G = Min_T\{Max_{i,j}\{w(y_i, y_j)\}\}$$

with $(y_i, y_j) \in E'$.

4.4 Building Good Searching Walks

We conjecture that finding the *MinMax* traversal of a graph is an NP-complete problem. In the following we propose several heuristic algorithms to construct "good" traversal and we compare them.

4.4.1 General Traversal Algorithm

Consider the following general traversal algorithm that visits the nodes in depth. If the node where the searching agent resides has one unvisited neighbour, the agent moves there; if it has more unvisited neighbours, it chooses one of them and moves there; if there are no unvisited neighbours, it moves through already visited nodes to reach a node that has not been visited yet.

```
Traversal(Vertex start):
    current = start
    While not all nodes visited
        Agent stands at the current node
        if there is only one unvisited neighbour
            Move to that node
        else if there are more unvisited neighbours
            Choose an unvisited neighbour and move
        else if there is no unvisited neighbour
            Move to an unvisited node or to a node with unvisited neighbours
passing through visited nodes
        End while
```

In the general traversal algorithm described above, there are two points where we can introduce some variations for obtaining a searching walk that is good for our purposes.

1. How to choose the next node when the current node has several unvisited neighbours.

2. Where to move after visiting a node without unvisited neighbours.

In the common *depth-first traversal* (DFT), for example, a random choice is performed for 1) and a backtrack for 2). Obviously an arbitrary DFT dose not necessarily result in an efficient searching strategy.

DFS Traversal

```
DFSTraversal(Vertex start):
    DFSStack.push(start)
    current = start
    While not all nodes visited
        next = findNextNode(current)
        current = next
    End while
```

```
findNextNode(Vertex current): return the next unvisited node to be visited in traversal,
current node is the node that the agent stands on
    If current has unvisited neighboring nodes
        nextNode = choosing one unvisited neighboring node
        DFSStack.push( nextNode )
        Return nextNode
    End if
    // there is no unvisited neighboring node, then finding one by backtrack
    lastDFSNode = getLastDFSNode();
    nextNode = choosing one unvisited neighboring node of lastDFSNode
    DFSStack.push( nextNode )
    Return nextNode
```

Backtrack in DFS Traversal

```
getLastDFSNode(): return the last visited node that has unvisited neighboring node in
backtrack
    While (true)
        lastDFS = (Vertex) DFSStack.pop()
        If lastDFS has unvisited neighboring nodes
            DFSStack.push( lastDFS )
            Return lastDFS
        End if
    End while
```

4.4.2 Choosing an unvisited neighbouring node

The following are different strategies to choose one among the unvisited neighboring nodes. Mainly, they are motivated by the empirical observation that it might be useful for our purposes to fully visit a nearby area before moving to another area of the graph.

- **Random**. The agent randomly chooses an unvisited neighboring node.
- **Priority Queue**. When the agent moves to a new node, the unvisited neighbouring nodes of that node are stored in a priority queue. When the agent has more than one unvisited neighboring nodes, it chooses the one with highest priority. If none of the unvisited nodes is in the queue:
  - **Basic**: The agent randomly chooses an unvisited neighboring node.
  - **Improved**: The agent checks the unvisited neighbors of its unvisited neighboring nodes. If it finds at least one of the unvisited neighboring nodes at distance two in the priority queue, it chooses the one with highest priority and it moves there.

- **Closest to the queue**. The agent chooses the one that is closest to a node in the priority queue (in case of ties it selects the one closest to a highest priority element in the queue).

- **Closest to start node**. When the agent has more than one unvisited neighboring nodes, it moves to the unvisited neighboring node that is closest to the start node.

- **Neighbour of least visited node**. When the agent has more than one unvisited neighboring nodes, it moves to the unvisited neighboring node whose neighbour has been visited least recently.

4.4.3 Moving to a non neighbouring unvisited node

The following are different strategies to move to an unvisited node when there is no unvisited neighboring node from the current agent's location.

- **DFT** (Depth-First Traversal). The agent backtracks to the most recently visited node that has unvisited neighboring node, and then continues the traversal.

- **Greedy**. The agent moves to the nearest unvisited node.

- **BFT** (Breadth-First Traversal). The agent moves to one of the unvisited node that is closest to the start node.

- **Hybrid**. The agent moves to the nearest unvisited node if there is only one such node; if there are more such nodes, the agent moves to the node among the nearest unvisited nodes that is closest to the starting node of the walk. This strategy combines the *greedy* and *BFT* strategies.

When considering a non-neighbouring unvisited node, we include in the walk the shortest path between the current node and the next. Notice that, in doing so we might visit new nodes.

The strategies described above (except for *Random* and *DFT*) are motivated by the empirical observation that it might be useful to fully visit an area around a visited node before moving to another area of the graph. In fact, visiting a neighbour of a node already visited creates a chord: since we would like to minimize the length of the chords, we would like to visit these nodes as soon as possible. On the other hand, we also want to maintain a walk of size $O(n)$ and to achieve this we cannot revisit nodes already visited too many times.

Let's see the sample traversal in figure 4.4. Suppose the searching agent moves from node a to node g via b, c, f, h, since there is no unvisited neighbor at node g, the searching agent has to move to a node with unvisited neighbors. Based on the DFS strategy, the agent will backtrack from g to c via h, f; based on the *Greedy* strategy, the agent will move to c directly. When the agent arrives c, there are two unvisited neighbors. Based on the *priority queue* or *closest to start node* strategy, it will move to node i first.

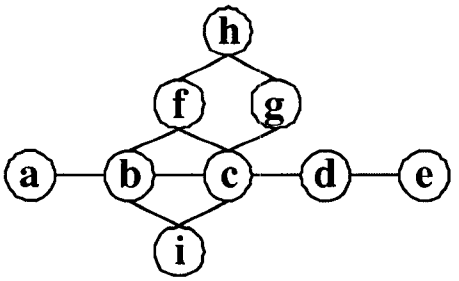

Figure 4.4: DFS and Greedy

4.5 Experimental Results

4.5.1 Graph Generation

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

Generating an arbitrary graph

An arbitrary graph of size n and degree d is 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 that there are no two edges between the same pair and that both a and b still have open ports.

Random graph generation:

- Start with n vertices $1, 2, \ldots, n$. Set $U = 1, 2, \ldots, n$. (U denotes the set of available vertices.)
- 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.
- Create a graph G with an edge from vertex v to vertex s if and only if there is a pair containing these vertices.

- If G is connected, output it, otherwise return to Step 1.

Generating d-regular arbitrary graph

A regular arbitrary graph of size n and degree d is connected graph with n vertices and $(n \times d)/2$ bidirectional edges. The edges are distributed arbitrarily amongst the vertices(i.e.: each vertex has d edges). However 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 the above would create an inefficient algorithm. We have found an algorithm that can quickly generate. The procedure used to generate a d-regular arbitrary graph G 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.

Regular Random graph generation:

- Start with $n \times d$ points $(1,1), (1,2), \ldots, (1,d), \ldots, (n,d) (n \times d$ even) in n groups. Put $U = \{(1,1), (1,2), \ldots, (1,d), \ldots, (n,d)\}$. ($U$ denotes the set of unpaired points.)

- 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.

- 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 group.

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

4.5.2 Experimental Setup

We now combine the different strategies for choosing the next neighbouring unvisited node and choosing a non-neighbouring unvisited node. For each algorithm, we record the length of the traversal walk, the maximum and average delay for the moving agent. Notice that the maximum delay corresponds to the length of the longest chord.

We run the algorithms on different random topologies of various size and density. For each type of graph, we generated 20 graphs with the same parameters, and we averaged the obtained results. We utilize *Java Universal Network/Graph Framework* [48] library to generate the graphs. We do this for random graphs and for random regular graphs of various degree.

The strategies we consider are the ones discussed in Sections 4.4.2 and 4.4.3, and are listed below:

- A1: *random + DFT .*

- A2: *improved priority queue + DFT .*

- A3: *improved priority queue + Greedy.*

- A4: *random + Greedy.*

- A5: *improved priority queue+ BFT.*

- A6: *random + BFT.*

- A7: *improved priority queue + Hybrid.*

We have observed all the results for all the combinations, we however show here only the ones from which we have obtained the most interesting results.

In this set of experiments we are interested in the maximum and average delay incurred

by the moving agent, and in the location time, depending on the traversal strategy followed by the searching agent. In all cases, the length of the traversal is only slightly higher than the number of nodes.

In the experiments, we generated 20 random graphs for each type of graph, and obtained the average values for the results. We have run experiments for various graph sizes n ($n = 100, 200, 500, 800, 1000$) and levels of density m (number of edges) ($m = 4n, 5n, 6n, 7n, 8n, 9n, 10n$). For each choice of n and m we generated 20 random graphs to count the average values. In each case we randomly select the starting node.

The graphic in Figure 4.5 *a*) shows that, among our heuristics, the ones with the best performance in terms of the length of the maximum chord are A_3 and A_7; that is: the *improved priority queue* heuristic as the choice of an unvisited neighbouring node, and either the *Greedy* or the *Hybrid* heuristic for the choice of a non-neighbouring unvisited node. This graph correspond to the case $n = 1000$; the results are quite consistent with different sizes of the graph.

Interestingly, for all heuristics, increasing the density of the network (i.e., its average degree), results in a decrease of the length of the traversal and, most of the times, also of the length of the maximum chord and average delay.

The graphic in Figure 4.5 *b*) shows the changes in average delay incurred by the moving agent as the number of edges increases, for the different strategies. Also in this case, the best performances are obtained by A_3 and A_7.

The various results for the plain DFT and some combinations of heuristics are reported in Appendix A for random graphs and in Appendix B for random regular graphs. We report below (in Tables 4.1, 4.2, 4.3, and 4.4) only the results for the case of random graphs with $n = 1000$.

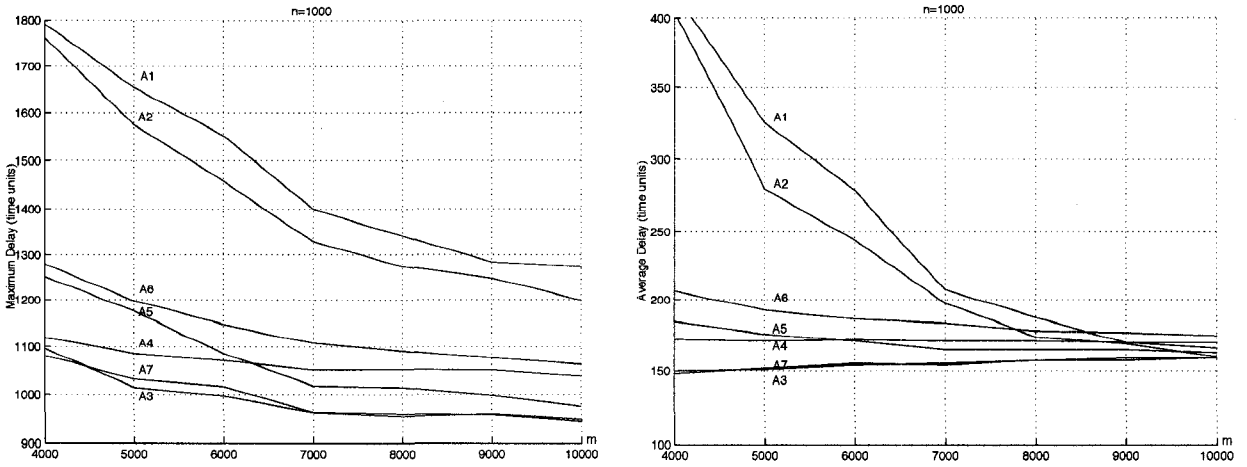

Figure 4.5: Comparison of techniques for random graphs when $n = 1000$: *a*) Max delay, *b*) Average delay.

m	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
4000	1791	418	791	1855		1081	148	509	1163
5000	1654	325	712	1705		1031	152	501	1124
6000	1551	279	669	1617		1031	152	501	1124
7000	1400	207	608	1464		964	154	498	1075
8000	1342	187	587	1393		960	157	479	1067
9000	1284	170	554	1319		958	159	502	1060
10000	1273	166	536	1321		944	159	491	1049

Table 4.1: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 1000$.

4.5.3 Observations

The experiments lead to the following observations.

1. With our best combinations of heuristics (*priority queue and hybrid* and *priority queue and greedy*), the maximum delay is reduced by approximately half compared to a plain Depth-first traversal; the average delay is reduced of 75 %. The reductions are more evident when the graph is sparse, they become less relevant when the graph is very dense (see Figure 4.5 and tables 9). Notice that there are graphs (for example the ring) where

m	A_4: Random + Greedy				A_5: Improved+BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
4000	1120	172	506	1149	1248	183	621	1329
5000	1086	171	502	1110	1132	175	609	1240
6000	1070	172	502	1089	1063	168	562	1178
7000	1051	171	499	1076	1024	166	556	1144
8000	1052	171	486	1065	1005	164	545	1116
9000	1041	170	497	1057	996	164	514	1098
10000	1037	170	492	1051	976	163	504	1082

Table 4.2: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 1000$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
4000	1761	404	891	1854	1096	150	521	1167
5000	1574	284	812	1660	1014	151	524	1121
6000	1458	244	731	1565	997	154	502	1095
7000	1330	197	703	1457	963	155	503	1076
8000	1274	174	615	1382	954	157	498	1067
9000	1248	169	610	1345	960	157	499	1059
10000	1198	160	599	1300	948	159	502	1050

Table 4.3: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 1000$.

it is impossible to find a walk of size $O(n)$ with a maximum chord of length smaller than $O(n)$.

2. In general, with any heuristic, the maximum delay is slightly shorter than the length of the traversal walk.

3. The choice of the heuristic to move to a non-neighbouring unvisited node deeply affects the performances. The traversals with Greedy or Hybrid heuristics have generally the best performance, followed by the BFT, while DFT leads to the worst performance. This confirms the intuition that it is more efficient to fully visit a neighbourhood of a visited node before proceeding to nodes that are further apart.

4. While the maximum delay always decreases with the increase of the number of edges,

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
4000	1279	206	625	1329
5000	1197	192	611	1231
6000	1146	186	572	1168
7000	1108	183	577	1135
8000	1091	178	523	1111
9000	1076	177	502	1097
10000	1062	175	501	1080

Table 4.4: A_6: maximum/average delay, location time, length of the walk. $n = 1000$.

the average delay is more stable and does not display this strong behavior.

5. Combinations A_4 (Random+ Greedy) and A_5 (Improved+ BFT) behaves in a similar way and they both improve the plain depth-first walk. Interestingly, A_4 outperforms A_5 for sparse graphs, while it becomes less efficient then A_5 for more dense graphs. In our experiments this happens roughly when $m = 6n$. Our intuition for this behavior is that the BFT heuristic heavily depends on the density; in fact, when the graph is too sparse BFT gives little chance to find a "good" (close to the start) unvisited node; increasing the number of edges, however, it is more likely that such a "good" node is found thus discovering shorter chords. On the other hand, the Greedy heuristic does not depend on the density since it can always move to the nearest visited node.

6. An interesting observation is that the performances are proportional to the walk's length, in the sense that strategies with longer walks give generally worse performances than strategies with shorter walks. The only exception is A_7 when compared with A_4. The performances of A_7, in fact, are better than those of A_4 although the length of the traversal walk of A_7 is slightly higher (at least for densities $m < 10n$).

7. Fixing the average degree and changing the graph size, the average and maximum delay increases linearly with the graph size. Furthermore, the lower is the minimum degree, the

more the choice of the heuristic affects the performance.

8. In random regular graphs we obtain similar simulation results regardless of the starting node. Thus, the average performances are more stable than in the case of random graphs (in random graph, in fact, the starting nodes will affect the performance deeply).

