Mobile Agent Planning Problems

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

KATSUHIRO MOIZUMI

Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire

October 23, 1998

Examining Committee:

Dr. George Cybenko (chairman)

Dr. Charles E. Hutchinson

Dr. Linda F. Wilson

Dr. Geoffrey M. Davis

Dean of Graduate Studies

For my grandfather Seiichi Moizumi

Abstract

Mobile agents have received much attention recently as a way to efficiently access distributed resources in a low bandwidth network. Planning allows mobile agents to make the best use of the available resources. This thesis studies several planning problems that arise in mobile agent information retrieval and data-mining applications. The general description of the planning problems is as follows: We are given sites at which a certain task might be successfully performed. Each site has an independent probability of success associated with it. Visiting a site and trying the task there requires time (or some other cost matrix) regardless of whether the task is completed successfully or not. Latencies between sites, that is, the travel time between those two sites also have to be taken into account. If the task is successfully completed at a site then the remaining sites need not be visited. The planning problems involve finding the best sequence of sites to be visited, which minimizes the expected time to complete the task. We name the problems *Traveling Agent Problems* due to their analogy with the Traveling Salesman Problem. This Traveling Agent Problem is NP-complete in the general formulation. However, in this thesis a polynomial-time algorithm has been successfully developed to solve the problem by adding a realistic assumption to it. The assumption enforces the fact that the network consists of subnetworks where latencies between machines in the same subnetwork are constant while latencies between machines located in different subnetworks vary. Different versions of the Traveling Agent Problem are considered: (1) single agent problems, (2) multiple agent problems (multiple agents cooperate to complete the same task) and (3) deadline problems (single or multiple agents need to complete a task without violating a deadline constraint at each location in the network). Polynomial and pseudo-polynomial algorithms for these problems have been developed in this thesis. In addition to the theory and algorithm development for the various Traveling Agent Problems, a planning system that uses these algorithms was implemented. Descriptions of the mobile agent planning system with its supporting components such as network

sensing system, directory service system, and clustering system, are also given in this thesis.

Acknowledgments

I would like to thank my advisor, Professor Cybenko, without whom I have not been able to obtain the Ph.D. degree. I truly appreciate his advice and inspiration on my thesis and life as well as his patience and generosity. I also would like to express my thanks to my committee members, Professor Charles Hutchinson, Professor Linda Wilson and Professor Geoffrey Davis, for their guidance and insight. I am also thankful to Professor Robert Gray and Brian Brewlington for their helpful feedback on my research and thesis. For those people who have been working with me in our lab and who have also been my good friends for the last four years, Wilmer Caripe, Yunxin Wu, Aditya Bhasin, Vladimir Ristanovic, Megumi Hasegawa, Daniel Bilar, Kartik Raghavan, Wei Shen, Rahul Ghate, Daniel Burroughs, Shankar Sundaram, Robert Grube, Guofei Jiang, thank you very much for your help. I would like to thank my friends at Dartmouth, Kangbin Chua, Sunil & Kavita Desai, Melanie Blanchard, Professor Carlos Alberola, Manoj George, Damnath DeTissera, Ann Bilderback, Ting Cai and Ming Guo for their warm friendship that has cheered me up and encouraged me to continue my work. Many thanks to my family that has taken good care of me from a great distance. I wouldn't have been able to survive writing this thesis without their support and understanding. Finally, I would like to thank all my friends, teachers and colleagues for everything they did for me.

Contents

Abstract iii

Acknowledgments iv

List of Tables vii

List of Figures ix

1 Introduction 1

2 Problem Statement 5

3 Background 9

3.1 Possible Solution Approaches 9

3.1.1 Artificial Intelligence 9

3.1.2 Dynamic Programming 14

3.2 Comparison of the Different Approaches 26

4 Theory and Algorithm 29

4.1 Traveling Agent Problem 30

4.2 Single Agent 33

4.2.1 Constant Latencies 33

4.2.2 The Multiple Subnetwork Case 38

4.3 Multiple Agents 44

4.3.1 Probability of Success = 0 47

4.3.2 Probability of Success = 1 48

4.3.3 Constant Probability ≥ 0.5 . . . 49
4.3.4 Constant Computation Time . . . 77
4.4 The Traveling Agent Problem with Deadlines . . . 77
4.4.1 Multiple Subnetwork Case . . . 81
4.4.2 Multiple Agents . . . 87

5 Implementation **90**
5.1 Overview of Mobile Agent Planning System . . . 90
5.2 Planning Module . . . 93
5.3 Directory Service Module . . . 95
5.4 Network-Sensing Module . . . 97
5.4.1 Literature of Network Monitoring . . . 97
5.4.2 Architecture of the Network Sensing System . . . 98
5.4.3 Implementation of the Network Sensing System . . . 101
5.5 Clustering Module . . . 105

6 Experimental Results **107**
6.1 Experimental Setup . . . 108
6.1.1 Overview of the Experiments . . . 108
6.1.2 Physical Configuration . . . 108
6.1.3 Software Configuration . . . 110
6.2 Experiments and results . . . 112
6.2.1 A Single Agent without Deadlines . . . 112
6.2.2 Multiple Agents without Deadlines . . . 121
6.2.3 A Single Agent with Deadlines . . . 128

7 Further Work **131**

8 Conclusion **133**

List of Tables

3.1 The transition probability matrix 19

3.2 The immediate cost matrix 20

3.3 The optimal policy for each state 21

3.4 Performance table 26

6.1 Performance comparison : TAP without deadlines 121

6.2 The setting of sub-experiments 122

6.3 Performance comparison : Prob $\simeq 0.9$ 122

6.4 Performance comparison : Prob $\simeq 0.7$ 123

6.5 Performance comparison : Prob $\simeq 0$ 124

6.6 Performance comparison : Constant computation time 125

6.7 Performance comparison : TAP with deadlines 130

List of Figures

2.1 An example of the planning problem for the mobile agent 8

3.1 A solution to the example problem 11

4.1 Definition of sites a, b, c, d, e. 41
4.2 Schedule T for multiple agents 46
4.3 A schedule for the m agent problem 51
4.4 The optimal assignment of the last two sites 55
4.5 The optimal assignment of the last two sites (2) 61
4.6 The optimal assignment of the last two sites (3) 63
4.7 The optimal assignment of the last two sites (4) 66
4.8 The optimal assignment of the last two sites (5) 68
4.9 The optimal assignment of the last two sites (6) 72
4.10 The optimal assignment of the last two sites (7) 76

5.1 The D'Agents' architecture 91
5.2 The architecture of the planning system 92
5.3 Pseudo-code for the planning agent 94
5.4 Pseudo-code for the directory service agent 96
5.5 Pseudo-code of the stationary network sensing agent 101
5.6 Pseudo-code of the network sensing server 102
5.7 Pseudo-code of the network sensing slave 104

6.1 Network setup for the experiments 109
6.2 Performance comparison (Greedy on Prob): Elapsed time 113

6.3 Performance comparison (Greedy on Prob): Execution time 114
6.4 Performance comparison (Greedy on Comp): Elapsed time 115
6.5 Performance comparison (Greedy on Comp): Execution time 116
6.6 Histogram (Greedy on Prob): Elapsed time 117
6.7 Histogram (Greedy on Comp): Elapsed time 118
6.8 Performance comparison (Greedy on Prob): Prob $\simeq 0.9$ 123
6.9 Performance comparison (Greedy on Prob): Prob $\simeq 0.7$ 124
6.10 Performance comparison (Greedy on Prob): Prob $\simeq 0$ 125
6.11 Performance comparison (Greedy on Comp): Prob $\simeq 0$ 126
6.12 Performance comparison (Greedy on Comp): Constant computation time 127
6.13 Success rate of planning methods (200 runs) 129

Chapter 1

Introduction

A mobile agent is a program which can migrate from one machine to another, performing useful actions, under its own control. It has been the subject of much attention in the last few years due to its advantage in accessing distributed resources in a low-bandwidth network [59, 30, 27]. One of the instances where a mobile agent can be very effective is in a client/server model. In a client/server model, a client may need access to a huge database on a server. This requires a large amount of data transmission over the network and may significantly waste bandwidth if the data transferred is not useful at the client side. However, a mobile agent can eliminate such wasted bandwidth. Unnecessary data transmission can be avoided by sending a mobile program to the server and by performing data manipulation at the server. Another advantage of a mobile agent is that it can perform robustly even if the connection fails because it does not require a continuous connection between the server and client.

D'Agents is a complete mobile-agent package developed at Dartmouth College [27]. It is based on the Tool Command Language (Tcl)[41], Java [10], and Python[11], which are machine-independent languages and can run on generic UNIX, PC or Macintosh machines. D'Agents is a user-friendly package. For example, an agent migration can be executed with a single instruction. It also includes simple instructions for direct communication between agents and for cloning of agents. A security system has been added so that a malicious agent cannot damage secure resources.

A desired functionality of D'Agents is agent planning based on network conditions or the situations of other agents. A planning system would decide what an agent should do next to complete its task given the current condition of the network. Since the condition of a computer network changes continuously, the sequence of the agent's actions cannot be predetermined. Thus, planning

for mobile agents needs to be executed in real time, basing decisions on the current condition of the network. The real-time planning system is necessary for efficient and robust performance of D'Agents because a mobile agent can be isolated or terminated due to unexpected disconnection of links or machine shutdowns.

Planning is a well-studied research topic, especially in the artificial intelligence fields [49, 13, 37, 18, 19]. In general, planning is defined to be the selection of a sequence of actions that will result in the achievement of a desired goal with minimal cost. Planning requires a description of the initial state of a system, the goal state and a set of possible actions and rules governing transitions between states together with their costs. Planning produces a sequence of minimal cost actions which takes the system from the initial state to the goal state.

Many useful planning methods have been proposed to deal with a static environment where the world does not change during the execution [49, 50, 23]. However, these methods cannot be applied to our mobile agent planning problem because of the dynamic changes in the problem environment during execution. Open-world planning [36] is one suitable method to handle a dynamic environment. This method decides the following action after observing the current state using sensors. It then executes the corresponding action instead of deciding the sequence of actions before execution. This open-world planning is also known as closed-loop control in the classical control literature.

The current state of the mobile agent is provided by a network sensing module. Network statistics, such as disconnection of links, shutdown of machines in the network, reliability of links, latency and available bandwidth of links, are collected by the network sensing module. Also, a mobile agent needs to obtain the location of the information resources if it is an information retrieval agent. Furthermore, it has to know about the location of other agents if it has to cooperate with them. Hence, a directory service that provides such information would be necessary.

The goal of this thesis is to study planning problems for mobile agents. Instead of solving the general planning problem, the thesis begins with a specific planning problem which is often observed in information retrieval and data-mining problems. These two applications are chosen as the domain of the planning problem because they require access to a huge amount of data across a network. Thus, usage of a mobile agent eliminates wasteful data transmission over the network.

A mobile agent often encounters cases where it has to move among several machines until it

completes its task. Although a directory service may be available to provide locations where an agent achieves its task, the service may not necessarily be able to provide the complete knowledge about a location. Therefore, the agent has to accept uncertainty with respect to successful completion of its task there. The agent migrates to some other machine if the task is unsuccessful. We assume that the directory service can provide the location information with a reference ranking, that is, the probability with which the agent completes its task at the respective location. In this situation, the planning problem is to decide in what sequence the machines should be visited in order to complete a task in minimum time.

However, the solution to the problem depends not only on the reference ranking of the machines but also on the network conditions such as the current latency, bandwidth, and load. For example, assume that there is a very busy site in Japan where an agent can finish its task with a high probability, while several unencumbered machines in the local network are of low probability. In such a situation it is difficult to decide whether to migrate to the machine in Japan, despite the cost of huge latency and computation time, or migrate to the machines in the local network. Thus, network statistics must be taken into consideration in planning. The mobile agent planning problem can thus be defined as: The problem of finding an optimal sequence of machines for a mobile agent to visit in order to complete its task in the minimum expected time, assuming the availability of the network statistics and the directory service. We named this mobile agent planning problem *Traveling Agent Problem* because of its analogy to the Traveling Salesman Problem [25].

Unfortunately, we have found that the problem is not easy to solve because it is NP-complete in the general formulation. In order to make the problem tractable, simplifications must be employed. We have analyzed how the complexity of the planning problem changes depending on different assumptions added to the general planning problem. Some of the assumptions that were imposed are: the network activity is static or dynamic, the agent(s) are single or multiple and the probability of success is variable or same. Under these assumptions, we have successfully reduced the complexity of the planning problem to a polynomial-time or pseudo-polynomial-time problem. (Pseudo-polynomial means polynomial in the values of input time, not the number of bits required to represent them.)

In addition to the development of theories and algorithms for the mobile agent planning problem, algorithms were implemented as the core of the planning system for D'Agents. The network sensing

module and the directory service module were also developed as a part of the planning system.

The thesis consists of eight chapters. The first chapter introduces the salient features of the planning problem and mobile agents. The second chapter states the planning problems involved in mobile agent applications. The third chapter describes general state-of-the-art planning methods and discusses suitable approaches to the mobile agent planning problem. The main part in the thesis is the fourth chapter where the Traveling Agent Problem is proposed and where the theories and algorithms are developed for the problem. The implementation and the experiments of the planning system are described in the fifth and the sixth chapters. The seventh chapter discusses possible future work and finally the eighth chapter covers the conclusions of the study.