9. With the same simulation conditions (same number of nodes and edges) regular graphs tend to have shorter traversal path lengths for all th various combinations of strategies; the same happens for the average delay time and the locating time. The average length of the traversal path in random regular graphs are $5\% \sim 15\%$ shorter than that of random graphs. This is more evident when the number of edges is small, or when the "bad" traversal algorithms are applied, i.e. A1 and A2.

4.6 Summary

The contribution of this chapter is the design of a new approach for locating mobile agents that involves neither forwarding pointers (only a single trace of size one) nor a central server. The approach is based on having the moving agent move at a variable speed, depending on the structural properties of the links it is traversing.

At this stage the algorithm could not seem applicable in practice because it is based on very strong assumptions about the environment, which is assumed to be synchronous. Notice however that synchronicity could be relaxed by slightly increasing the length of the trace left by the agents; in fact, preliminary studies suggest that this approach would work also in environments that are not synchronized provided the moving agent leaves a short trace of length 2. Further notice that while computing the delay we have not taken into consideration the time the agent has to actually spend at each node. Depending on

its tasks, the agents might have to spend a certain amount of time thus decreasing the forced delay.

The overall time needed to locate an agent is still quite high; however, we are currently working on several possible improvements that we believe will highly decrease the average delay. The employ of more than a single searching agent would considerably decrease the locating time. For example, static searching agents could be placed at crucial nodes (the ones which have long chords along the searching walk) while a single searching agent could traverse the walk; alternatively, several searching walk could traverse concurrently thus reducing the locating time.

While here we propose the main idea of exploiting the structure of the network for locating purposes, we are now working on improvements like the ones mentioned above to make the technique applicable and efficient in a real mobile agent environment.

Chapter 5

Locating Mobile Agents through Network Traversal in Asynchronous Networks

In Chapter 4, we proposed a new methodology for locating mobile agents in synchronous systems. The methodology has two ingredients. The first one asks for the moving agent to delay its traversal for a time that depends on the structure of the graph. The second one is the fact that the searching agent moves along a predefined traversal path to track the moving agent without any delay. The agents are assumed to move across one edge in one time unit for all edges, i.e., their move is synchronized. In this chapter, we further study the new methodology in asynchronous systems running simulations when the time for moving across an edge is unpredictable.

5.1 The Setup

Depending on the network traffic conditions, the traveling time for agents may not be exactly one unit time to cross one edge. A variable value E may be applied to the traveling time. It means the real traveling time for a certain edge may be within a range $[1-E, 1+E]$.

To facilitate the simulation and evaluation, we suppose agents take one time unit to move across one edge, which is used for computing the delaying time for the moving agent. Because it is under the real asynchronous environment, the moving agent and the searching agent may take less or longer time than 1 time unit to move across one edge. We compute the delaying time based on the fixed one unit time as the crossing time and the real moving time is $1 \pm E$ for agents. Randomly set the crossing time for the moving agent and searching agent for each moving. Of course, the crossing time falls in the range of $[1-E, 1+E]$. If $1-E < 0$, the crossing time falls in the range of $[0, 1+E]$. To do the simulation, we use $(1 \pm E) \times 100$ during the simulation, then divide 100 for the final results to make the simulation easier. Random graphs are used for the simulation and 200 random graphs are generated for each type of graph. Since it is asynchronous network, the moving agent will wait one more time unit except the normal delay time defined in the chapter 4. The final data is rounded to integers.

Obviously, this simulation methodology in the asynchronous system will not guarantee that the searching agent locates the moving agent within one traversal. In this setting we are interested in evaluating the average delay of the moving agent in the new model, the locating success rate and the location time when the moving agent is successfully located. The simulations show that the success rate mostly depends on the E and is irrelevant to the different locating algorithms, so we only record success rate for strategy $A1$.

5.2 Results and Observations

The tables describing the result of the various combination of strategies for asynchronous networks and $n = 1000$ are below, the tables for other values of n are reported in the Appendix.

m	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
4000	1789	419	798	1865		1080	149	508	1162
5000	1651	315	710	1702		1030	150	502	1119
6000	1552	280	670	1607		1018	156	511	1099
7000	1400	207	608	1464		966	155	498	1085
8000	1322	188	588	1394		963	158	478	1065
9000	1283	171	559	1318		959	160	499	1061
10000	1274	165	547	1322		946	161	489	1048

Table 5.1: Results for A_1 and A_7 with $n = 1000$.

m	A_4: Random + Greedy					A_5: Improved+BFT			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
4000	1119	172	508	1148		1245	181	620	1320
5000	1087	172	501	1110		1130	176	601	1241
6000	1071	171	501	1088		1062	167	552	1176
7000	1050	171	498	1077		1026	166	548	1141
8000	1052	170	489	1064		1010	165	542	1106
9000	1040	171	498	1058		998	164	511	1088
10000	1038	171	492	1050		975	164	502	1084

Table 5.2: Results for A_4 and A_5 with $n = 1000$.

In the simulations, we compute the delay time based on the agents moving on an edge in 1 time unit while the real crossing time is randomly chosen from $[1 - E, 1 + E]$. Thus, we obtain the results for the maximum delay, average delay, location time and length of the walk are similar as the synchronous results. The average meeting time is around the $30 \sim 45$ percentage of the full traversal path length. With the increasing of the graph size, the percentage is increasing a little bit. With the increasing of the graph size, the

m	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
4000	1751	404	890	1850		1095	152	516	1168
5000	1572	285	819	1662		1016	151	514	1122
6000	1468	246	721	1568		998	156	507	1097
7000	1331	198	705	1456		960	152	505	1074
8000	1272	173	618	1379		951	158	499	1060
9000	1246	168	613	1347		958	156	492	1055
10000	1188	162	597	1302		949	149	492	1049

Table 5.3: Results for A_2 and A_3 with $n = 1000$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
4000	1267	208	625	1329
5000	1198	193	611	1231
6000	1136	186	572	1168
7000	1118	184	577	1135
8000	1092	177	523	1111
9000	1071	174	502	1097
10000	1063	175	501	1080

Table 5.4: Results for A_6 with $n = 1000$.

meeting time trends to be more stable for each single search. The figure 5.1 (b) shows this observation where the number of the nodes is fixed to 800.

Because of the randomly chosen crossing time, the strategy cannot guarantee that the SA locates the MA within one traversal. The simulations show that the locating success rate is irrelevant to the locating algorithms or the size or the number of links of the network. The success rate depends mostly on the value of E. Increasing the E will decrease the success rate. If $E < 1$, it affects the rate little. The success rate decreases sharply after $E = 1.5$. The figure 5.1 (a) shows this observation where the number of the nodes and edges are fixed to 500 and 400 respectively.

m	A_1: Success rate (%)					
	E= 0.1	0.2	0.5	1	1.5	2
4000	100	100	98	87	83	71
5000	100	98	96	88	84	76
6000	100	99	97	89	85	77
7000	100	100	97	88	83	78
8000	100	98	96	90	86	75
9000	100	100	95	90	87	76
10000	100	99	96	91	87	77

Table 5.5: Success rate for $n = 1000$.

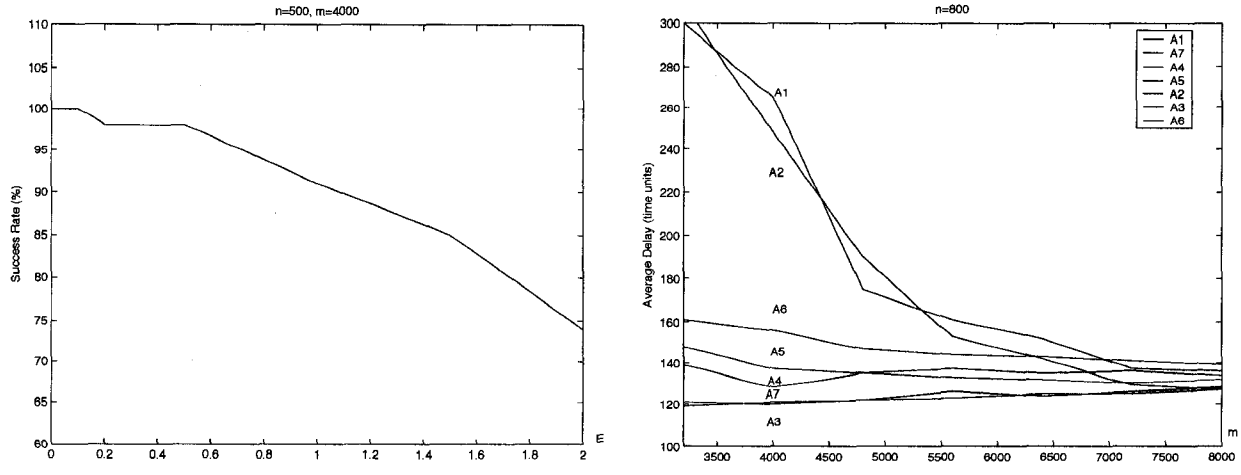

Figure 5.1: Comparisons: *a*) Success rate *b*) Average delay.

Chapter 6

Locating Mobile Agents through Network Traversal in Bounded Transmission Networks

In the last Chapter, we have seen from simulations that the delay time cannot guarantee the SA to locate the MA within one traversal if the agents move totally asynchronously. We now consider a model called bounded transmission network, which is in between the synchronous and the asynchronous ones. Each edge has a bounded transmission time when agents moves across it. Based on the bounded transmission time, the delay time for the moving agent is computed and the SA cannot miss the SM in the traversal.

In the following, we describe the solution to the agent locating problem in a setting where the time it takes to traverse a link is bounded by a fixed predefined amount different from edge to edge (*bounded transmission delay*).

6.1 Definitions

We assume that the time for an agent to move across an edge is bounded by a predefined value and different from edge to edge. Each edge has a weight ω that corresponds to the maximum time it will take for a mobile agent to move across that edge. Thus, whenever an agent move across (x, y) it will take at most time $\omega(x, y)$. The definitions for the traversal path, physical chord and virtual chord are the same as the definitions in the former chapters, but the weights of the chords and the delaying time are changed according to the weighted edges. As in the previous chapters, we assume that the moving agent is followed by a single forwarding pointer.

In the synchronous network, the weight of a chord is defined as: Let $y_i, y_j \in T$: $weight(y_i, y_j) = i - j$ if it is virtual chord ($y_i = y_j$); $weight(y_i, y_j) = i - j + 1$ if it is physical chord ($y_i \neq y_j$). This value is the sum of the number of edges from node y_i to node y_j plus the edge of the chord, and since the agents will not take time to move across a virtual chord, the weight of the virtual chord is just the number of edges from node y_i to node y_j.

In the bounded transmission model the agents will take at most $\omega(x, y)$ time to move across an edge (x, y) instead of the constant one time unit. Similarly to the definition of the synchronous network, the weight of the chord is the sum of the weights for all edges from node y_i to node y_j plus the edge weight of the chord: $weight(y_i, y_j) = \Sigma_{k=i}^{j-1}\omega(y_k, y_{k+1}) + \omega(y_i, y_j)$. If the chord is a virtual chord, the nodes in the traversal path $y_i = y_j$, then it will take no time for the agents to move across the virtual edge, $\omega(y_i, y_j) = 0$. Thus, the weight of the virtual chord is $\Sigma_{k=i}^{j-1}\omega(y_k, y_{k+1})$.

The definitions for the labels of nodes and edges are the same as the ones given in

Chapter 4 but are based on the weights of chords. To distinguish the moving directions, we use the positive and negative symbols for the physical chord labels. The label of a chord where the moving agent moves forward is marked as negative. We always consider the weights for edges and chords are absolute values.

6.2 Bounded Transmission

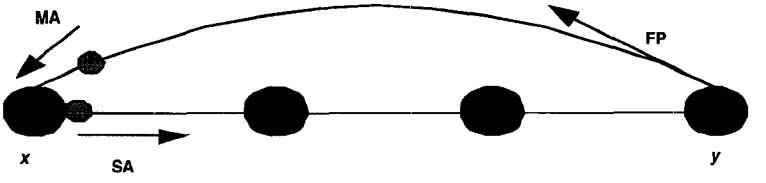

Figure 6.1: Traverse in bounded transmission networks

In the following we consider the extreme case when the transmission delay along a link is precisely ω. If the transmission on a link is some value $t < \omega$ an agent would have to wait an additional amount $\omega - t$ and the result of the section would follow.

To compute the delay time in the bounded transmission model, let us make some observations base on the extreme case of figure 6.1. At some moment, the searching agent just moves away from node x towards node y along the path, while the moving agent is about to arrive to node x. To guarantee that the searching agent does not miss the moving agent, the forward pointer must be at node y when the searching agent moves from node x to y; moreover, the moving agent or the forward pointer must be at x when the searching agent moves across chord (y, x). Obviously, if the moving agent waits $\lambda_x(y, x) = \Sigma_{k=x}^{y-1}\omega(k, k+1) + \omega(y, x)$ at node x, the moving agent will move to node y, and follows the forward pointer at node y to x, then finally locate the moving agent at node x.

We can actually reduce the waiting time since reaching a forward pointer instead of the moving agent itself is enough to locate the MA.

Suppose the moving agent arrives at node x from link (z, x) and has to move to node y through link (x, y). An appropriate amount of time so to be sure that during this time, either the searching agent finds he moving agent (this will happen if the searching agent is not too far), or the searching agent is still behind the moving agent along the searching walk, or the moving agent is behind, but the forwarding pointer is within reach. The correct delay depends on the label $\lambda_x(z, x)$ of the link the agent just traversed (physical chord), and on the label of the node $\beta(x)$ (virtual chord).

DELAY

Agent reaching x from (z, x), moving to y through (x, y) in G.

If $\lambda_x(z, x) < 0$

  move-to-y

If $\lambda_x(z, z) > 0$

  WAIT $Max\{\beta(x) - \lambda_y(x, y), \lambda_x(z, x) - \omega(z, x), \lambda_x(z, x) - \lambda_y(x, y)\}$

  move-to-y

In other words:

- No delay is added if the last movement leaded the agent "forward" in the searching path.

- if all the adjacent links of the node in G where the agent resides correspond to physical chords in G', the delay corresponds to the maximum between the weight of the chord the agent just traversed minus the weight of that edge and the weight minus the weight of the chord the agent wants to traverse.

- if some adjacent links of the node x in G where the agent resides correspond to physical and some to virtual chords in G', the delay corresponds to the maximum between the

weight of the virtual chords, the weight of the chord the agent just traversed minus the weight of that edge and the weight minus the weight of the chord the agent wants to traverse. Intuitively, here we also have to consider that the searching agent is moving along one of the cycles in the searching walk departing from x (because of the existence of virtual links). Before moving away from x, the moving agent has to give the searching agent enough time to finish its cycle and to return and find the moving agent in x (Figure 6.2).

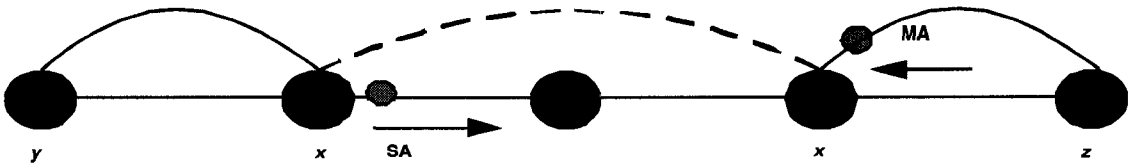

Figure 6.2: Waiting time for virtual chords

In the following, let SA denote the searching agent, MA the moving agent, and FP the forwarding pointer.