Chapter 2

Problem Statement

The goal of this thesis is to study network and resource planning problems for mobile agents. A concrete result is a planning module for Agent-Tcl. This module will allow agents to finish their task with minimum execution time or with minimal resource conflicts, e.g., bandwidth, CPU load and so on.

One of the biggest potential applications for mobile agents is information retrieval [48] and data-mining [22], both of which involve access to huge amounts of data across a network. Instead of transmitting data across the network, an agent migrates to the machine where the database is located, performs its task there, and then comes back to the original machine carrying a result. Thus, the mobile agent can utilize the bandwidth of the network much more efficiently than by accessing the distributed database remotely using a direct connection. The total computation time taken can be shortened, especially when data-transmission is the bottleneck of the task.

When an agent migrates to different machines continuously, the total time of the continuous migration will vary depending on the sequence of machines to visit. In order to minimize the total execution time of a task, a planning module is required to decide the sequence of machines to visit.

The existence of a planning module becomes especially important if agents have to work in a dynamically and rapidly changing environment like a wireless network where disconnection of links, shutdown of computers, and configuration changes in the network happen frequently. In such an environment, without a planning module, an agent may not be able to even achieve its task because it can be isolated or terminated.

Let us consider mobile agent planning for information retrieval and data-mining. Required components for planning are essentially a directory service and a network-sensing module. The

directory service provides the mobile agent with a list of possible locations where the agent may be able to find its desired information. The network-sensing module collects network statistics off-line and allows the mobile agent to access them in order to obtain the current network status.

This directory is updated in real time as the network configuration and the contents of distributed databases change. In our scheme, the directory service provides not only locations where the agent might find the desired information, but also the probability that the agent will successfully find the information at each location.

The network sensing module acquires network statistics such as link status, machine shutdown, machine loads, link bandwidth, latency and the location of congestion and possibly other status information. These network statistics help agents determine how efficient and safe their migration will be. By referring to the network statistics, the mobile agent can find the best route to some specific location, decide if the location can be safely visited, and improve its operation.

The planning module is supposed to provide the sequence of actions that minimizes the agents' total execution time and maximizes the safety and robustness of their migration. The degree of safety and robustness can be expressed as part of the expected execution time since an isolated or terminated agent can be recovered by running a new copy with a startup penalty time. Therefore, we can redefine the planning problem as the problem of deciding the sequence of actions that minimizes cost, i.e., the agent's total execution time.

In general, the planning problem of minimizing a certain cost is formulated mathematically as follows:

The Planning Problem – An instance of the planning problem is given by
$\pi = < S, A, C, R, s_0, s^* >$ where

- $S = s_i$ is the set of states. $s(t)$ is the state at time t.
- $A = a_i$ is the set of actions. $a(t)$ is the action taken at time t.
- $R = r_i$ is a set of random parameters. $r(t)$ represents the random parameter at time t.
- $C = c_i$ is a set of immediate costs. $c(s(t), a(t), r(t))$ specifies the immediate cost for the action at time t.

- s_0 and s^* denote the initial and goal state respectively. The result of planning should be a sequence $< a(1), a(2), ..., a(n) >$ such that:

$$\text{total expected cost } = E\{ \sum_{i=1}^{n} c(s(i), a(i), r(i))\}$$

  is minimized.

In this planning problem, the cost is the time spent in taking an action. The possible set of states are the network conditions, more precisely, the network statistics, and possibly, the locations of other mobile agents if there is a need to communicate or cooperate with them. The set of actions includes migration to some specific machine as well as various network sensing actions that actively obtain more up-to-date network statistics. The mobile agent is allowed to take more complex actions such as communicating with other agents and making a clone.

Generally, state transition is decided based on the action a_i, the state s_i and the random variable r_i, and is therefore stochastic. However, if the state transition is assumed to be deterministic, r_i does not contribute to the state transition. In mobile agents' planning problems, the state transition will be stochastic because the state of the system is a function of the network statistics, which do not change deterministically.

Next, we consider the above definition of the planning problem with the help of a concrete example of a mobile-agent planning problem.

Example:

Assumption – *The collected network statistics are the latency and bandwidth of the links between machines and the load on each machine. When the size (in bytes) of the mobile agent and the expected duration of its computation are known, the expected value of the agent's traveling and computation time can be calculated. The network statistics stay the same during the plan's execution.*

Planning problem – The task of the mobile agent is to retrieve required information in the network. Consider the following example shown in Fig.2.1. The agent begins at its home machine "Tgdwls1". The directory service provides a list of machines ("Lost-ark", "Temple-doom", and "Bald") with the probabilities that the agent might find the information on each machine (0.9, 0.4, and 0.7, respectively). By referring to the network sensing module, the agent estimates the traveling time between the machines and the computation time at each machine. The agent is allowed to take

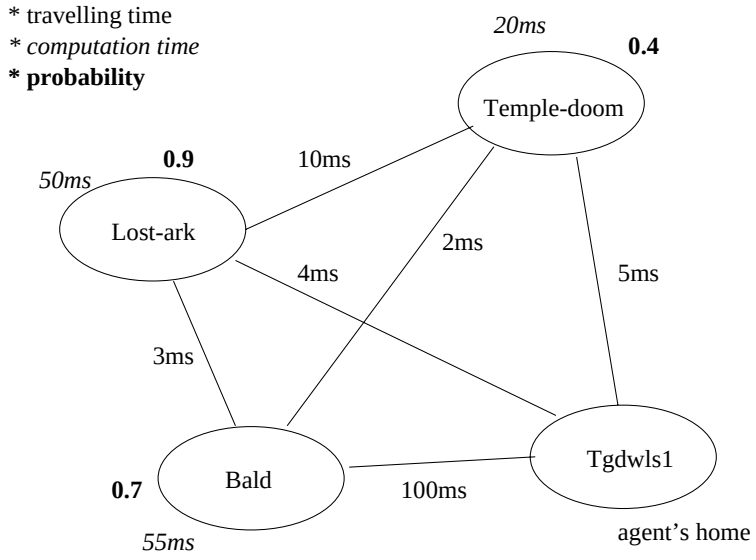

Figure 2.1: An example of the planning problem for the mobile agent

an action by migrating to any of the machines. What is the sequence of actions that the mobile agent should take in order to finish as quickly as possible? This planning problem is equivalent to a routing problem for determining the best sequence of machines to be visited.

In this example, a state is a location of the mobile agent. An immediate cost is the traveling time between machines and the computation time. A random variable is the probability of successfully finding required information. Assuming that the agent migrates the machines in the order of "Temple-doom", "Lost-ark" and "Bald", the total expected time of this tour will be:

$$Total\ Expected\ Time = 5 + 20 + \{0.4 * 5 + (1 - 0.4) * \{10 + 50 + \{0.9 * 4 + 0.1 * \{3 + 55 + 100\}\}\}\}.$$

Because of the probability of success in finding information, this planning problem is stochastic.

In the following chapter, this example will be solved using dynamic programming.

Chapter 3

Background

3.1 Possible Solution Approaches

This section surveys several state-of-the art planning techniques. Comparisons between these planning methods will suggest good possible solution techniques for the mobile agent planning problem.

3.1.1 Artificial Intelligence

Planning is one of the most studied fields in Artificial Intelligence. Instances of the planning problem, such as states, actions and state transitions, are represented in propositional logic form. Using backward or forward reasoning, the state-space is searched for the best path or the best sequence of actions that connects the start and goal states.

Traditional planning in AI (Artificial Intelligence) assumes that the system's state during execution is known in advance and that no exception will occur. The best known traditional planning algorithm is POP (Partial-Order Planner) [49, 50]. POP uses the STRIPS language [23] where the planning problem is described in terms of states, operators (or actions), the initial state and the goal state. The actions are described in terms of their preconditions and effects. POP constructs the plan, i.e., the sequence of actions, by searching the state space backward from the goal state to the initial state. The main feature of POP is that it can represent a plan in which some actions are ordered with respect to each other and others are not. As a result, it can represent all the possible plans efficiently.

An example of POP will help us understand the basic concepts of AI planning. The backward reasoning used in the POP algorithm is fundamental and still widely used in modern planning

algorithms.

Example – A mobile agent should go and collect information on tomorrow's flight schedule and weather forecast at sites FS and WF, respectively, and bring this information back to the home machine. What is the plan for the mobile agent?

Answer – First of all, the planning problem must be described in the STRIPS language. The states are represented by conjunctions of literals. In our example the initial and goal states are described:

- the initial state: At(Home) $\wedge$ $\neg$ Have(Schedule) $\wedge$ $\neg$ Have(Weather)
- the goal state : At(Home) $\wedge$ Have(Schedule) $\wedge$ Have(Weather)

The actions (operators) are described as the combination of three components: the operator description, the precondition, and the effect. An action is possible only if the precondition is satisfied. Our example has the three operators:

- Go : Op(ACTION: Go(there), PRECOND: At(here),
  EFFECT: At(there) $\wedge$ $\neg$ At(here))
- Get : Op(ACTION: Get(x), PRECOND: At(site) $\wedge$ Offer(site, x),
  EFFECT: Have(x))
- Finish : Op(ACTION: Finish, PRECOND: Have(Schedule) $\wedge$
  Have(Weather) $\wedge$ At(home)).

POP is a backward planner. The first step is to find an operator that will achieve the goal. It is not difficult to see that this operator is the finish operator. To execute the operator, its precondition, i.e., Have(Schedule) $\wedge$ Have(Weather) $\wedge$ At(home), has to be achieved. Thus, POP looks for operators that will achieve the precondition, Have(Schedule) $\wedge$ Have(Weather) $\wedge$ At(home). The Get operators for Schedule and Weather are found and added to the plan. Thus, on each step, POP extends the plan by adding an operator that achieves the precondition of the current step. If POP cannot find the operator, it backtracks to the previous step. In our example, the next step is to add the operators that achieve the preconditions of Have(Schedule) and Have(Weather). These preconditions are At(site) $\wedge$ Offer(site, Schedule) and At(site) $\wedge$ Offer(site, Weather) respectively. Using the propositions Offer(FS, Schedule) and Offer(WF, Schedule), we can find that the preconditions to be achieved are At(FS) for Have(Schedule) and At(WF) for Have(Weather). Thus, we

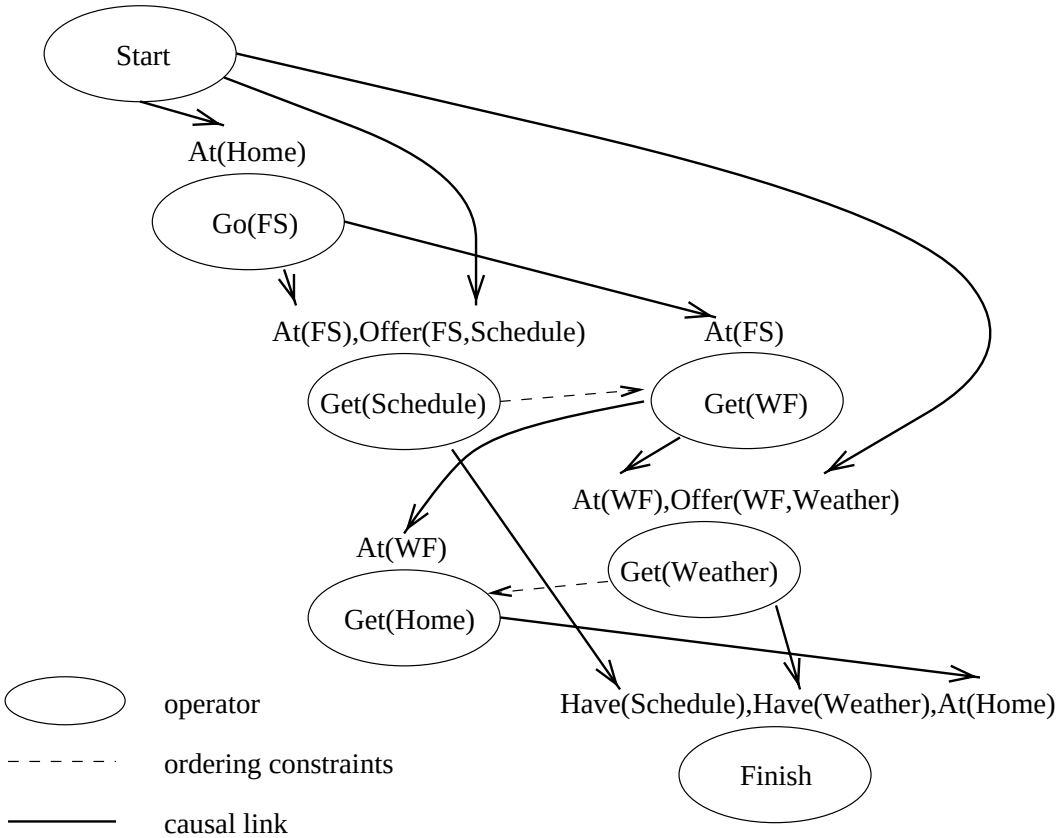

Figure 3.1: A solution to the example problem

add the Go operators Go(FS) and Go(WF) to the plan, linking them to the At(Home) condition in the initial state. This means that both Go(FS) and Go(WF) now have the precondition At(Home). Assume that we take the step Go(FS), which creates the condition At(FS) and deletes At(Home). Thus we cannot achieve the precondition At(Home) of the condition Go(WF). In the same way, the Go(WF) does not satisfy the precondition of Go(FS). Thus, if a new step is incompatible with an existing step, we say that a threat is caused. The threat is resolved by imposing an order to the incompatible steps. In order to resolve the threat in our problem, for example, we take the Go(FS) operator first and then the Go(WF) operator, by adding the link from site WF to FS. However, taking Go(WF) after Go(FS) causes a threat with Get(Schedule). This threat can be resolved by taking Go(WF) after Get(Schedule). Finally, the precondition of the Finish step At(Home) must be achieved. This can be done by adding the Go operator Go(Home) at the end of the plan. The solution to the planning problem is shown in Fig.3.1.