Theorem 6.2.1. *Let $T = [y_1, y_2, \ldots, y_k]$ be the searching walk with SA initially in y_1 and let the delay of the MA be defined as above. At any time t before SA locates MA, either SA or its FP is "before" MA or it is between MA and FP. In the latter case, however, SA will reach FP before it expires.*

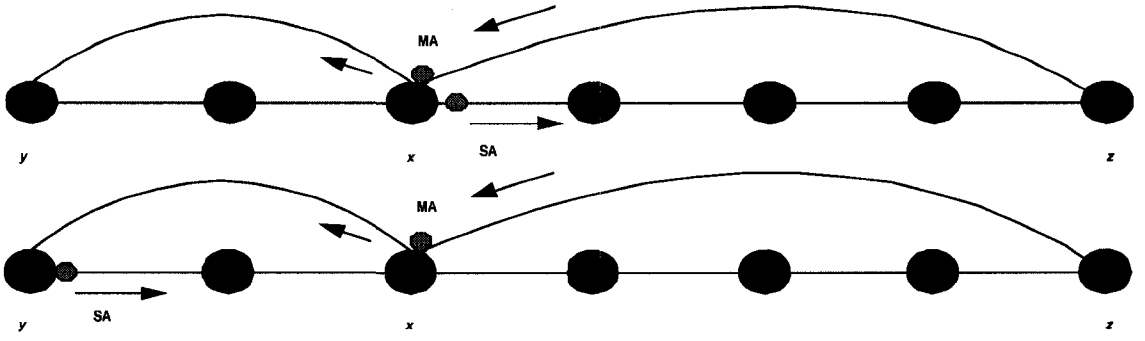

Figure 6.3: Waiting time for physical chords

Proof. By induction on the movements of MA, it is true when the MA starts the search from the first node of the traversal path. At one certain moment, let's see the case shown in Figure 6.3 for the physical chords. Suppose the MA moves from node z to x and will move to node y.

- The SA is moving between nodes x and z when the MA arrives node x.

  To guarantee the FP points to node x exists when SA reaches node z, the MA must wait at node x for the time that the SA takes to move from x to z, which is the sum of all edge weights between the two nodes and also equals $\lambda_x(z,x) - \omega(z,x)$ by definition.

  During the SA moves from node z to x via the chord, the MA may start move to node y and leaves the FP at node x. The FP cannot expire before the SA reaches node x. By the definition for the waiting time, the MA must wait at node y for at least $\lambda_y(x,y) - \omega(x,y)$ and it will take time $\omega(x,y)$ to move across the chord edge, thus, the FP will exist at node x for time $\lambda_y(x,y)$ after the MA leaves node x. Since the SA will take time $\omega(z,x)$ to move across the chord from z to x, if $\omega(z,x) - \lambda_y(x,y) \leq 0$, no extra time beyond $\lambda_x(z,x) - \omega(z,x)$ is needed for the MA to wait at node x; $\omega(z,x) - \lambda_y(x,y) > 0$, the MA has to wait $\omega(z,x) - \lambda_y(x,y)$ more to guarantee when SA reaches at node x the FP is still available. The total time for the later case is $(\lambda_x(z,x) - \omega(z,x)) + (\omega(z,x) - \lambda_y(x,y)) = \lambda_x(z,x) - \lambda_y(x,y)$. To sum up, the MA must wait at node x for $Max\{\lambda_x(z,x) - \omega(z,x), \lambda_x(z,x) - \lambda_y(x,y)\}$ to guarantee the SA can locate MA or its FP at node x.

- The SA is moving between nodes y and x when the MA arrives node x. The SA will take at most $\lambda_y(y,x) - \omega(x,y)$ to reach node x and the MA will wait at node x at

least $\lambda_x(z,x)-\omega(z,x)$. The SA can locate the MA at node x if $\lambda_y(y,x)-\omega(x,y) \geq \lambda_x(z,x)-\omega(z,x)$; otherwise, the SA can locate the forward pointer of the MA at node x since the MA will wait at node y at least $\lambda_y(y,x)-\omega(x,y)$ and its forward pointer will exist at node x at least $\lambda_y(y,x)$.

When there exists virtual chords at node x, see Figure 6.2, before moving away from x, the moving agent has to give the searching agent enough time to finish its cycle and to return and find the moving agent in x (Figure 6.2). The weight of the virtual chord is the sum of all weights for the edges in the cycle, which is $\beta(x)$. We can further investigate waiting time on the node with virtual chords. Actually, the SA can locate FP instead of MA at node x. As we discussed above, when the MA moves from z to x, it must wait at least $Max\{\lambda_x(z,x)-\omega(z,x), \lambda_x(z,x)-\lambda_y(x,y)\}$, and its FP will exist at least $\lambda_y(x,y)$. Thus, if $Max\{\lambda_x(z,x)-\omega(z,x), \lambda_x(z,x)-\lambda_y(x,y)\}+\lambda_y(x,y) > \beta(x)$, the MA waits $Max\{\lambda_x(z,x)-\omega(z,x), \lambda_x(z,x)-\lambda_y(x,y)\}$ at node x is enough; otherwise, the MA has to wait $\beta(x)-\lambda_y(x,y)$.

To sum up, the MA waits $Max\{\beta(x)-\lambda_y(x,y), \lambda_x(z,x)-\omega(z,x), \lambda_x(z,x)-\lambda_y(x,y)\}$ at node x such that the SA can locate the MA or its FP at node x when the MA moves from a "backward" chord.

We now discuss why the MA does not wait when it moves from a "forward" chord. See Figure 6.4, we suppose the MA moves from node z to x then to y, and the chord(z,z) is forward chord and the MA will not wait at node x.

- Chord (x,y) is also forward chord. When the SA reaches node x, it will locate the FP if the MA is moving across the chord; or, if the MA already leaves the node y and continue move forward, the SA will locate it sooner or later since the MA moves forward away; if the MA already leaves the node y and moves backward, the

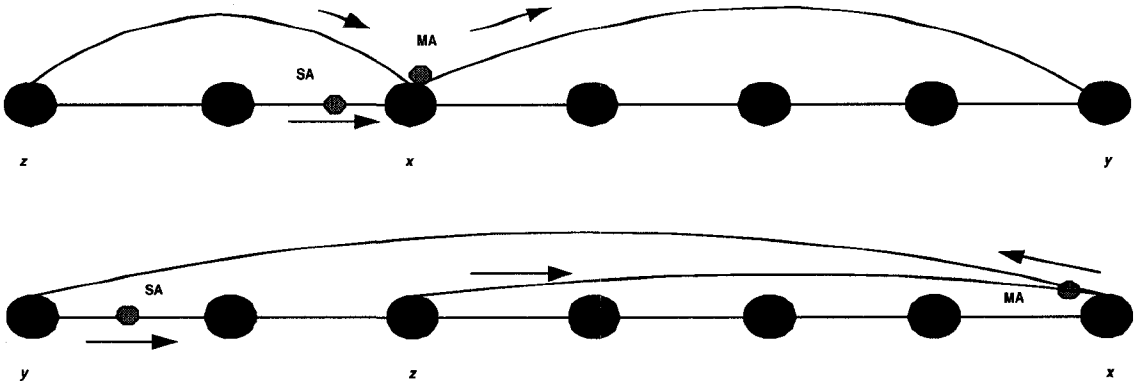

Figure 6.4: Moving from forward chord

prove is the same as the following case.

- Chord (x, y) is also backward chord. The SA will locate the FP at node y since the MA will wait $\lambda_x(z, x) - \omega(z, x)$ at node y. Once the SA locate the FP, the SA will not miss the FP or MA in the further searching and then locate the MA finally since the SA moves "faster" than MA.

□

Based on the above proof, the searching agent locates the moving agent by the end of its traversal in the bounde transmission delay model.

The above definitions for the weight of chords and the waiting time for the moving agent is applicable when all the weights of edges are the same, say $\omega = 1$. In fact, this version of the delay time for the moving agent would provide an improvement to the one of Chapter 4. The reason is that, the searching agent always tries to locate the forward pointer in the extreme case, while in the former chapter, the searching agent sometimes tries to locate the moving agent instead of its forward pointer in the extreme case.

Chapter 7

Locating Mobile Agents through Network Traversal in Trees

In chapter 4, we presented a new strategy for tracking mobile agents that is based on a semi-cooperative approach. In this chapter, we further study it for *tree* networks. We consider depth first traversals, that is, we study the delay for the searching agent to locate the moving agent when the searching walk is obtained by a depth first traversal (DFT). We describe a distributed algorithm to determine the best DFT, i.e., the one that generates the minimum delay for the searching agent. The corresponding search strategy is are also given.

7.1 Preliminaries

First of all notice that a searching walk of $G = (V, E)$, when G is a tree, creates a corresponding weighted graph $G' = (V', E')$ as defined in Chapter 4, where all chords

are virtual chords. In other words, all the chord in G' correspond to occurrences of the same node of G (and do not correspond to any link of G). The searching walk is a tree traversal, and obviously different traversals will give rise to different chord lengths.

7.2 Terminology

We now introduce some terminology. Given a tree $T = (V, E)$, $\omega(T)$ of T denote the smallest length of the largest chords of all possible tree traversal (we will cal this chord $MinMax$ chord). Let $\omega_x(T)$ denote the smallest length of the largest chords among all possible DFT when starting from node x.

In a tree T, the removal of a link (x, y) will disconnect it into two trees, one containing x (but not y), the other containing y (but not x); we shall denote them by $T[x - y]$ and $T[y - x]$, respectively. T_x denotes the tree T rooted at node x. $|T| = |V| = n$ denotes the size of tree T. Given a rooted tree T_x and one of its neighbors y, let $T_x[y]$ (then $T_x[y] = T[y - x] \cup e(x, y) \cup v_x$) denote the subtree of T_x rooted at x, $\omega(T_x[y])$ denote the length of the minimum value of the maximum chords of all possible tree traversal for the search starting from node x, and $|T_x[y]|$ be the size of the subtree $T_x[y]$.

Since there are no cycles in the tree, the searching agent cannot move from one leaf to another without traversing the internal nodes. In other words, visiting all the leaves implies the visit of all the nodes of the tree. The following follows from simple properties of trees.

Lemma 7.2.1. *In any tree there is always an optimal searching walk that starts from a leaf and ends in a leaf.*

Lemma 7.2.2. *A tree T can always be traversed by an agent with $2n - 4$ moves, and*

cannot be traversed with less than $n-1$*, where* $n = |T|$.

Proof. Based on Lemma 7.2.1, at least two edges in the traversal may be traversed once, and other edges may be traversed at most twice. Thus, maximum number of the moves is $2(|T|-1)-2 = 2n-4$. For example, a star topology (one root has $n-1$ degrees) can achieve such maximum number. The the minimum number of the moves is trivial since every edges must be traversed at least once. For example, a straight line can achieve such minimum moves. □

We suppose that a tree has at least one node that has three or more than three branches in the following study.

7.2.1 Introduction to Saturation

Saturation [78] is a technique to collect information in a tree. Saturation can be started by any number of initiators, and is composed of three stages: (1) the *activation* stage, started by the initiators, in which all nodes are activated; (2) the *saturation* stage, started by the leaf nodes, in which a unique couple of neighbouring nodes is selected; and (3) the *resolution* stage, started by the selected pair.

The *activation* stage is a wake-up: each initiator sends an activation (i.e., wake-up) message to all its neighbours and becomes *active*; any non-initiator, upon receiving the activation message from a neighbour, sends it to all its other neighbours, and becomes *active*; active nodes ignore all received activation messages. Within finite time, all nodes become *active*, including the leaves. The leaves will start the second stage.

Each active leaf starts the *saturation* stage by sending a message (call it M) to its only neighbour, referred now as its "parent," and becomes *processing.* (note: M messages

will start arriving within finite time to the internal nodes.) An internal node waits until it has received an M message from all its neighbours *but one*, sends a M message to that neighbour that will now be considered its "parent," and becomes *processing*. If a *processing* node receives a message from its parent, it becomes *saturated*.

The *resolution* stage is started by the *saturated* nodes; the nature of this stage depends on the application. In this chapter, this stage is used to make all nodes in the tree *saturated*.

In this chapter, since we initiate the tree information collection process from all leaves, only the last two stages will involve in the process. Each edge in the last two stages will transfer two messages, thus, the number of message exchanges is $2(|T| - 1) = 2n - 2$.

7.3 Depth-First Traversal Searching Walks

We now consider locating strategies based on depth-first traversals. We first apply *Saturation* to collect useful tree information that will be used to find a searching walk. Based on the collected information, the optimal traversal for all possible DF traversals can be found. We finally prove the correctness.

7.3.1 Some Properties of DFT

Based on the DFT, we can easily obtain the following lemmas.

Let T be a tree rooted in x with $k \geq 3$ children which are root of subtrees (see Figure 7.1(a)). Let $\omega_x(T_x[y_i])$ denote the minimum value of the maximum chord lengths of all possible traversal for traversing the subtree $T_x[y_i]$ starting from x, and $\omega(T_x)$ the

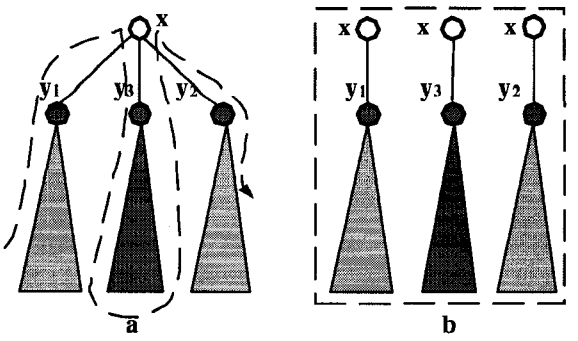

Figure 7.1: Rooted Tree Traversal

minimum value of the maximum chord lengths of all possible traversal for traversing T_x with root x.

Lemma 7.3.1. *Let $y_1, y_2, \ldots, y_k$ be the $k \geq 3$ children of x, and assume, without loss of generality, that $|T_x[y_i]| \geq |T_x[y_{i+1}]|$ for all $i < k$. Then $\omega(T_x) = max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$.*

Proof. We first prove that $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$ can always be obtained by a depth first traversal of T. It is evident that $2 \times |T_x[y_3]|$ is not greater than the chords generated by traversing in depth first all the subtrees $T_x[y_i]$, where $i \geq 3$, starting from x and terminating in x. By definition, $\omega_x(T_x[y_i])$ is the minimum value of the maximum chord lengths of all possible traversal for traversing from $T_x[y_i]$ from x without moving back to x. $T_x[y_1]$ can be traversed starting from a leaf and terminating in x (such a leaf can always be found) thus determining $\omega_x(T_x[y_1])$; $T_x[y_2]$ is traversed as last subtree determining $\omega_x(T_x[y_2])$. Thus, $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$ can be obtained by traversing $T_x[y_1]$ first, $T_x[y_2]$ last, and the other $T_x[y_i]$ as intermediate subtrees. We now prove that $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$ is indeed minimal. To traverse T_x, the traversal of all subtrees $T_x[y_i]$ except two, starts from x and terminates in x. The other two subtrees can be traversed as first and last: the traversal of the first does not start in x but terminate in it, the traversal of the last starts in x but does not terminate

in it. The number of moves necessary for the intermediate subtrees are $2 \times |T_x[y_i]|$. Suppose that the first and last subtrees are different from $T_x[y_1]$ and $T_x[y_2]$ (i.e., they are intermediate trees). They will generate a chord with length at least $2\times|T_x[y_1]| > 2\times|T_x[y_i]|$ or $2 \times |T_x[y_2]| > 2 \times |T_x[y_i]|$ to traverse one of them. Since, for$i, j \neq 1, 2$ we have: $2 \times |T_x[y_i]| + 2 \times |T_x[y_j]| + \sum_{l \neq i,j} |T_x[y_i]| < 2 \times |T_x[y_1]| + 2 \times |T_x[y_2]| + \sum_{i \neq 1,2} |T_x[y_i]|$, it follows that $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$ is minimal. The optimal DF traversal for T_x is shown as a dashed curve in Figure 7.1 (a). □

Lemma 7.4.2 shows that, if the two maximum subtrees of root x can be traversed from the leaf to that root at first, or from the root to the leaf at latest, we can obtain the minimum maximum chord for traversing T_x and the related search strategy. This offers an intuitive idea that if we consider each node in T as root respectively, we can obtain the optimal maximum chord, $\omega(T) = min_{x \in V}\{\omega(T_x)\}$.