End of Example

POP can handle any planning problem as long as it is described in the STRIPS language, but it does not have an ability to search the state space efficiently. Thus the main focus of POP research is to develop an efficient search algorithm by employing a cost function or by adding constraints, for example, OPLAN [13]. OPLAN improves POP with a more expressive language that can represent resources and time. OPLAN then produces a plan that satisfies given resource constraints such as monetary limit. In OPLAN Resource consumption is associated with actions (steps). If a newly added action violates the resource constraint, the action is discarded and the search backtracks to the previous step to look for another action. Once a plan is produced, a scheduler decides when and where each step in the plan should be performed.

Remember that the above planning algorithms assume static and complete world knowledge. However, such assumptions are not applicable in the real environment because of the uncertainty and unpredictability therein. For example, it is very difficult to obtain a complete world knowledge if the world is changing. In addition, the effect of an action taken during execution may be unpredictable in the real environment.

To handle the uncertainty or incomplete knowledge of a dynamically changing world, a number of researchers are working on conditional planning and replanning methods. Conditional plans

include conditional branches where a sensing action chooses the appropriate partial plan for current conditions [37, 18]. The replanning method detects errors by using sensors during execution. If an error is detected, it re-plans to find another way to achieve the goal from the current situation (PLANEX [19] IPEM [1]).

A conditional planning problem can be expressed in an extended version of STRIPS. A possible sensing action would be to check if a certain information x is available at site y:

Op(ACTION: CheckInfo(x, y),

PRECOND: At(y),

EFFECT: KnowsWhether(Available(x,y))).

Using the sensing actions at the conditional branches, all possible conditional plans are built. This conditional planning can deal with incompleteness of the world by taking a sensing action. However, the scalability of the approach is a problem. The number of possible conditional plans grows exponentially with the number of conditional branches. Moreover, since only one plan is actually employed, the computation time needed to find the other plans becomes prohibitive.

The replanning method re-calculates the plan once errors are detected. There are two ways to detect errors in the replanning method. The first way, execution monitoring, compares the preconditions of the remaining steps with information collected by the sensors; any conflict is an error. Thus, even a future error can be detected. The second way, action monitoring, checks the precondition of the step which is about to be executed. If the precondition is not satisfied, an error has occurred. This method is simple and efficient, but it cannot detect any error that may happen some time in the future.

The cost of actions or the likelihood of a particular outcome can be taken into account in the planning process [15]. This information eliminates unlikely plans from consideration and either maximizes the likelihood of successful execution or minimizes the cost of execution. Decision networks [16, 38] and Markov decision processing [5] use probability theory to handle likelihood and use a cost function to handle costs.

Decision networks use Bayesian networks [43, 44] where causal links between belief states are weighted with the conditional probabilities. Based on observations or actions, the probability of a belief state is updated using Bayes' rule. There are decision states where actions should be chosen.

Actions cost a certain amount and their effects on other belief states are described in the conditional probabilities. The action that increases the probability of the goal state with minimum expected cost is chosen at the optimal action at each decision state. An advantage of a decision network is that, for each potential action, it can estimate the likelihood of the goal state without halting the planning process to perform sensing actions as in conditional planning. Its disadvantage is that the computation time for a Bayesian network that represents a complex system will be extremely large.

Markov decision processing has gained attention among AI researchers who work in problem domains that involve uncertainty and incomplete knowledge. In Markov decision processing, a problem is formulated as a set of states; An agent takes an action to move from one state to another state until it reaches the goal state; each action has an immediate cost. The objective of Markov decision processing is to find the policy (a map of states and their optimal actions) that minimizes the total expected cost (an expected accumulated immediate cost). Effective and simple calculation methods to find the optimal policy are available, such as Dynamic Programming [3, 28]. The advantages and disadvantages of dynamic programming will be discussed below.

3.1.2 Dynamic Programming

Dynamic programming has received increasing attention to treat the planning problem because of its great ability to deal with incompleteness and uncertainty in the world and to obtain an optimized plan. Dynamic programming was first proposed by Richard Bellman [3] as the modern approach to sequential decision-making problems.

In dynamic programming a system configuration is considered as a state, and the system moves from one state to another by taking an action. The next state depends on the action taken at the current state. There is a cost and a set of transition probabilities associated with each action at each state. Each transition probability specifies the likelihood that the action will lead to a certain next state. The objective of dynamic programming is to provide a optimal policy, i.e., a set of the best action at each each state, rather than a sequence of actions to be taken from the initial state to the goal state so that the expected total cost is minimal.

The essential characteristic of dynamic programming is that once the system is described as a set of states, actions, costs, and transition probabilities, the optimal actions (i.e., the optimal policy) can be calculated with simple matrix manipulation. Moreover, the transition probabilities enable

dynamic programming to handle a stochastic system, i.e., a system with uncertainty.

However, as an assumption, the effect of the actions at each state must be independent of the previous state, i.e., the effect of action should depend on the current state only and not on how the system arrived at that state (Markov property). As long as this assumption is satisfied, dynamic programming is a very general method. Because of this generality, it has a large variety of applications.

The most common algorithms for dynamic programming are the value iteration algorithm of Bellman [3], and the policy iteration algorithm of Ron Howard [28]. In this subsection, we examine these two methods.

Formulation of Dynamic Programming

Dynamic programming is formally defined by the tuple $< S, A, P, C >$ where

S : a set of states

A : a set of agent actions

P : $S \times A \rightarrow S$: state transition function

This function specifies that the outcome state of an action at the current state follows the transition probability P. $P(x,\ a,\ y)$ gives the probability that the system make a transition from state x to state y when action a is taken at x.

C : $S \times A \rightarrow \Re$: immediate cost function

This function specifies the immediate cost of a action at a state. $C(x,\ a)$ gives the immediate cost to take action a at state x. It can be a random variable.

A policy, π : $S \rightarrow A$, specifies an action for each state. The goal of dynamic programming is to find an optimal policy π^* whose expected total cost function $V_{\pi^*(x)} \leq V_{\pi(x)}$ for all states x and policies π, where

$$V_{\pi(x)} = E[C(x, \pi(x)) + \gamma \sum_{y \in S} p(x, \pi(x), y) V_{\pi(y)}]. \tag{3.1}$$

γ is a discount factor that takes a value between 0 and 1 for the infinite horizon problem so that $V_{\pi(x)}$ has a finite converged value. A larger discount factor makes the near future cost more important.