7.3.2 Information Collection

In this subsection, we show how Saturation can be used to collect, for each node x, the following information about its subtrees: the *size* and the *length*. The size is the number of nodes of the subtree, while the *length* refers to the length of the longest chord for the optimal traversal if the starting node is x. We will use this information when computing the starting point corresponding to the overall optimal traversal.

Algorithm COLLECTION

- Status: S = {ACTIVE, PROCESSING, SATURATED};

S_{INIT} = {ACTIVE};

```
ACTIVE
    Spontaneously
    begin
        Neighbours:= N(x);
        if |Neighbours| = 1 then
            length:= 1;
            size:= 1;
            M:=("SATURATION", length, size);
            parent ⇐ Neighbours;
            send(M) to parent;
            become PROCESSING;
        endif
    end

    Receiving(M)
    begin
        Process_Message;
        Neighbours := Neighbours - {sender};
        if |Neighbours| = 1 then
            Prepare_Message;
            parent ⇐ Neighbours;
            send(M) to parent;
            become PROCESSING;
        endif
    end
```

```
PROCESSING
    Receiving(M)
    begin
        Store:  (length, size) from all branches;
        forall y ∈ N(x) - {parent} do
            length1:= length(i):  i is the largest branch, i ∈ N(x) − {y};
            size2:= size(j):  j is the second largest branch, j ∈ N(x)−{y};
            length:= Max(length1, size2);
            size:= 1 + ∑ size(z) : z ∈ N(x) − {y};
            M:=("RESOLUTION", length, size);
            send(M) to y;
        endfor
        become SATURATED;
    end


Procedure Prepare_Message
begin
    length1:= length(i):  i is the largest branch, i ∈ N(x) − {parent};
    size2:= size(j):  j is the second largest branch, j ∈ N(x) − {parent};
    length:= Max(length1, size2);
    size:= 1 + ∑ size(z) : z ∈ N(x) − {parent};
    M:=("SATURATION", length, size);
end
```

7.3.3 Derivation of the Optimal Depth First Search Algorithm

This section is devoted to the description of a linear distributed algorithm returning an optimal DF traversal strategy for trees. Let $T = (V, E)$ be a tree with $n = |T|$ nodes. In a tree, the maximum chord by optimal DF search and the number of movement can be affected by:

- start node for searching
- size of the subtrees
- topology of the subtrees

Since the choice of a start node for the search affects the maximum chord, let $\omega_x(T)$, denote the minimum values of maximum chord lengths of all possible traversal starting from $x \in V$, that is: $\omega(T) = min_{x \in V}\{\omega_x(T)\}$.

The most important property of our search strategies is given by the following lemma.

Lemma 7.3.2. *Let $y_1, y_2, \ldots, y_k$ be the $k \geq 2$ children of y in T_x, and assume, without loss of generality, that $|T_y[y_i]| \geq |T_y[y_{i+1}]|$ for all $i < k$. Then $\omega_x(T_x[y]) = max\{\omega_y(T_y[y_1]), 2|T_y[y_2]|\}$.*

Proof. The proof is similar as Lemma 7.4.2. We decompose T_x and study the tree $T_x[y]$ rooted in y. Differently from Lemma 7.4.2, the searching agent starts its search in $T_x[y]$ from node x. To traverse $T_x[y]$ with root x, all subtrees $T_y[y_i]$ except one need to start the traversal from y moving back to y to continue the traversal. Thus, the moves for those subtrees are $2 \times |T_x[y_i]|$. Only one subtree can be traversed last, thus not requiring to

move back to y obtaining minimum value of $\omega_y(T_y[y_i])$. If the one subtree is not $T_y[y_1]$, the traversal of $T_y[y_1]$ will require $2 \times |T_y[y_1]|$ moves. Following the same reasoning as in Lemma 7.4.2 we can obtain $\omega_x(T_x[y]) = max\{\omega_y(T_y[y_1]), 2|T_y[y_2]|\}$. □

This lemma gives the intuitive idea that if we can recursively apply this method from any node to leaves, we can obtain the ω for any subtrees.

Labeling and Optimal Chord

We now describe how to use the information collected in Algorithm COLLECTION to compute the starting location corresponding to the best traversal path.

Given a node $x \in V$, consider the following local labeling λ_x of the edges incident on x. Let $e = \{x, y\}$ be a edge incident to x. If y is a leaf, then $\lambda_x(e) = 1$. Otherwise, let $z_1, \ldots, z_{degree(y)-1}$ be the neighbors of y distinct from x. Let $\ell_{z_i} = \lambda_y\{y, z_i\}$ and $|T_y(z_i)|$ be size of the subtree rooted at y, without loss of generality, $|T_y(z_i)| \geq |T_y(z_{i+1})|$. Then $\lambda_x(e) = max\{\ell_{z_1}, 2|T_y(z_2)|\}$.

Clearly, to obtain label $\lambda_x(e)$, we need to know the size of the subtrees δ, applying Algorithm COLLECTION. If y is a leaf, then $\delta_x = 1$. Otherwise, $\delta_x = 1 + \sum_{i=1}^{k-1} |T_y(z_i)|$, $k = degree(y)$.

Notice that every link $e = \{x, y\} \in E$ will be assigned two labels $\lambda_x(e)$ and $\lambda_y(e)$. We first prove that the labels assigned by the λ_x's can be computed in $O(n)$ messages. Each message contains the information of the label and size of subtrees (ℓ, δ).

Lemma 7.3.3. *All links can be labeled according to the λ_x's in sequential time $O(n)$, and distributively with $O(n)$ messages.*

Proof. Distributively, it can be easily done with $O(n)$ messages using *saturation* as described in section 7.3.2. Each message consists of a label ℓ and the size of the subtree δ that will be used for future computation. Initially, all nodes are *ready*. Each leaf x independently sends a message $(\ell, \delta) = (1, 1)$ to its only neighbor, and becomes *active*. A ready node x, upon receiving a message (ℓ, δ) from a neighbor y, will set $\lambda_x(\{x, y\}) = \ell$. When it has labeled all its incident links except one, say $\{x, z\}$, node x will then compute $\lambda_z(\{x, z\})$ and δ_z based on the existing labels $\lambda_x(\{x, y\})$ and sizes of subtrees for $y \neq z$. Then x sends $\lambda_z(\{x, z\})$ and δ_z to z and becomes *active*. An active node x, upon receiving a message (ℓ, δ) from the neighbor z, sets $\lambda_x(\{x, z\}) = \ell$. For each neighbor $y \neq z$, x then computes $\lambda_y(\{x, y\})$ and sizes based on the existing labels and sizes of subtrees, and sends $\lambda_y(\{x, y\})$ and sizes to y. Then x becomes *done*.

Indeed, the converge-cast structure of the ready-to-active process insures that a ready node x will eventually receive $degree(x) - 1$ messages and becomes active. The diverge-cast structure of the active-to-done process insures that an active node x will eventually complete and become done. To complete the two phases, each edge will transmit two messages from the two extremities of it. Thus, the total number of messages to accomplish the labeling is $2 \times (|T| - 1) = 2n - 2$. □

We now show the relationship between the λ_x's and the maximum chord of the search strategy.

Lemma 7.3.4. *For each* $e = \{x, y\} \in E$, $\omega_x(T_x[y]) = \lambda_x(e)$.

Proof. By induction on the height $h(y)$ of $T_x[y]$. The lemma trivially holds for $h(y) = 0$ (i.e., y is a leaf) since $\lambda_x(e) = 1 = \omega(T_x[y])$. Assume that the lemma holds whenever $0 \leq h(y) < k$, and consider now the case when $h(y) = k$. Let $y_1, y_2, \ldots, y_d$ be the children

of y in $T_x[y]$, where, w.l.o.g., $|T_y[y_i]| \geq |T_y[y_{i+1}]|$, and $d = deg(y) - 1$ if $y \neq x$ and $d = degree(y)$ otherwise. By definition of λ_x, $\lambda_x(\{x,y\}) = max\{\lambda_y(\{y,y_1\}), 2|T_y[y_2]|\}$. On the other hand, by Lemma 7.4.3, $\omega_x(T_x[y]) = max\{\omega_y(T_x[y_1]), 2|T_x[y_2]|\}$. Since the height of $T_y[y_i]$ is $h(y_i) < k$, then, by induction hypothesis, $\lambda_y(\{y,y_i\}) = \omega_x(T_y[y_i]) = \omega_x(T_x[y_i])$. Thus $\lambda_x(\{x,y\}) = \omega_x(T_x[y])$. □

Consider now the following labeling of the nodes of T which will assign to each node $x \in V$ a label $\mu(x)$. Let $e_1, e_2, \ldots, e_{degree(x)}$ be the $degree(x)$ links incident to x. Assume, w.l.o.g., that $|T_x(e_i)| \geq |T_x(e_{i+1})|$ for all $i < deg(x)$.

Set, if $degree(x) = 1$, $\mu(x) = \lambda_x(e_1)$; if $degree(x) = 2$, $\mu(x) = max\{\lambda_x(e_1), \lambda_x(e_2)\}$; if $degree(x) \geq 3$, by Lemma 7.4.2 and 7.4.4, $\mu(x) = max\{\lambda_x(e_1), \lambda_x(e_2), 2|T_x[e_3]|\}$.

We can obtain $\omega_x(T) = \mu(x)$. Thus, $\omega(T) = min\{\mu(x), x \in V\}$.

Since a direct consequence of Lemma 7.3.3 is that all $\mu(x)$'s can be computed in $O(n)$ work serially or distributively, we get that the ω in a tree can be compute serially in $\Theta(n)$ time, and distributively with $\Theta(n)$ messages. Interestingly, our algorithm not only computes the maximum chord, but also provides a way to set up a optimal search strategy, still in linear time, as shown in the following section.

The Optimal DFT Strategy

Suppose that one of the subtree $T_x[x_i]$ of root x is completely searched, then the searcher should never enter such subtree again; if one of the subtrees is partially searched, the searcher should complete it. Then, if all subtrees of x are searched, the search is finished and the searcher has no need to move back to the root; otherwise, the searcher must move back to the root and continue the search in the unsearched subtrees. The moves

to traverse the subtree and move back to x is $2|T_x[x_i]|$. In the above description, if the largest subtree $T_x[x_1]$ is traversed last, the searcher would not have to move back to the root for this tree, thus saving a long trip back.

We now show that the labelings λ and size of subtrees δ contain all the information needed to construct an optimal search strategy. Given an unweighted tree T and the labelings and sizes (λ, δ), consider the search strategy constructed as follows, with $x \in V$ as starting node. By Lemma 7.2.1, we can find a leaf with the smallest label as the search start node, and the searcher moves from it to its neighbor. Suppose the searcher stands at node x, it will traverse all unvisited subtrees except the largest, and move back to x. Then the searcher traverses the largest subtree last. The traversal continues until all nodes are visited.

(a) Choose a leaf x such that λ_x is minimum;

(b) Start the search from x and move to its only neighbor;

(c) Traverse the unsearched subtrees of an internal node y;
Let $y_1, y_2, \ldots, y_{degree(y)-1}$ be the $degree(y) - 1$ neighbors of y, where, w.l.o.g., $|T_y[y_i]| \geq |T_y[y_{i+1}]|$.

(c-1) Traverse the unsearched subtrees $T_y[y_2], \ldots, T_y[y_{degree(y)-1}]$ and move back to y;

(c-2) Traverse the biggest subtree $T_y[y_1]$ and not move back;

(d) Do the traversal like (c) until all nodes are visited;

Figure 7.2: Optimal DF Search Algorithm

Figure 7.4 shows the labeling procedure which implies the ω and traversal route. In the

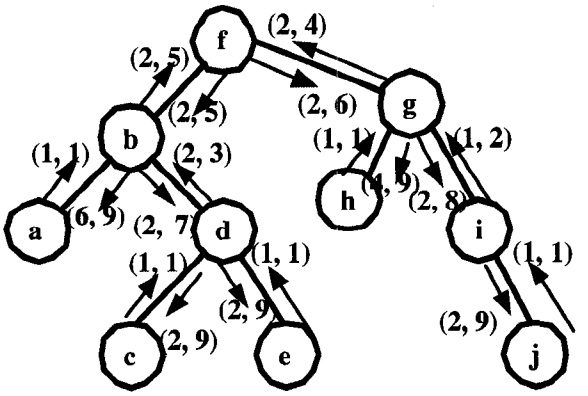

Figure 7.3: Sample Labeling and Traversal

tree, each edge transmits two messages to accomplish the labeling. Leaf nodes c, e, j have the smallest labels, they can be used as starting nodes and the maximum delay time is 2 for the moving agent. In terms of the search strategy showed in Figure 7.2, the traversal walk of the searching agent can be $j \rightarrow i \rightarrow g \rightarrow h \rightarrow g \rightarrow f \rightarrow b \rightarrow a \rightarrow b \rightarrow d \rightarrow c \rightarrow d \rightarrow e$.

We can easily obtain the value of ω and the number of moves for a complete binary tree with the size n, the number of edges $m = n-1$ and the height $h = \log(n+1)-1$ (because $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$). The ω is $2 \times \frac{\frac{n-1}{2}-1}{2} = \frac{n-3}{2}$. The searching agent may start the search from any leaf, and each edge will be visited twice, except $2h$ edges that are visited once. Thus, the number of moves is $2m-2h = 2(n-1)-2(\log(n+1)-1) = 2(n-\log(n+1))$.

Worst Case Situations

Given a tree with nodes n, for all possible non-isomorphic trees, we want to find the ω for the optimal DF search and the corresponding traversal path length.

We fist show that given a subtree with size $n'(n' \geq 2)$, the largest possible label sent from a node to a parent is $n' - 1$.

Lemma 7.3.5. *For subtree $T_x[y]$, $\lambda_x(\{x, y\}) \leq |T_x[y]| - 1$.*

Proof. By induction on the height $h(y)$ of $T_x[y]$, where $T[y-x]$ is the subtree. The lemma trivially holds for $h(y) = 1$ (i.e., all y's child nodes are leaves), since if y has one leaf, $|T_x[y]| - 1 \geq \lambda_x(e) = 1$; if y has more than one leaf, $|T_x[y]| - 1 \geq \lambda_x(e) = 2$. Assume that the lemma holds whenever $1 \leq h(y) < k$, and consider now the case when $h(y) = k$. If $T_x[y]$ has only one child y_1 of y in $T_x[y]$, $|T_x[y]| - 1 > |T_y[y_1]| - 1 \geq \lambda_y(e) = \lambda_x(e)$; If $T_x[y]$ has more than one child, let $y_1, y_2, \ldots, y_d$ be the children of y in $T_x[y]$, where, w.l.o.g., $|T_y[y_i]| \geq |T_y[y_{i+1}]|$, and $d = deg(y) - 1$ if $y \neq x$ and $d = degree(y)$ otherwise. By definition of λ_x, $\lambda_x(\{x, y\}) = max\{\lambda_y(\{y, y_1\}), 2|T_y[y_2]|\}$. If $\lambda_y(\{y, y_1\}) \geq 2|T_y[y_2]|$, $|T_x[y]| - 1 \geq |T_y[y_1]| - 1 \geq \lambda_y(e) = \lambda_x(e)$; if $2|T_y[y_2]| \geq \lambda_y(\{y, y_1\})$, $|T_x[y]| - 1 \geq |T_y[y_1]| + |T_y[y_2]| \geq 2|T_y[y_2]| = \lambda_x(e)$. To sum up, $|T_x[y]| - 1 \geq \lambda_x(e)$. In other words, given a subtree with size $n'(n' \geq 2)$, the possible maximum value of its up sending label is $n' - 1$. □

Theorem 7.3.1. *Given a tree T with size n, $\omega(T) \leq 2\lfloor \frac{n-1}{3} \rfloor$.*

Proof. First, we prove that there exists a tree such that its ω is $2\lfloor \frac{n-1}{3} \rfloor$. We can construct such tree, suppose there exists a root node x with three branches in the tree, w.l.o.g., $|T_x[x_1]| \geq |T_x[x_2]| \geq |T_x[x_3]|$. Nodes in the tree are distributed as mean as possible for each branch. Thus, $|T_x[x_3]| = \lfloor \frac{n-1}{3} \rfloor$. Based on Lemma 7.4.2, $\omega(T_x) = max\{\omega(T_x[x_1]), \omega(T_x[x_2]), 2|T_x[x_3]|\} = max\{\lambda(T_x[x_1]), \lambda(T_x[x_2]), 2|T_x[x_3]|\}$. In terms of Lemma 7.3.5, $\lambda(T_x[x_1]) \leq \lceil \frac{n-1}{3} \rceil - 1 < 2\lfloor \frac{n-1}{3} \rfloor = 2|T_x[3]|$ and $\lambda(T_x[2]) \leq \lceil \frac{n-1}{3} \rceil - 1 < 2\lfloor \frac{n-1}{3} \rfloor = 2|T_x[3]|$. Thus, $v(T_x) = 2\lfloor \frac{n-1}{3} \rfloor$.

Second, we prove that there is no tree such that its ω is bigger than $2\lfloor \frac{n-1}{3} \rfloor$. Given a tree, suppose x be the center node of the tree and also be considered as the root. Let $x_1, x_2, \ldots, x_d$ be the children of x in T_x, where, w.l.o.g., $|T_x[x_i]| \geq |T_x[x_{i+1}]|$, and $d = deg(y)$. Because of the property of the center in a tree, $|T_x[x_1]| - |T_x[x_2]| \leq 1$. In terms of

Lemma 7.3.5, $\lambda_x(\{x, x_1\}) \leq \lceil \frac{n-1}{2} \rceil$, $\lambda_x(\{x, x_2\}) \leq \lfloor \frac{n-1}{2} \rfloor$, and $|T_x[x_3]| \leq \lfloor \frac{n-1}{3} \rfloor$. In terms of Lemma 7.4.2, $\omega(T_x) = max\{\omega(T_x[x_1]), \omega(T_x[x_2]), 2|T_x[x_3]|\} = max\{\lambda(T_x[x_1]), \lambda(T_x[x_2]), 2|T_x[x_3]|\} \leq 2\lfloor \frac{n-1}{3} \rfloor$. Note that $\omega(T) = min_{y \in V}\omega(T_y)$. Thus, $\omega(T) \leq \omega(T_x) \leq 2\lfloor \frac{n-1}{3} \rfloor$. □

We further study the number of moves for the trees with such ω. Based on the above construction for such kind of tree, the topology of the subtree $T_x[x_3]$ will not affect the moves for traversing, since each edges in that subtree will be visited twice. In terms of Theorem 7.2.2, the possible maximum number of moves for $T_x[x_1]$ and $T_x[x_2]$ can be $2(|T_x[x_1]|-1)-2$ and $2(|T_x[x_2]|-1)-2$, respectively; and the possible minimum number of moves for $T_x[x_1]$ and $T_x[x_2]$ can be $|T_x[x_1]|-1$ and $|T_x[x_2]|-1$. Thus, the number of moves for traversing trees with the possible maximum ω is between $\frac{n-1}{3} + \frac{n-1}{3} + 2\frac{n-1}{3} = \frac{4(n-1)}{3}$ and $2(n-1)-4$.

7.4 Oscillating Traversal Searching Walks

In the previous section, the depth-first traversal strategy is used to do the searching. Unfortunately, in many cases, DFT cannot achieve the optimal maximum delay time(*MinMax* Chord). Consider the example of Figure 7.4. It is easy to see that the the length of the maximum chord using a DFT is 6, which is double the size of each subtree, and corresponds to starting from a leaf of a subtree, traversing the three subtrees in DFT terminating in another leaf. Consider instead the strategy where the first and the last subtrees of the figure are traversed in the same way, but the central subtree is traversed moving from a to each leaf and back, obtaining the sub-path: $a \rightarrow b \rightarrow c \rightarrow b \rightarrow a \rightarrow b \rightarrow d \rightarrow b \rightarrow a$. In this way the maximum chord length can be decreased from 6 to 4. Notice that, while decreasing the delay, we however increase the number of moves of the searching agent in

the central subtree from 6 to 8. If we are willing to increase the number of moves, we can improve on the DFT strategy.

In the following, we modify the DF traversal to obtain the maximum chord close to the *MinMax* Chord in a tree and the corresponding search algorithm for it.

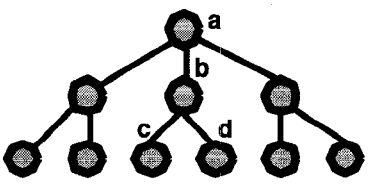

Figure 7.4: Sample

7.4.1 Some definitions

We say that a subtree is *completely searched* when all its nodes have been searched; *completely unsearched* when all its nodes are unsearched; *partially searched* otherwise. We will have each node store a search status to mark each of its branches. An *intermediate node* is a node that connects an area that is completely searched to an area that is not completely searched. We call *monotone search mode* a way of searching where the searching agent does not move in a subtree that is already completely searched. Note that, monotone mode does not mean that a visited node will not be revisited (in a tree this is unavoidable), but that subtrees already fully visited are not re-visited.

In the following we make some observations to understand how to modify the approach of the previous section based on DFT, to be able to obtain a smaller maximum chord.

- First of all, to have a monotone search, the search strategy must guarantee that if the searching agent arrives at an intermediate node from an unsearched area, it

understands it should not enter a completely searched sub-tree. Thus, we have to make sure that the moving agent waits enough time units at the intermediate nodes, in such a way that when the searching agent moves from the intermediate node to one unsearched or partial searched branch and then back to the intermediate node, it can locate the moving agent or its trace.

- Suppose that the searching agent moves from an intermediate node to a **leaf node** in the unsearched area A. To accomplish a monotone search, at this point it could move back to the intermediate node immediately, or continue searching other nodes in A coming back to the intermediate node if such movement does not lengthen the generated virtual chords. This observation shows the facts that, if the searching agent moves to one internal node in the branch and then moves back to the intermediate node, the search work done may become vain because the mobile agent may reside at the leaf node in that branch, and the search needs to visit the leaf node sooner or later. If all leaves in that branch are searched, of course all the internal nodes in that branch are searched too.

- When the searching agent moves from the intermediate node to a leaf node in one branch and some unsearched sub-branches connect to the nodes in the just searched path, if the mobile agent resides in those sub-branches, and it may enter the searched path through the connecting node. We assume that when the searching agent moves from the intermediate node along the just searched path, it will visit the leaf node in the sub-branch that is **closest to** the just searched leaf. Figure 7.5 shows this search strategy. Suppose the searching agent moves from intermediate node a to search the middle branch. It may moves along $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow d \rightarrow c \rightarrow b \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow f \ldots$ The completely searched sub-branches in the branch

will not be searched again. This search continues until all nodes in the branch are completely searched. From the point of view of nodes/sub-branches searched order, the effect seems more like a DFS.

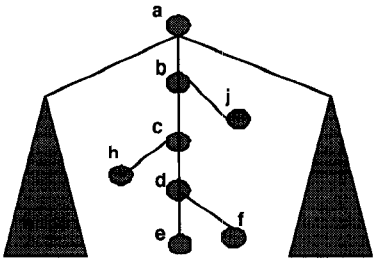

Figure 7.5: pseudo-DFS

The difference between the real DFS and the third assumption is that, for a real DFS in a subtree, the searching agent searches all the nodes in that subtree, then it moves back to the intermediate node. For the pseudo-DFS, it will move back and forth between the intermediate node and the leaf nodes, and the completed searched sub-branches are more like depth first searched.

A node is called "DONE" when itself and all of its neighbors except one are searched. Then, it will move to that unsearched neighbor, in turn, that node becomes intermediate node.

The searching agent can, at least, move back and forth from the intermediate node to the leaf nodes in the partial searched subtree.

One trick can be used to reduce the traversal path. In Figure 7.5, the depth for the subtree is 4 and its maximum virtual chord is 8. To traverse leaf h and i, the searching agent can moves along path $a \rightarrow b \rightarrow c \rightarrow h \rightarrow c \rightarrow b \rightarrow j \rightarrow b \rightarrow a$ instead of path $a \rightarrow b \rightarrow c \rightarrow h \rightarrow c \rightarrow b \rightarrow a \rightarrow b \rightarrow j \rightarrow b \rightarrow a$, and it will not increase the maximum virtual chord for intermediate node a. It means, if the traverse path in subtree $T_x[y]$

matches two conditions: the generated virtual chord is not greater than $2 \times D(T_x[y])$ and the *closest-to* rule mentioned above, the searching agent can traverse as many as possible leaves in one back-forth round to reduce the traversal path.

We say the depth of subtree $T_x[y]$ is that, the longest distance from node x to all leaf nodes, $D(T_x[y])$.

Lemma 7.4.1. *Suppose a root x has at least three branches, part of the search strategy is that the searcher moves from one branch to x and completely traverses the subtree $T_x[y]$, then moves to another branch via x. Assume the maximum chord length for that search is $\omega'(T_x)$, $\omega'(T_x) \geq 2 \times D(T_x[y])$.*

Proof. To visit all nodes in subtree $T_x[y]$ from x, the searching agent will move back and forth between the intermediate node and the leaf nodes. If the searching agent moves to one internal node in the branch (not to the leaf node) and then moves back to the intermediate node, the leaf node must be searched in the future. It will obtain no shorter maximum chord length for completely search such branch, comparing to moving to the leaf node at the beginning. As a consequence, the maximum chord length must be no smaller than the double depth of the subtree. □

Lemma 7.4.2. *Let $y_1, y_2, \ldots, y_k$ be the $k \geq 3$ children of x, and assume, without loss of generality, that $D(T_x[y_i]) \geq D(T_x[y_{i+1}])$ for all $i < k$. Then $\omega(T_x) = max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2D(T_x[y_3])\}$.*

Proof. Firstly, we prove that $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2D(T_x[y_3])\}$ is not greater than the chord length based on back-forth traversing the T. It is evident that $2 \times D(T_x[y_3])$ is not bigger than the chords generated by back-forth traversing all the subtrees $T_x[y_i]$, where $i \geq 3$, from x in the first and moving back to x at least. In terms of the definition of

$\omega_x(T_x[y_i])$, it is the minimum value of the maximum chord lengths of all possible traversal for traversing from $T_x[y_i]$ from x and do not move back to x. If $T_x[y_1]$ is traversed firstly from a leaf to x with $\omega_x(T_x[y_1])$, and such leaf can always be found; and $T_x[y_2]$ is traversed lastly with $\omega_x(T_x[y_2])$. Thus, $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2|T_x[y_3]|\}$ is not bigger than the chord length based on back-forth traversing the T. Secondly, we prove it is minimal for such traversal. To traverse T_x with root x, all subtrees $T_x[y_i]$ except two ones need to start the traversal from x in the first and moving back to x at least to continue the traversal. Thus, the maximum number of moves for each back-forth round for those subtrees are $2 \times D(T_x[y_i])$. The two exceptional ones can be traversed in the first and at latest, such that they do not need to traverse from x (traversed in the first) or move back to x (traversed at latest), their minimum speed can be $v_x(T_x[y_i])$. If the two exceptional subtrees are not $T_x[y_i](i = 1, 2)$, it will generate chord length with $2 \times D(T_x[y_i])$ to traverse one of them. $2D(T_x[y_i]) \geq \omega_x(T_x[y_i])$, and the definition of $\omega_x(T_x[y_i])$, $max\{\omega_x(T_x[y_1]), \omega_x(T_x[y_2]), 2D(T_x[y_3])\}$ is minimal. □

Given a rooted tree T_x, and one of its neighbors y, let $T_x[y]$ denote the subtree of T_x rooted in y, and let $\omega_x(T_x[y])$ denote the minimum values of maximum chord lengths for back-forth traversing $T_x[y]$ from x. The most important property of our search strategies is given by the following lemma.

Lemma 7.4.3. *Let $y_1, y_2, \ldots, y_k$ be the $k \geq 2$ children of y in T_x, and assume, without loss of generality, that $D(T_y[y_i]) \geq D(T_y[y_{i+1}])$ for all $i < k$. Then $\omega_x(T_x[y]) = max\{\omega_y(T_y[y_1]), 2D(T_y[y_2])\}$.*

Proof. The proof is similar as Lemma 7.4.2. We decompose T_x and study the tree $T_x[y]$ with reserving x. Now we consider y as the root in the new tree. Different from

Lemma 7.4.2, the searching agent starts search from x, for the tree rooted at y. To traverse $T_x[y]$ with root x, all subtrees $T_y[y_i]$ except one need to start the traversal from y in the first and move back to y at least to continue the traversal. Thus, the moves for each back-forth round for those subtrees are not bigger than $2 \times D(T_x[y_i])$. The exceptional one can be traversed at latest, such that it does not need to move back to y, the optimal chord length can be $\omega_y(T_y[y_i])$. If the exceptional subtree is not $T_y[y_1]$, it may generate chord with length $2 \times D(T_y[y_1])$ to back-forth traverse it. Since $2D(T_y[y_1]) \geq \omega_y(T_y[y_1])$, and the definition of $v_y(T_y[y_i])$, we can obtain $\omega_x(T_x[y]) = max\{\omega_y(T_y[y_1]), 2D(T_y[y_2])\}$. □

In the saturation, the *depth* and the *length* are collected for the subtrees. The label $\lambda_x(e)$ can be obtained by

Lemma 7.4.4. *For each* $e = \{x, y\} \in E$, $\lambda_x(e) = \omega_x(T_x[y])$.

7.4.2 Traversal Algorithm

At any moment, one subtree of a rooted tree T_x may be *complete searched*, *unsearched* and *partial searched*. The intermediate node denotes the node that connects the complete searched subtrees and more than one partial searched or unsearched subtrees.

The oscillating traversal starts from one leaf and terminates when all nodes are traversed. It traverses all nodes in one subtree by moving from the intermediate node to one leaf of that subtree and back to the intermediate node, then initiate another back and forth round to traverse another leaf until all leaf are traversed. If all leaf nodes in the subtree are searched, of course all nodes in that subtree are traversed. When all subtrees except one are complete searched, the searching agent will move to the node that connects the current intermediate node and the one unsearched subtree. That node will become

the new intermediate node. The searching agent will not move back to the subtree that are complete traversed. The algorithm is shown in Figure 7.6.

(a) Choose a leaf x such that λ_x is minimum;

(b) Start the search from x and move to its only neighbor;

(c) When the searching agent stands at an intermediate node:

(c-1) If there is no partial searched subtree, it chooses an unsearched subtree without longest depth to search.

(c-2) If there exists a partial searched subtree, it tries to completely search that subtree.

(d) When all subtrees are completely searched, except the subtree with the longest depth, move to that subtree and do (c);

(e) When all nodes are traversed, the search terminates;

Figure 7.6: Optimal Oscillating Search Algorithm

7.5 Simulations for Traversal in Trees

In the tree, the generated traversal path can only contain virtual chords. Obviously, if a node has only one or two edges, the moving agent at that node will not wait since the searching agent can locate it regardless of its movements. If the node has three or more neighbours, the moving agent may wait at that node for the time of its maximum virtual chord length.