The above equation (3.1) is the basis for dynamic programming. This equation explains that the total cost from the state x following the policy π is the sum of the immediate cost of the action suggested by the policy π at the state x and the discounted expected total cost from the next state y following the policy π.

There are two main algorithms for obtaining an optimal policy π^*. The first algorithm is value iteration that obtains the optimal policy by repeated update of the total cost of each state based on the cost estimates of the other states. This update process is done using the Bellman equation:

$$V(x,a) = \min_a \{E[C(x,a)] + \gamma \sum_{y \in S} p(x,a,y) V(x)^*\}$$

where $V(x)^* = \min_a V(x,a)$. Value iteration repeats this update process until the left side converges. Then, the optimal policy $V(x)^*$ is obtained by finding for each state, the action a' such that $V(x,a') = \min_a V(x,a)$.

The next algorithm is policy iteration. Policy iteration starts with a random policy and tries to improve this policy by finding an action which is better than the current one for each state. The iteration process consists of two steps. The first step is to compute $V_{\pi(x)}$ for each state $x \in S$ under the current policy. Then, the next step is to find an action a such that

$$C(x,\pi(x)) + \gamma \sum_{y \in S} p(x,a,y) V_{\pi(y)} < V_{\pi(x)}$$

for each state $x \in S$, and if such an action a is found, $\pi(x)$ is changed to a. This iteration is repeated until no policy update is found.

Let us look at the four elements of $< S, A, P, C >$ for the agent planning problems in more detail.

S : a set of states

States in the agent planning problem should describe the situations that a mobile agent encounters in the network. In other words, the states should contain the parameters that affect the agent. Such parameters include:

1. location of agents
2. a list of machines that the agent is going to visit
3. probabilities that the agent will be able to find the desired information on each machine

4. available bandwidth between all machines in the network

5. latency between all machines in the network

6. current CPU power of the machines

7. probability of machine breakdown

8. reliability of links

The first parameter is obtained by keeping track of the agent migrations. The second and third parameters are obtained by referring to the directory. The parameters from the fourth and the seventh are collected by the network sensing module. Since the state space of dynamic programming is discrete, the value of each parameter has to be quantized. The state space will consist of all combinations of the quantized values of each parameter. It is easy to imagine how huge the state space becomes. The size of the state space increases exponentially with the number of parameters. The level of quantization determines the degree of exponential growth.

A : a set of actions

An action changes the current state to a new state. Possible actions are follows:

- move an agent to one of the sites
- clone an agent
- terminate an agent
- merge two agents
- get more accurate information about the current state

The first four actions are basic agent functionalities. The fifth action accesses the network sensing facilities to collect the most up-to-date network statistics. The state of the network is changing continuously. As more time passes, the network statistics become less accurate. By collecting the most up-to- date information, the agent can make better decisions.

P : state transition function (a set of transition probabilities)

The outcome of an action is not always deterministic since unexpected things may happen in the network during execution. For example, the user of a certain machine may turn it off unexpectedly

when the agent is migrating to that machine. Because of such uncontrollable disturbances, the state transition of an action is stochastic rather than deterministic.

Dynamic programming uses transition probabilities to deal with the stochastic nature of the world. In practice, these transition probabilities have to be obtained for all possible combinations of states and actions by collecting a huge amount of state transition data in advance.

C: immediate cost function (a set of costs)

The cost function usually represents the value to be optimized. In this case, since the planning problem is to minimize the total execution time of a task, the cost should represent time. However, there are also constraints on an agent's resource usage, so the cost should also include a penalty for over-usage of the resources. Therefore, the cost in the planning problem is a combination of weighted time and penalties.

An immediate cost in dynamic programming is the cost that is charged when a certain action is taken in a state. The total cost is the sum of all immediate costs over the course of execution. Dynamic programming finds the best action to be taken at each state to minimize the total cost. The set of these best actions is the optimal policy.

Example of Dynamic Programming

For a better understanding of how dynamic programming can be applied to the planning problems of mobile agents, consider the planning problem that was described in chapter 2 (Fig.2.1). We will solve this problem with value iteration.

Example – In the problem, the state is represented as the current location of the agent plus the set of machines where the agent has already searched the database. More specifically, the state is represented as a vector of integer elements, each element corresponding to one of the machines in the network. For example, in the planning problem in Fig.2.1, a state is [$s(1)$ $s(2)$ $s(3)$ $s(4)$]. Each element $s(i)$ indicates the situation of machine i ("Lost-ark", "Temple-doom", "Bald", and "Tgdwls1") and takes a value of 0, 1, 2, or 3. The values indicate "not searched yet", "searched already", "the agent is there without searching the database", and "the agent is there searching the database", respectively. The initial state is [0 0 0 3] which means that the agent is at the home machine "Tgdwls1" and has not searched the databases on other machines. Since the agent does not need to search the database at the home machine, the value of the home machine is set to 3.

state	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
search at Lost-ark	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
move to Temple-doom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
move to Bald	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
move to Tgdwls1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

state	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*
search at Lost-ark	0	0	0	0	0	0	0	0.1	0	0	0	0	0	0	0	0.9
move to Temple-doom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
move to Bald	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
move to Tgdwls1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3.1: The transition probability matrix for state [2 0 0 1] (25)

Terminal states are [1 1 1 3], [1 1 3 1], [1 3 1 1], [3 1 1 1] and [0 0 0 0]. The first state shows that the agent has visited all the machines and returned to the starting machine without finding the information. The second shows that the agent has found the information. An action is to move to a specific machine, changing the values of state elements from '0' to '2' and from '3' to '1', or to search the database at the current location, changing the values of a state element from '2' to '3'. For example, if the agent moves from "Lost-ark" to "Temple-doom", the state, say [3 0 0 1], changes into [1 2 0 1]. The cost for an action is the latency to move from one location to another location, or the overhead of searching a database. These latencies and overheads are depicted in Fig.2.1 in chapter 2. The transition probability is "1.0" for movement to other locations, meaning that the state transition for the movement action is deterministic. The state transition for the searching action is stochastic, and the state may transit to state [0 0 0 0] if the desired information has been found, or may just change the value of an element from '2' to '3'.