Based on the DFT traversal algorithm and oscillating traversal algorithm, we can

find a leaf node with smallest label value and consider that node as the starting node. To compare with other traversal algorithms, we randomly choose a node as the starting node, then perform a depth-first search traversal and generate the traversal path. We randomly generate 20 graphs for each type of trees for the simulation. Then we obtain the average values for the maximum delay time and the average delay time and the locating time respectively.

n	Saturation-DFS			
	Max. D.	Av. D.	Loc. T.	Length
100	32	1.33	75	182
200	65	1.98	166	386
500	173	2.33	411	983
800	267	2.61	675	1582
1000	352	2.72	943	1980

Table 7.1: Locating performance in Tree by DFS

n	Saturation-Oscillating			
	Max. D.	Av. D.	Loc. T.	Length
100	11	1.03	276	765
200	13	1.24	452	1489
500	18	1.29	1875	6893
800	20	1.48	3121	8975
1000	21	1.77	4876	12034

Table 7.2: Locating performance in Tree by Oscillating

The simulations show that increasing the size of trees, the maximum and average delays are increased accordingly. For the oscillating traversal algorithm, the maximum delay increases slightly, and its traversal path and locating time increase sharply compared with the other two traversal algorithm. Although the maximum delay for depth-first traversal is much greater than that of the oscillating traversal, the differences for the average delay time are not very large and the locating time for depth-first traversal is

n	Random DFS			
	Max. D.	Av. D.	Loc. T.	Length
100	182	2.12	68	189
200	368	2.84	145	387
500	821	3.53	376	898
800	1489	4.08	534	1505
1000	1902	4.51	765	1982

Table 7.3: Locating performance in Tree by Random

much shorter. Thus, we conclude that the saturation based depth-first traversal is a good traversal algorithm to locate the agent in a tree.

Chapter 8

Locating Mobile Agents through Network Traversal in Meshes

In Chapter 4, we presented a new strategy for tracking mobile agents that is based on a semi-cooperative approach. In Chapter 7 we have considered the case of the tree network with search paths based on DFT. In this chapter, we further study it in the Mesh networks focusing on Hamiltonian search path (each node is visited only once), when possible.

8.1 Terminology and Definitions

8.1.1 Introduction to Graph Layout

Graph layout problems [32] are a particular class of combinatorial optimization problems whose goal is to find a linear layout of an input graph in such way that a certain objective cost is optimized. A linear layout is a labelling of the vertices of a graph with distinct

integers.

A *layout* of an undirected graph $G = (V, E)$ with $n = |V|$ vertices is a bijective function $\varphi : V \rightarrow [n] = \{1, \ldots, n\}$. A common way to represent a layout φ of a graph G is to align its vertices on a horizontal line, mapping each vertex u to position $\varphi(u)$, as shown in Figure 8.1. We denote by $\Phi(G)$ the set of all layouts of a graph G.

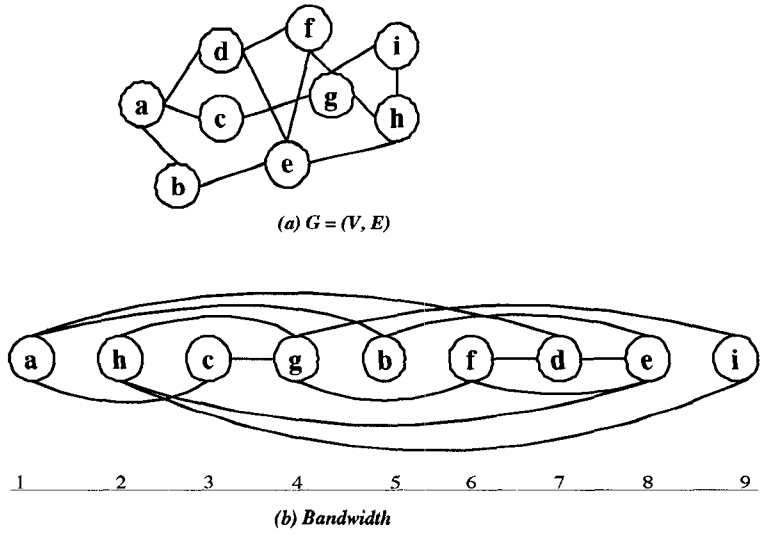

Figure 8.1: Graphical Representation of Layout

Given a layout φ of G and an edge $(u, v) \in E$, we call *length* of (u, v) is $length((u, v), \varphi, G) = |\varphi(u) - \varphi(v)|$, or the distance between its endpoint images.

The *Bandwidth* of a graph $G = (V, E)$ is the layout that minimizes the maximum length of the edges. That is, it is a layout $\varphi^* \in \Phi(G)$ such that $BW(\varphi^*, G) = MIN\{BW(G)\}$, where

$$BW(\varphi^*, G) = \max_{(u,v)\in E} length((u, v), \varphi, G)$$

It is clear that there is a relationship between the search path we are trying to construct to minimize the longest chord (MinMax traversal) and the bandwidth of a graph. In fact,

our traversal of $G = (V, E)$ creates a layout for the corresponding $G' = (V', E')$, but in this case the linear arrangement of nodes is connected; that is: if $\varphi(v) = \varphi(u) + 1$, then $(u, v) \in E'$.

8.2 Search Traversals in the Mesh

Let us restrict for the moment to search paths that are also Hamiltonian paths. In this case we have:

Lemma 8.2.1. *A lower bound on the bandwidth of G is also the lower bound on the MinMax chord of G.*

Proof. By definition of bandwidth and MinMax chord, the traversal corresponding to the MinMax chord is a special kind of bandwidth. The only difference between them is that, in the nodes layout of the bandwidth, consecutive nodes do not necessarily have to be connected. Thus, the lower bound on bandwidth is also the the lower bound on MinMax Traversal. □

Note the that reverse is not true. Note also that the traversal path that contains the MinMax chord may not be an Hamiltonian path in general.

We now show a couple of examples of different searching walks in the mesh (see Figure 8.2). Let the number of nodes in the square mesh be n. The width (the number of edges) of the mesh is $\sqrt{n} - 1$. The length of the maximum chord in the first traversal (*Snake*) a is $2(\sqrt{n} - 1) + 1 = 2\sqrt{n} - 1$; The length of the maximum chord in the second (*Concentric*) b is $4(\sqrt{n} - 1) - 1 = 4\sqrt{n} - 5$. Obviously, the walk in subfigure a is better than walk in sub figure b in terms of the maximum delay time.

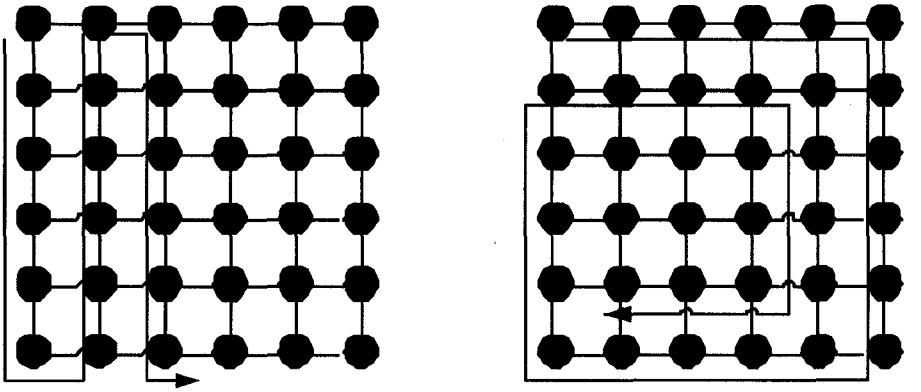

Figure 8.2: Traversal on Mesh. a) *Snake*, b) *Concentric.*

When restricting the searching walks to Hamiltonian paths, we have:

Theorem 8.2.1. *Consider a $m_1 \times m_2$ mesh. The traversal constructed as in Figure 8.2 a) is the optimal traversal (in order of magnitude).*

Proof. It is easy to see that the MinMax chord (and thus the required delay for the moving agent) is $2 \times min\{m_1, m_2\} + 1$. The optimal bandwidth in the Mesh is $min\{m_1, m_2\}$ [32], and by Theorem 8.2.1 it follows that the MinMax chord has length $\Omega(min\{m_1, m_2\})$. □

8.3 Simulations for the Square mesh

.

We now apply the heuristics defined in Chapter 4 in the case of the mesh, to see how much they differ to the

- A1: *random + DFT* .

- A2: *improved priority queue + DFT* .

- A3: *improved priority queue + Greedy.*

- A4: *random + Greedy.*

- A5: *improved priority queue+ BFT.*

- A6: *random + BFT.*

- A7: *improved priority queue + Hybrid.*

The searching agent starts from the corner node in the Mesh. We can obtain similar or a little bit worse results if the agent starts from other node.

Actually, we obtain the same results for all the algorithms applying *priority queue* (two-hop checking) strategy to choose an unvisited neighboring node (i.e.,A2=A3=A5), since such traversal walks results in hamiltonian paths and nodes always have unvisited neighbouring nodes. Thus, we only record the results of one of those algorithms. The results are close to the optimum and the traversal walk is a hamiltonian path. If we apply *priority queue* one-hop checking, the results are a little bit worse and the traversal walk is or close to a hamiltonian path.

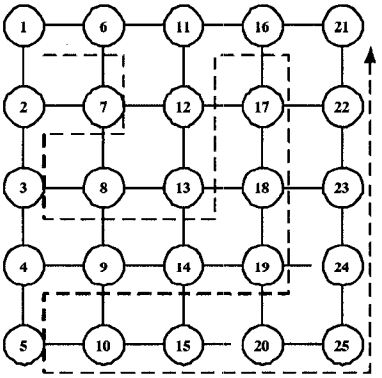

Figure 8.3: Traversal Walk with Priority Queue in Mesh

Figure 8.3 shows the traversal walks with *priority queue* in the square mesh. Suppose the number of nodes in the mesh is n and the width of the mesh is $\sqrt{n}-1$, the length of

the maximum chord in traversal walk with *priority queue* is $2(\sqrt{n}-1)+2(\sqrt{n}-2)+1 = 4\sqrt{n}-5$.

In the mesh, if all the edges have labels, the agent can move along as Figure 8.2 *a* to accomplish the optimum solution; if there is no label, the length of the maximum chord in traversal walk with *priority queue* can be achieved.

width $= \sqrt{n}-1$	n	A1		A2		A4		A6	
		walk	max	walk	max	walk	max	walk	max
3	16	18	15	15	11	17	17	19	17
4	25	35	31	24	15	33	32	36	23
5	36	54	53	35	19	39	25	48	34
6	49	76	71	48	23	54	41	65	52
7	64	115	113	63	27	73	58	95	77
8	81	135	125	80	31	93	85	115	94
9	100	178	165	99	35	121	81	134	97
10	121	203	181	120	39	138	130	186	169
11	144	243	233	143	43	176	152	242	201
12	169	298	291	168	47	218	203	260	129
13	196	377	375	195	51	257	236	356	303
14	225	432	425	224	55	278	175	364	284
15	256	442	351	255	59	323	308	463	415
16	289	544	519	288	63	370	345	494	330
17	324	590	577	323	67	380	265	648	503
18	361	705	705	360	71	449	390	770	631
19	400	766	755	399	75	518	485	816	532
20	441	850	833	440	79	564	507	866	698
21	484	911	897	483	83	585	543	1075	829

Table 8.1: Locating performance in Mesh, continue

width $= \sqrt{n} - 1$	n	A1		A2		A4		A6	
		walk	max	walk	max	walk	max	walk	max
22	529	1007	901	528	87	674	597	1104	928
23	576	1109	1077	575	91	736	677	1244	1057
24	625	1119	1163	624	95	801	732	1365	1268
25	676	1297	261	675	99	840	753	1384	1203
26	729	1439	1437	728	103	949	949	1581	1225
27	784	1554	1549	783	107	982	863	1925	1426
28	841	1611	1549	840	111	1154	956	2157	1642
29	900	1763	1763	899	115	1154	956	2157	1642
30	961	1847	1777	960	119	1195	1063	2273	1714
31	1024	1934	1893	1023	123	1288	1161	2376	1636
32	1089	2131	2109	1088	127	1357	1286	2546	2135
33	1156	2297	2295	1155	131	1436	1392	2624	1864

Table 8.2: Locating performance in Mesh

Chapter 9

Conclusion and Future Work

In any mobile agent system, the ability to communicate with agents in real-time is essential for retrieving any data or information that they have collected, and for supporting coordination and cooperation among them. Communication with a mobile agent subsumes the problem of locating mobile agent. The classical location techniques are based on centralized solutions and on forwarding pointers. Both approaches require a certain host or a registration server to initiate a lookup. In this thesis, we presented several distributed mechanisms to locate mobile agents.

In chapter 3, we presented a fully distributed mechanism based on distributed hashing to locate mobile agents in a system. However, that scheme has two disadvantages: It only supports mobile agents with "Flat" characteristics; the storage load may not be balanced. In the future, we plan to do some research to solve the problems. For example, a new scheme based on a distributed binary tree for locating mobile agents, such that the search name space can be a hierarchical structure, and the load for each coordinator is balanced. One possible idea is that, the name space is organized as a binary tree and the tree

nodes are linearized by pre-order traversal and then partition them into segments. The partitions are assigned to coordinators. Then, each node will manage a range of key name space, and that name space is discrete and has lexicographic order. To achieve the load balance, When a node joins the system, it will check the load information of the nodes it knows from local view. Through repeated checking, a node with heavy load is found and its load is split and assigned to the new node. Furthermore, each node will periodically check its load, the heavy load will be transferred to its direct neighbors. Thus, with the evolution of the system, it will approach to a balanced states.

In chapter 4, a traversal based search strategy is presented to locate the mobile agent. The idea is based on appropriate delays that the mobile agents must perform while moving on the network so to facilitate its tracking, should it be needed. A problem called $MinMax$ chord is initiated for the maximum delay, and it is conjectured as a NP-complete problem. The problem seems very similar to the bandwidth minimization problem, but there are some intrinsic differences. On one hand, that problem does not require that successive vertices be adjacent (more relaxed) and on the other hand, it does not allow vertex repetitions (less relaxed). An interesting problem to further investigate is to determine whether or not the MinMax chord problem is NP-complete.

In chapter 5, we extend the model from the synchronous network to the asynchronous network. The new model assumes that the agents move across an edge with a constant time in normal conditions, and there may exist some delays within a certain value of time depends on the network traffic. In this case we are not guaranteed to locate the moving agent within one traversal; we run simulations to see what is the success rate and the average waiting time when the delay is randomly selected.

In chapter 6, we consider a bounded transmission model, where the time it takes to

traverse a link is unpredictable but bounded. In this case the moving agent is guaranteed to be located and we give the location algorithm.

The results on the mesh topology of Chapter 8 provide only preliminary observations and can be the basis for further research. For example, it is open to determine whether the best traversal indicated for the mesh is indeed optimal (in our results it is optimal only in order of magnitude). Also, our theoretical studies were concentrated on traversals based on hamiltonian paths, traversals non necessarily based on Hamiltonian paths are still to be investigated.

Finally, to decrease the delay time and the locating time, two strategies could be adopted: prolonging the expire time of the forward pointer, or deploying more searching agents to look for the moving agent. Even one extra searching agent can reduce the delay time significantly. If more searching agents are deployed to search the moving agent, some searching agents can guard at the node that connect the searched and unsearched areas. This strategy may initiate some new problems and generate interesting questions, such as *given a certain number of searching agents, what is the new maximum delay time ?*, or *given a fixed maximum delay time, what is the minimum number of searching agents required?* We leave these problems for future research.

Bibliography

[1] M. Adler, H. Racke, N. Sivadasan, C. Sohler, and B. Vocking, "Randomized Pursuit-Evasion in Graphs," in *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2002.

[2] Aglet, http://www.trl.ibm.com/aglets

[3] S. Albers and M. R. Henzinger, "Exploring unknown environments," *SIAM Journal on Computing*, 29:1164-1188, 2000.