There are 32 states in the state matrix V of the planning problem as shown in Table 3.1. Table 3.1 shows the transition probability matrix P for state [2 0 0 1] (23). Please note that a number in the parentheses after a state [s(1) s(2) s(3) s(4)] represents its state number assigned in Table 3.1. The columns correspond to the destinations state and the rows correspond to the actions. For example, the "search at Lost-ark" action has probabilities 0.9 and 0.1 for next states [0 0 0 0] (32) and [3 0 0 1] (24), since the probability that the agent will find the desired information at "Lost-ark" is 0.9. For the "moving" actions, the probability that the agent will successfully move to its desired machine is 1. So we see '1' in the entry for states 15, 7 and 1, respectively.

state	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
search at current machine	0	0	0	0	0	0	55	55	55	55	55
move to Lost-ark	4	4	4	4	4	4	3	3	3	3	3
move to Temple-doom	20	20	20	20	20	20	2	2	2	2	2
move to Bald	55	55	55	55	55	55	0	0	0	0	0
move to Tgdwls1	0	0	0	0	0	0	100	100	100	100	100

state	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
search at current machine	55	55	55	20	20	20	20	20	20	20	20
move to Lost-ark	3	3	3	10	10	10	10	10	10	10	10
move to Temple-doom	2	2	2	0	0	0	0	0	0	0	0
move to Bald	0	0	0	2	2	2	2	2	2	2	2
move to Tgdwls1	100	100	100	5	5	5	5	5	5	5	5

state	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*
search at current machine	50	50	50	50	50	50	50	50	inf	inf
move to Lost-ark	0	0	0	0	0	0	0	0	inf	inf
move to Temple-doom	10	10	10	10	10	10	10	10	inf	inf
move to Bald	3	3	3	3	3	3	3	3	inf	inf
move to Tgdwls1	4	4	4	4	4	4	4	4	inf	inf

Table 3.2: The immediate cost matrix

The rest of the entries in the matrix are 0. The other states have similar transition matrixes. The immediate cost matrix is shown in Table 3.2. The columns correspond to the current states and the rows represent the actions in each state. The cost for each pair has only 2 components. One represents the searching overhead, and the other represents the movement latency. The expected-total-cost vector V for each state has one element for each action. Initially, the value of each cost element has some random value.

Once the problem is described in the form of P, C, and V, the optimal policy can be derived from Bellman's equation (1). Using value iteration, on each iteration, the expected-total-cost vector V is updated as:

$$V_{new} \leftarrow \min_{a}\{C(a) + \gamma P(a) V_{old}\}$$

where a is the policy vector and γ is a discount factor. Note that a larger discount factor makes the near future cost more important. As the number of iterations reaches infinity, V gets close to the optimal value V^*. Then, the optimal policy a^* is obtained by calculating:

$$a^* = arg\ V(a^*).$$

State	*State description*	*Optimal action*
1	0003	2
2	0013	2
3	0103	2
4	0113	2
5	1003	3
6	1013	3
7	0021	2
8	0031	2
9	0121	2
10	0131	2
11	1021	3
12	1031	3
13	1121	1
14	1131	terminal
15	0201	4
16	0301	4
17	0211	4
18	0311	4
19	1201	1
20	1301	4
21	1211	1
22	1311	terminal
23	2001	1
24	3001	4
25	2011	1
26	3011	4
27	2101	1
28	3101	4
29	2111	1
30	3111	terminal
31	1113	terminal
32	0000	terminal

Table 3.3: The optimal policy for each state

For example, the value iteration algorithm produces the optimal policy that is shown in Table 3.3. The numbers in the optimal action column, 1, 2, 3, 4, and 5 correspond to actions, "search at the current machine", "move to Lost-ark", "move to Temple-doom", "move to Bald", and "move to Tgdwls1".

As demonstrated in the example, if a set of states, actions, costs and transition probabilities are available for the planning problem, the optimal policy can be obtained easily through simple matrix calculations, such as those of value iteration or policy iteration. The usage of transition probabilities allows the planer to handle uncertainty and incomplete world knowledge. Moreover, since dynamic programming produces state-action rules rather than a sequence of actions, replanning

is not necessary. In short, the ability to represent uncertainty and cost allows dynamic programming to be applied to a wide range of real world problems.

Difficulties of solving dynamic programming problems

Although dynamic programming is attractive because of its simplicity, flexibility and applicability, it suffers from several drawbacks. In practice, it is difficult to use dynamic programming for the agent planning problem. One reason is that the state space, and thus the computation time, is extremely large. Another reason is that conventional dynamic programming has problems with a dynamically changing environment such as a computer network. The final reason is that it is difficult for conventional dynamic programming to handle an environment where the current state is not observable. A network sensing module may not be able to measure some of the network statistics, for example, and thus the agent might not be able to determine its current state. Let us look at these three difficulties in detail.

The state space is extremely huge

As you have seen before, the state is made up of several different parameters. Moreover, each parameter has a range of values. It is easy to imagine how large the state space can become. In particular, the continuous parameters, such as latency and CPU power, will increase the size of the state space easily if finer quantization is required . In addition, parameters such as latency and bandwidth are actually matrices, each of their elements is the value for a link in the network. Thus, it is almost unrealistic to use all possible parameter combinations as the resulting state-space size will be enormous. The size of the state space directly affects the computation time, so tremendous computation power will be required to solve the problem unless the size of the state space is somehow decreased.

The network is dynamically changing

Conventional dynamic programming assumes that the transition probabilities $P(x, a, x')$ and the immediate cost function $C(x, a)$ of a system are available a priori. Once the system configuration changes, the probabilities and the cost function must be recollected and the optimal policy must be recalculated from scratch.

However, a computer network can change its configuration frequently. For example, adding of new computers, upgrading computers and operating systems, and users leaving and registering are

all considered configuration changes and these are observed every few days or hours. In addition, for wireless computer networks, which are of particular interest in our work , changes in geographical configuration can occur every few minutes.

Thus, conventional dynamic programming is not suitable for the agent planning problem in a dynamic computer network. We have to look for other methods that can obtain the optimal policy in a dynamically changing system. These methods should not require the transition probabilities and immediate cost function to be available in advance.

Uncertainty in Network Statistics (partially observable states)

We are studying efficient network sensing methods to collect network statistics. Most the efficient network sensing methods collect information at certain intervals. The information will contain uncertainty depending on how new it is. Since part of the system's state is determined by network statistics, determining the state of the system would involve uncertainty. And without knowing the state, an accurate optimal decision cannot be made.

Actively requesting network statistics would decrease the uncertainty, and make the agent's decision more accurate. However, this would take time and also require network resources (e.g., bandwidth). It is necessary to have some criteria that determines how often the sensing is done. Questions such as whether the sensing should be done at fixed intervals, or whether it should be done depending on the agent's current state because some states may involve riskier decisions than others, now arise.

Overcoming the difficulties

We have surveyed some difficulties that affect the agent planning problem. In this section, we discuss possible solutions to these difficulties. One difficulty is the huge size of state space. There are several possible ways to reduce the size of the state space. The first approach is to use feature extraction which maps a state i into some small set of features $f(i)$. The second approach is to aggregate states into subsets. Another difficulty is uncertainty in the dynamically changing system environment. Q-learning will be a suitable approach to this difficulty. Let us discuss these approaches in detail.