[4] S. Alouf, F. Huet, P. Nain, "Forwarders vs. centralized server: An evaluation of two approaches for locating mobile agents," *Performance Evaluation*, Vol. 49(1-4): 299-319, 2002.

[5] S. Alpern, "Rendezvous search: A personal perspective", *Operation Research*, 506(5): 772-795, 1976.

[6] S. Alpern and S. Gal, "The Theory of Search Games and Rendezvous", Kluwer, 2003.

[7] ARA, http://wwwagss.informatik.uni-kl.de/Projekte/Ara/index_e.html

[8] M. Asaka, S. Okazawa, A. Taguchi, and S. Goto, "A method of tracing intruders by use of mobile agents," In *Proc. 9th annual conference of the Internet Society (INET'99)*, 1999.

[9] Yariv Aridor, Mitsuru Oshima, "Infrastructure for Mobile Agents: Requirements and Design", *Mobile Agents Second International Workshop, MA'98*, LNCS 1477, pages: 38-49, 1998.

[10] B. Awerbuch, M. Betke, and M. Singh, "Piecemeal graph learning by a mobile robot," *Information and Computation*, 152:155-172, 1999.

[11] Joachim Baumann, "Mobile Agents: Control Algorithms", LNCS 1658, 2000.

[12] L. Barrière, P. Flocchini, P. Fraigniaud, N. Santoro, "Capture of an Intruder by Mobile Agents," in *Proc. of 14th Symposium on Parallel Algorithms and Architectures(SPAA)*, pages 200-209, 2002.

[13] L. Barrière, P. Flocchini, P. Fraigniaud, and N.Santoro, "Election and rendezvous in fully anonymous systems with sense of direction," In *Proc. 10th Coll. on Structural Information and Communication complexity (SIROCCO'03)*, pages 17-32, 2003.

[14] L. Barrière, P. Flocchini, P. Fraigniaud, and N.Santoro, "Can we elect if we cannot compare," In *Proc. 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA'03)*, pages 200-209, 2003.

[15] L. Barrière, P. Fraigniaud, N. Santoro, D. M. Thilikos, "Searching is not Jumping," in *Proc. of 29th Workshop on Graph Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science Vol. 2880, pp.34-45, 2003.

[16] V. Baston and S. Gal, "Rendezvous search when marks are left at the starting points," *Naval Research Logistics*, 48(8):722-731, 2001.

[17] M. Bender, A. Fernandez, D. Ron, A. Sahai, and S. Vadhan, "The power of a pebble: Exploring and mapping directed graphs," In *Proc. 30th ACM Symp. on Theory of Computing (STOC'98)*, pages 269-287, 1998.

[18] M. Bender and D. K. Slonim, "The power of team exploration: two robots can learn unlabeled directed graphs," In *Proc. 35th Symp. on Foundations of Computer Science (FOCS'94)*, pages 75-85, 1994.

[19] L. Bernardo, P. Pinto, "A scalable location service with fast update responses," *In IEEE Global Telecommunications Conference (GLOBECOM'98)*, v. 5, pp:2876-2881, 1998.

[20] D. Bienstock and P. D. Seymour, "Monotonicity in graph searching," *J. Algorithms*, 12(2):239-245, 1991.

[21] M. Blum and D. Kozen, "On the power of the compass (or, why mazes are easier to search than graphs)," In *19th Symposium on Foundations of Computer Science (FOCS'78)*, pages 132-142, 1978.

[22] R. Breisch, "An Intuitive Approach to Speleotopology," *Southwestern Cavers* VI(5), pp.72-78, 1967.

[23] G. Cabri, L. Leonardi, F. Zambonelli, "Mobile-Agent Coordination Models for Internet Applications," *Computer*, pp.82-89, 2000.

[24] J. Chalopin, "Election in the qualitative world," In *Proc. of 13th International Colloquium on Structural Information and Communication Complexity (SIROCCO'06)* 2006.

[25] W. E. Chen, C. R. Leng, "A Novel Mobile Agent Search Algorithm", *First International Workshop on Mobile Agents 97 (MA'97)*, LNCS 1219, pages:162-173, 1997.

[26] D'Agents, http://agent.cs.dartmouth.edu/.

[27] S. Das, P. Flocchini, S. Kutten, A. Nayak, N. Santoro, "Map Construction of Unknown Graphs by Multiple Agents," *Theoretical Computer Science*, 385(1-3): 34-48, 2007.

[28] M. Demirbas, A. Arora, M. G. Gouda, "A pursuer-evader game for sensor networks," In *Proceedings of the Sixth Symposium on Self-Stabilizing Systems*, LNCS 2704, pp.1-16, 2003.

[29] X. Deng, T. Kameda, and C. H. Papadimitriou, "How to learn an unknown environment: The rectilinear case," *Journal of the ACM*, 45:215-245, 1998.

[30] X. Deng and C. H. Papadimitriou, "Exploring an unknown graph," *Journal of Graph Theory*, 32(3):265-297, 1999.

[31] A. Dessmark, P. Fraigniaud, and A. Pelc, "Deterministic rendezvous in graphs," In *Proc. 11th European Symposium on Algorithms (ESA'03)*, pages 184-195, 2003.

[32] J. Diaz, B. J. Petit, M. Serna, "A survey of graph layout problems," In *ACM Computing Surveys (CSUR)*, Volume 34, Issue 3, pp.313-356, 2002.

[33] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc, "Tree exploration with little memory," *Journal of Algorithms*, 51:38-63, 2004.

[34] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro, "Multiple agents rendezvous in a ring in spite of a black hole," In *Proc. of 7th International Conference on Principles of Distributed Systems (OPODIS'03)*, 2003.

[35] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes, "Robotic exploration as graph construction," *Transactions on Robotics and Automation*, 7(6):859-865, 1991.

[36] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk, "Multiple Mobile Agent Rendezvous in the Ring," *Proc. LATIN 2004*, LNCS 2976, pp.599-608, 2004.

[37] P. Flocchini and M. Xie, "A Fully Distributed Mechanism for Locating Mobile Agents," in *2nd International IEEE Multi-Conference on Computing in the Global Information Technology (ICCGI 2007)*, pp. 41-51, 2007.

[38] P. Flocchini, A. Roncato, and N. Santoro, "Computing on anonymous networks with sense of direction," *Theor. Comput. Sci.*, 1-3(301):355-379, 2003.

[39] P. Flocchini and M. Xie, "A Decentralized Solution for Locating Mobile Agents," in *Proc. Euro-Par 2007*, LNCS 4641, pp. 618-628, 2007.

[40] R. J. Fowler, "Decentralized object finding using forwarding addresses," Ph.D. Thesis, Technical Report 85-12-1, Department of Computer Science, University of Washington, USA, 1985.

[41] P. Fraigniaud, L. Gasieniec, D. Kowalski, and A. Pelc, "Collective tree exploration," *In 6th Latin American Theoretical Informatics Symp. (LATIN'04)*, pages 141-151, 2004.

[42] P. Fraigniaud and D. Ilcinkas, "Digraph exploration with little memory," In *21st Symp. on Theoretical Aspects of Computer Science (STACS'04)*, pages 246-257, 2004.

[43] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg, "Graph exploration by a finite automaton," In *29th Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 451-462, 2004.

[44] P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil, "Space lower bounds for graph exploration via reduced automata," In *Proc. of the 12th Int. Colloquium on Structural Information and Communication Complexity(SIROCCO'05)*, pages 140-154, 2005.

[45] L. Gasieniec, E. Kranakis, D. Krizanc, and X. Zhang, "Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring," In *Proc. of 32nd Conference on Current Trends in Theory and Practice of Computer Science(SOFSEM'06)*, pages 282-292, 2006.

[46] M. R. Garey, D. S. Johnson, "Computers and Intractability : A Guide to the Theory of NP-Completeness," ISBN 0716710447, 1979.

[47] H. C. Hsiao, M. Baker, C. T. King, "A Peer-to-Peer Mechanism for Resource Location and Allocation Over the Grid. ," *Proc. of the Second International Symposium on Parallel and Distributed Processing and Applications (ISPA 2004)*, LNCS 3358, pp. 604-614, 2004.

[48] Jung, "Java Universal Network/Graph Framework,", http://jung.sourceforge.net/.

[49] Georgia Kastidou, Evaggelia Pitoura, George Samaras, "A Scalable Hash-Based Mobile Agent Location Mechanism", *23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, 2003.

[50] Georgia Kastidou, Evaggelia Pitoura, George Samaras, "A Scalable Mobile Agent Location Mechanism", *Proceedings of the Second International Workshop on Mobile Agents*, LNCS, pages: 38-49, 1998.

[51] E. Korach, D. Rotem and N. Santoro, "Distributed Algorithms for Finding Centers and Medians in Networks," *ACM Trans. on Programming Languages and Systems*, 6(3):380-401, 1984.

[52] D. R. Kowalski and A. Pelc, "Polynomial deterministic rendezvous in arbitrary graphs," In *Proc. of 15th Int. Symposium on Algorithms and Computation (ISAAC'04)*, pages 644-656, 2004.

[53] D. Kozen, "Automata and planar graphs," In *Fundations Computatial Theory (FCT'79)*, pages 243-254, 1979.

[54] E. Kranakis, D. Krizanc, and E. Markou, "Mobile agent rendezvous in a synchronous torus," In *Proc. of 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 653-664, 2006.

[55] E. Kranakis, D. Krizanc, S. Rajsbaum. "Mobile agent rendezvous: A Survey". *Proc. 13th Int. Coll. on Structural Information and Communication Complexity (SIROCCO)*, 1-9, 2006.

[56] E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk, "Mobile agent rendezvous in a ring," In *Int. Conf. on Distibuted Computing Systems (ICDCS 03)*, pages 592-599, 2003.

[57] A. S. LaPaugh, "Recontamination does not help to search a graph," *Journal of the ACM*, 40(2):224-245, 1993.

[58] T. Li, K. Lam, "An optimal location update and searching algorithm for tracking mobile agent," *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages:517-526, 2002.

[59] T. Li, Zhang Junfang. "An optimal and secure tracking scheme for mobileagents," *Engineering of Intelligent Systems (EIS2002)*, 2002.

[60] Y. Lien, C. R. Leng, "On the search of mobile agents," *Proc. of The 7th IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC'96)*, pp. 703-707, 1996.

[61] G. De Marco, L. Gargano, E. Kranakis, D. Krizanc, A. Pelc, and U. Vaccaro, "Asynchronous deterministic rendezvous in graphs," *Theoretical Computer Science*, 355(3):315-326, 2006.

[62] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson and C. H. Papadimitriou, "The complexity of searching a graph," *Journal of the ACM*, 35(1):18-44, 1988.

[63] P. Mell and M. McLarnon, "Mobile Agent Attack Resistant Distributed Hierarchical Intrusion Detection Systems", *International Symposium on Recent Advances in Intrusion Detection (RAID'99)*, 1999.

[64] D. Milojicic et al, "MASIF: The OMG Mobile Agent System Interoperability Facility", *Mobile Agents Second International Workshop, MA'98*, LNCS 1477, pages: 50-67, 1998.

[65] MASIF,http://www.fokus.gmd.de/research/cc/ecco/masif/.

[66] D. Milojicic et al, "Mobile objects and agents", *In Processing of the USENIX Conference on Object-Oriented Technologies and Systems*, 1998.

[67] H. Muller, "Automata catching labyrinths with at most three components." *Elektronische Informationsverarbeitung und Kybernetic*, 15(1):3-9, 1979.

[68] M. Naor, U. Wieder, "Know Thy Neighbor's Neighbor: Better Routing for Skip-Graphs and Small Worlds," *Proc. of the Third Int. Workshop on Peer-to-Peer Systems (IPTPS '04)*, LNCS 3279, pp.269-277, 2005.

[69] Magic Odyssey, http://www.genmagic.com/technology/odyssey.html.

[70] B. C. Ooi, W. S. Ng, K. Tan, and A. Y. Zhou, "Information Acquisition Through an Integrated Paradigm: Agent + Peer-to-peer," *Proceedings of the Second International Workshop on Agent and Peer-to-peer Computing, AP2PC 2003*, Lecture Notes in Computer Science Vol.2872, pp.1-12, 2003.

[71] P. Panaite and A. Pelc, "Exploring unknown undirected graphs," *Journal of Algorithms*, 33:281-295, 1999.

[72] T. D. Parsons, "Pursuit-Evasion in a Graph," *Theory and Applications of Graphs*, Lecture Notes in Mathematics, Volume 642, pages 426-441, 1976.

[73] T. D. Parsons, "The Search Number of a Connected Graph," in *Proc. of 9th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pages 549-554, 1978.

[74] M. O. Rabin, "Maze threading automata," *Technical Report Seminar Talk*, University of California at Berkeley, 1967.

[75] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A Scalable Content-Addressable Network," *Pro. of ACM SIGCOMM*, 161-172, 2001.

[76] V. Roth and J. Peters, "A Scalable and Secure Global Tracking Service for Mobile Agent", *Proc. 5th Int. Conference (MA2001)*, LNCS 2240, pages:169-181, 2001.

[77] O. D. Sahin, F. Emekci, D. Agrawal, and A. El Abbadi, "Content-Based Similarity Search over Peer-to-Peer Systems," *Proceedings of Second International Workshop on Databse, Information Systems, and Peer-to-Peer Computing (DBISP2P 2004)*, Lecture Notes in Computer Science Vol.3367, pp.61-78, 2004.

[78] N. Santoro, "Design and Analysis of Distributed Algorithms," Wiley, *ISBN 0-471-71997-8*, pp.71-89, 2007.

[79] CL. E. Shannon, "Presentation of a maze-solving machine," In *8th Conf. of the Josiah Macy Jr. Found. (Cybernetics)*, pages 173-180, 1951.

[80] O. Shehory, "A Scalable Agent Location Mechanism", *Lecture Notes in Artificial Intelligence, Intelligent Agents VI*, 1999.

[81] A. Di Stefano, C. Santoro, "Locating Mobile Agents in a Wide Distributed Environment", *IEEE Transactions on Parallel and Distributed Systems*, Vol.13 No.8, pages: 844-864, 2002.

[82] A. Di Stefano, L. Lo Bello, C. Santoro, "Naming and locating mobile agents in an Internet environment", Proc. of Third International Enterprise Distributed Object Computing Conference, EDOC'99, pages:153 -161, 1999.

[83] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," *ACM SIGCOMM*, pp.149-160, 2001.

[84] TACOMA, http://www.cs.uit.no/forskning/DOS/Tacoma/

[85] Voyager, http://objectspace.com/voyager.

[86] X. Yu and M. Yung, "Agent rendezvous: A dynamic symmetry-breaking problem," In *Int. Coll. on Automata Languages and Programming (ICALP'96)*, pages 610-621, 1996.

Appendix A: Tables for Chapter 4 - Random Graphs

The following tables describe the results of the various combination of traversal strategies described in Chapter 4 in the case of random graphs for variable values of n.

m	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3200	1364	299	691	1432		877	120	461	933
4000	1302	264	630	1363		804	120	450	895
4800	1166	176	562	1196		776	122	436	875
5600	1109	160	538	1137		769	125	411	861
6400	1079	154	540	1125		760	125	412	852
7200	1028	136	504	1063		763	126	408	844
8000	1015	135	506	1048		752	127	405	841

Table 9.1: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 800$.

m	A_4: Random + Greedy				A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
3200	890	137	461	921	992	149	531	1080
4000	879	128	443	894	906	137	489	992
4800	853	137	426	872	846	134	463	942
5600	842	137	421	861	814	132	452	907
6400	839	136	422	854	790	131	433	888
7200	833	136	416	845	790	130	431	875
8000	828	135	409	840	783	131	428	868

Table 9.2: A_4, A_5maximum/average delay, location time, length of the walk. $n = 800$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
3200	1396	309	654	1464	869	119	461	926
4000	1284	248	628	1355	822	121	443	894
4800	1146	190	562	1243	786	122	430	874
5600	1055	153	553	1159	773	123	423	861
6400	1036	142	512	1105	774	125	425	854
7200	955	129	497	1042	760	125	411	843
8000	955	127	480	1037	749	127	412	838

Table 9.3: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 800$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3200	1023	160	525	1054
4000	960	155	489	986
4800	909	147	460	932
5600	880	145	448	903
6400	863	143	441	889
7200	860	142	436	877
8000	845	139	428	864

Table 9.4: A_6: maximum/average delay, location time, length of the walk. $n = 800$.

m	A_1: Random + DFT				A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	854	187	442	896	522	74	291	582
2500	749	128	396	790	497	75	276	561
3000	682	93	345	703	475	77	272	546
3500	687	98	341	712	474	77	263	541
4000	662	90	339	685	472	78	252	532
4500	618	81	318	642	480	80	260	530
5000	638	84	321	654	472	80	249	526

Table 9.5: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 500$.

m	A_4: Random + Greedy				A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	553	85	279	574	591	88	320	657
2500	540	86	278	554	543	84	312	610
3000	528	85	265	543	500	83	287	581
3500	526	86	261	538	495	82	281	565
4000	519	85	260	531	483	81	265	549
4500	516	85	258	527	486	82	269	544
5000	516	85	261	526	482	81	256	535

Table 9.6: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 500$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	799	154	421	859	532	75	289	586
2500	755	138	408	818	505	75	272	556
3000	642	91	351	712	481	76	271	545
3500	670	106	350	739	482	77	276	539
4000	592	79	322	656	467	78	264	531
4500	571	76	311	627	470	79	255	528
5000	587	78	312	647	471	79	248	525

Table 9.7: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 500$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
2000	624	99	312	650
2500	578	95	297	606
3000	554	90	286	572
3500	547	90	278	563
4000	532	87	261	546
4500	529	87	266	540
5000	527	87	257	539

Table 9.8: A_6: maximum/average delay, location time, length of the walk. $n = 500$.

m	A_1: Random + DFT				A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
800	317	63	261	337	203	31	111	232
1000	285	48	248	301	191	31	112	223
1200	270	41	242	285	184	30	108	217
1400	261	37	231	271	182	31	102	214
1600	236	32	220	248	182	31	103	211
1800	235	31	218	242	185	31	102	211
2000	237	31	218	246	184	32	99	209

Table 9.9: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 200$.

m	A_4: Random + Greedy				A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
800	217	34	112	228	223	34	124	253
1000	211	34	110	222	200	32	111	234
1200	208	34	106	216	195	32	107	225
1400	209	35	105	217	187	32	108	222
1600	205	34	104	213	184	32	100	214
1800	203	33	101	210	185	32	101	214
2000	203	34	101	209	185	32	103	211

Table 9.10: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 200$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
800	303	56	160	331	203	30	112	229
1000	260	40	142	296	190	30	112	221
1200	254	41	140	286	187	30	104	216
1400	256	37	138	281	191	31	102	215
1600	226	31	136	281	185	31	101	211
1800	222	32	122	253	185	31	100	210
2000	221	33	121	247	185	31	99	209

Table 9.11: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 200$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
800	234	38	121	247
1000	222	37	116	234
1200	212	36	111	223
1400	214	36	107	222
1600	207	35	106	216
1800	204	34	101	212
2000	204	34	101	211

Table 9.12: A_6: maximum/average delay, location time, length of the walk. $n = 200$.

Appendix B: Tables for Chapter 4 - Random Regular Graphs

The following tables describe the results of the various combination of traversal strategies described in Chapter 4 in the case of random regular graphs. In this case the size of the graph varies as well as its degree.

d	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	1599	402	817	1719		1031	65	539	1120
4	1594	391	803	1706		958	80	511	1090
5	1508	275	766	1612		921	95	507	1083
6	1456	201	724	1524		908	108	509	1078
7	1322	187	643	1381		883	119	499	1073
8	1304	165	602	1345		879	128	510	1072
9	1298	158	599	1310		898	133	501	1063

Table 9.13: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 1000$

m	A_4: Random + Greedy					A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	1223	137	602	1272		1184	107	601	1303
4	1160	154	558	1193		1092	108	589	1212
5	1135	162	534	1159		1034	114	556	1177
6	1110	165	535	1135		1036	126	508	1171
7	1091	168	513	1118		987	132	509	1149
8	1087	170	507	1109		993	141	510	1147
9	1078	170	489	1096		979	144	502	1130

Table 9.14: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 1000$.

d	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	1209	316	688	1418		1026	139	516	1112
4	1182	298	687	1465		970	140	509	1090
5	1156	241	674	1397		931	145	504	1085
6	1145	195	614	1358		912	138	511	1080
7	1178	165	603	1387		914	144	506	1077
8	1154	159	611	1340		896	142	501	1075
9	1138	160	605	1321		895	148	507	1067

Table 9.15: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 1000$.

d	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3	1548	223	789	1635
4	1426	219	687	1472
5	1334	212	667	1378
6	1278	204	635	1312
7	1213	196	587	1250
8	1186	191	578	1211
9	1160	188	545	1188

Table 9.16: A_6: maximum/average delay, location time, length of the walk. $n = 1000$.

d	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	1347	289	697	1423		791	50	421	887
4	1302	243	669	1376		779	63	425	870
5	1287	218	627	1312		719	75	416	868
6	1268	186	588	1289		712	87	417	859
7	1212	152	567	1267		725	97	411	860
8	1167	134	534	1213		721	96	412	861
9	1124	131	504	1178		713	95	410	858

Table 9.17: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 800$.

d	A_4: Random + Greedy					A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	976	112	487	1017		944	80	495	1067
4	933	123	452	956		854	84	432	984
5	900	128	448	929		801	91	409	959
6	884	132	435	909		824	102	411	824
7	872	132	431	891		821	109	399	821
8	873	131	427	899		819	108	403	823
9	869	134	422	892		817	108	406	819

Table 9.18: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 800$.

d	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	980	302	556	1129		809	118	423	885
4	902	267	487	1091		765	120	418	872
5	945	206	511	1148		728	121	413	868
6	935	188	502	1121		719	122	426	862
7	931	168	510	1110		716	125	415	860
8	940	143	502	1101		724	124	419	856
9	956	126	499	1102		713	126	407	855

Table 9.19: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 800$.

d	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3	1377	207	698	1445
4	1200	187	579	1251
5	1095	172	563	1138
6	1040	165	487	1070
7	987	156	443	1011
8	988	155	432	1009
9	985	151	444	1005

Table 9.20: A_6: maximum/average delay, location time, length of the walk. $n = 800$.

d	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	834	180	423	869		473	30	268	557
4	814	130	412	845		461	40	259	547
5	798	99	397	813		458	48	251	543
6	739	89	346	768		441	54	252	540
7	734	97	314	756		429	60	247	538
8	702	87	325	728		438	63	248	534
9	669	85	332	699		445	67	246	532

Table 9.21: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 500$.

d	A_4: Random + Greedy					A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	602	69	311	631		548	46	319	647
4	573	77	292	592		504	51	301	609
5	558	80	281	579		516	56	285	596
6	549	81	271	567		476	60	286	586
7	542	83	267	560		484	67	279	580
8	539	83	269	550		471	68	268	568
9	530	84	268	548		468	70	266	559

Table 9.22: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 500$.

d	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	655	151	376	771		510	75	278	561
4	520	132	321	671		451	77	269	543
5	538	101	314	684		442	75	265	540
6	554	102	324	695		435	76	253	539
7	555	93	321	673		453	77	255	537
8	578	81	318	684		440	78	234	534
9	572	76	320	675		435	78	228	531

Table 9.23: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 500$.

d	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3	811	121	420	865
4	715	110	362	757
5	672	104	332	699
6	618	98	311	653
7	602	97	312	627
8	580	93	287	602
9	572	92	286	593

Table 9.24: A_6: maximum/average delay, location time, length of the walk. $n = 500$.

d	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	468	65	241	500		291	18	163	334
4	387	48	236	496		269	23	161	324
5	376	43	229	470		257	28	158	323
6	384	41	211	438		261	32	153	324
7	382	38	208	445		253	35	153	322
8	380	32	210	435		256	37	150	319
9	378	33	198	406		259	40	151	320

Table 9.25: A_1, A_7: maximum/average delay, location time, length of the walk. $n = 300$.

d	A_4: Random + Greedy					A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	356	43	191	385		319	25	182	374
4	340	46	172	357		295	28	174	356
5	339	48	169	347		309	32	164	347
6	325	49	162	341		282	36	161	342
7	321	50	158	335		272	38	159	342
8	318	50	159	332		270	39	160	334
9	318	51	147	330		269	41	151	333

Table 9.26: A_4, A_5: maximum/average delay, location time, length of the walk. $n = 300$.

d	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3	374	58	212	442		291	36	163	334
4	305	51	187	398		267	38	157	326
5	307	50	189	387		265	33	158	324
6	304	48	178	395		252	35	155	323
7	321	46	180	393		256	34	152	325
8	329	43	181	395		253	32	151	320
9	323	41	178	384		262	34	151	319

Table 9.27: A_2, A_3: maximum/average delay, location time, length of the walk. $n = 300$.

d	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3	474	69	251	511
4	407	62	212	431
5	384	60	198	405
6	358	56	187	381
7	347	55	176	366
8	337	54	175	354
9	332	53	174	346

Table 9.28: A_6: maximum/average delay, location time, length of the walk. $n = 300$.

Appendix C:Tables for Chapter 5

The following tables describes maximum delay, average delay, location time, length of the walk, and success rate for variable values of n in the case of Asynchronous networks.

m	A_1: Random + DFT				A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
3200	1365	300	690	1430	878	121	463	943
4000	1300	265	630	1365	805	120	451	889
4800	1167	175	562	1193	770	122	433	876
5600	1106	161	536	1135	768	126	415	860
6400	1080	152	542	1126	762	124	417	853
7200	1030	137	501	1067	762	126	409	845
8000	1016	136	505	1043	754	128	406	842

Table 9.29: Results for A_1 and A_7 with $n = 800$.

m	A_4: Random + Greedy				A_5: Improved+BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
3200	894	139	462	922	992	148	532	1079
4000	881	128	443	895	908	137	489	993
4800	855	135	426	872	847	135	465	945
5600	842	137	423	862	816	133	454	909
6400	839	135	422	856	793	132	434	887
7200	829	136	416	845	791	130	432	877
8000	828	134	411	841	782	132	429	868

Table 9.30: Results for A_4 and A_5 with $n = 800$.

m	A_2: Improved Priority + DFT					A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
3200	1396	309	654	1464		869	119	461	926
4000	1284	248	628	1355		822	121	443	894
4800	1146	190	562	1243		786	122	430	874
5600	1055	153	553	1159		773	123	423	861
6400	1036	142	512	1105		774	125	425	854
7200	955	129	497	1042		760	125	411	843
8000	955	127	480	1037		749	127	412	838

Table 9.31: Results for A_2 and A_3 with $n = 800$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
3200	1020	161	496	1055
4000	965	156	488	987
4800	908	147	458	930
5600	881	144	448	904
6400	864	143	442	889
7200	860	141	438	878
8000	847	139	427	865

Table 9.32: Results for A_6 with $n = 800$.

m	A_1: Success rate (%)					
	E= 0.1	0.2	0.5	1	1.5	2
3200	100	97	96	90	85	75
4000	100	100	96	91	86	74
4800	100	99	95	92	88	76
5600	100	98	97	93	87	74
6400	100	99	96	91	85	75
7200	100	98	95	92	86	76
8000	100	100	96	91	84	77

Table 9.33: Success rate for $n = 800$.

m	A_1: Random + DFT				A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	856	186	444	898	524	75	292	584
2500	745	126	396	791	495	75	274	562
3000	682	99	343	731	474	76	260	546
3500	686	96	340	713	473	77	263	543
4000	661	93	338	686	471	76	250	530
4500	618	85	317	643	479	80	263	531
5000	638	81	322	655	480	81	246	528

Table 9.34: Results for A_1 and A_7 with $n = 500$.

m	A_4: Random + Greedy				A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	550	85	270	585	539	87	315	659
2500	541	87	274	552	542	84	312	612
3000	528	85	261	539	512	83	280	580
3500	528	86	262	538	496	82	282	562
4000	516	85	259	532	483	81	260	549
4500	510	86	258	528	482	82	262	542
5000	511	85	260	525	483	81	253	536

Table 9.35: Results for A_4 and A_5 with $n = 500$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
2000	791	154	410	860	530	76	234	588
2500	754	136	400	819	504	76	265	555
3000	644	107	358	718	484	76	268	551
3500	673	98	351	729	480	77	267	542
4000	595	88	328	658	468	78	264	533
4500	577	79	310	629	465	79	254	529
5000	585	78	311	632	468	78	245	522

Table 9.36: Results for A_2 and A_3 with $n = 500$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
2000	620	92	311	652
2500	571	93	292	606
3000	558	89	279	570
3500	544	90	278	569
4000	532	86	262	543
4500	527	87	263	542
5000	526	86	250	533

Table 9.37: Results for A_6 with $n = 500$.

m	A_1: Success rate (%)					
	E= 0.1	0.2	0.5	1	1.5	2
2000	100	99	97	90	83	72
2500	99	98	97	91	84	73
3000	100	98	98	91	85	74
3500	100	100	95	92	87	73
4000	100	98	96	91	85	71
4500	100	100	95	92	84	76
5000	100	99	94	90	83	75

Table 9.38: Success rate for $n = 500$.

m	A_1: Random + DFT					A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length		Max. D.	Av. D.	Loc. T.	Length
400	134	22	72	144		93	14	53	113
500	130	22	71	146		90	15	52	112
600	128	19	62	137		91	14	51	106
700	122	17	60	125		89	15	54	105
800	112	18	61	122		90	16	52	105
900	114	16	58	120		88	15	51	103
1000	109	17	52	117		89	15	50	104

Table 9.39: Results for A_1 and A_7 with $n = 100$.

m	A_4: Random + Greedy				A_5: Improved + BFT			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
400	16	102	53	114	98	15	45	120
500	18	101	52	112	90	15	55	112
600	17	102	51	110	92	16	43	109
700	16	100	53	106	88	15	39	105
800	17	96	50	107	85	16	41	109
900	16	99	49	104	88	15	42	103
1000	17	98	48	102	90	14	380	103

Table 9.40: Results for A_4 and A_5 with $n = 100$.

m	A_2: Improved Priority + DFT				A_3: Improved Priority + Greedy			
	Max. D.	Av. D.	Loc. T.	Length	Max. D.	Av. D.	Loc. T.	Length
400	128	20	65	149	92	16	51	114
500	125	19	67	143	91	15	53	112
600	105	15	61	126	89	15	48	108
700	106	15	59	123	89	16	52	107
800	105	15	59	121	90	16	51	106
900	102	15	56	114	91	16	51	104
1000	101	15	56	112	89	16	50	103

Table 9.41: Results for A_2 and A_3 with $n = 100$.

m	A_6: Random + BFT			
	Max. D.	Av. D.	Loc. T.	Length
400	106	18	54	118
500	104	17	55	112
600	103	18	53	110
700	100	17	51	108
800	100	16	52	106
900	99	17	51	105
1000	98	17	51	104

Table 9.42: Results for A_6 with $n = 100$.

m	A_1: Success rate (%)					
	E= 0.1	0.2	0.5	1	1.5	2
400	100	98	95	90	82	73
500	100	98	96	91	80	71
600	100	99	93	90	83	74
700	100	99	92	89	83	76
800	100	98	94	91	85	74
900	99	100	95	90	82	76
1000	100	99	93	91	84	75

Table 9.43: Success rate for $n = 100$.