Feature extraction

Decreasing the number of parameters is one way to decrease the size of the state space. Feature extraction [6] is used to map the original parameters into a small number of parameters, i.e., features,

that capture the important characteristics of the state. These features are usually identified by humans based on the knowledge of, or the experience with, a system. This method will be especially useful when we deal with bandwidth and latency matrices. The matrices will be reduced to vectors which have much less complexity.

Given a feature vector $f(i) = (f_1(i), f_2(i), ..., f_m(i))$, one wants to find the best coefficient $r(k)$ for each elements f_k of the vector in the following:

$$V(f(x), r) = \sum_{k=1}^{m} r(k) \cdot f_k(x)$$

The best coefficients will be the one which minimize :

$$\sum \{V(x) - \sum_{k}^{m} V'(f_k(x),\ r(k))\}^2.$$

These coefficient can be obtained using linear or non-linear least square methods.

Aggregation method

The aggregation method [5] turns the system into another system of smaller dimension. The states of the original system are merged into aggregated subsets $S_1, S_2, ..., S_m$. These subsets must cover the entire space

$$S = S_1 \bigcup S_2 \bigcup ... \bigcup S_m.$$

Aggregation of states is useful when the transitions between subsets capture the global behavior of the original system. In this case, the states in each aggregated subset have similar state transitions.

Q-learning

As mentioned before, conventional dynamic programming is not suitable for a system whose environment is dynamically changing. Recently, Q-learning [58] was introduced as a potential solution to this problem. Unlike conventional dynamic programming, Q-learning does not require the transition probabilities and immediate cost function to be available a priori. Thus, Q-learning can handle a system whose behavior (whose transition probabilities and cost function) is unknown or a dynamically changing system whose behavior become unknown every time the system changes. However, application of Q-learning to a dynamically changing system requires the change to be enough slow so that Q-learning can obtain the optimal solution because the convergence proof Q-learning requires

stationarity of the system (i.e., the transition probabilities and immediate cost function stay the same).

Q-learning uses additional notations compared to conventional dynamic programming. Let us define $Q^*(x,a)$ as a Q-factor, the expected cost of taking action a in state x followed by the optimal policy. The optimal cost of x, $V^*(x)$ can be expressed as:

$$V^*(x) = \min_a Q^*(x,a).$$

Thus equation (3.1) can be written as:

$$Q^*(x,a) = C(x,a) + \gamma \sum_{x'} P(x,a,x') \min_{a'} Q^*(x',a).$$

Then the optimal policy $\pi^*(x) = arg\min_a Q^*(x,a)$.

The Q-factors can be calculated on-line without using C and P as follows:

$$Q(x,a) = (1-\beta)Q(x,a) + \beta\{c(x,a) + \gamma \max_{a'} Q(x',a')\}$$

where $c(x,a)$ is the observed cost for action a at state x, x' is the observed next state, and $0 \leq \beta \leq< 1$.

$Q(x,a)$ will converge with probability 1 to the optimal value $Q^*(x)$, if (1) each action at each state is executed an infinite numbers of times and (2) β satisfies the following:
For all x, a,

$$\sum_{i(x,a)} \beta_{i(x,a)} = \infty \quad and \quad \sum_{i(x,a)} \beta^2_{i(x,a)} \leq \infty$$

where $i(x,a)$ is the number of updates of $Q(x,a)$.

As we can see, Q-learning derives the optimal policy based on history samples only. One requirement is to control the system so that all actions in all the states are explored an infinite numbers of times. Then, the optimal policy can be found even if the system is dynamically changing as long as the system's change is slow enough. However, this requirement is hard to satisfy in real systems. Although error bounds are used to get convergence in a finite number of iterations, the convergence rate of Q-learning is still very slow, which makes the application of Q-learning less attractive.

	model-based	*cost*	*uncertainty*	*optimization*	*scalability*
partial order planning	yes	possible	no	no	no
conditional planning	yes	possible	yes	no	no
re-planning	yes	possible	yes	no	no
dynamic programming	yes	yes	yes	yes	no
Q-learning	no	yes	yes	yes	no

Table 3.4: Performance Table

3.2 Comparison of the Different Approaches

We have considered several different planning methods. In this subsection, we compare these approaches using several criteria. The result of the comparisons is shown in Table 3.4.

- Model-based

  Any method that requires a complete model of the problem domain is called a model-based method. The first four methods are model-based, while the last, Q-learning, is a model-free method. Unlike conventional dynamic programming, Q-learning does not require transition probabilities or the cost function to be known in advance. Although known as a model-free method, the state space and actions for Q-learning must be known in advance, and they cannot be altered during execution. The model-free method is best applied to situations where the environment changes dynamically, i.e., where the transition probabilities or the cost function changes during the execution.

- Representation of cost

  All the methods in Table 3.4 are able to represent cost (or time). Although the first three methods do not have the ability to represent cost by themselves, a "post-process" scheduling step can be performed to impose cost constraints on their generated possible plans. An ability to represent cost is built-in for dynamic programming as well as Q-learning. The handling of cost is much more efficient in dynamic programming than in the previous three methods.

- Handling of uncertainty

  Traditional POP assumes that perfect world knowledge is available before execution and that the world never changes during execution. Thus, it cannot handle uncertainty from lack of world knowledge and any unexpected change in the world. The second and third methods

compensate for the first one's (i.e., POP's) lack of handling of uncertainty and a dynamically changing world with a feature to detect an error of a plan and to alter the plan during execution. Both dynamic programming and Q-learning are designed to efficiently handle the stochastic world where uncertainty can be represented with probabilities.

- Availability of optimization theory

  One advantage of dynamic programming and Q-learning over AI planning methods is that the optimality of their solutions has been proved mathematically [5, 3, 58].

- Scalability

  All the methods suffer from a scalability problem. For example, in the AI planning methods, the size of the state space increases exponentially as the number of literals used in the problem description increases. In the same way, with dynamic programming and Q-learning, the size of the state space increases exponentially as the number of parameters used to describe the state increases. Aggregating states into subsets is one common method to solve the scalability problem in all five methods.

To summarize, dynamic programming, including Q-learning, is more suitable to the problem of minimizing cost or time in an uncertain environment than AI planning methods because (1) there is mathematical guarantee of optimality and (2) they can handle uncertainty more easily or more efficiently. If the environment changes during the execution or its transition probabilities and cost function are not available a priori, Q-learning is the method of choice among dynamic programming methods because of its model-free nature. However, Q-learning has to overcome the problem of slow convergence in addition to the scalability problem. On the other hand, if it is feasible to assume a stationary environment during the entire execution, conventional dynamic programming should be employed as the solution to the planning problem, as this would avoid the slow convergence problem of Q-learning.

This thesis assumes that the environment is stationary or that the dynamically changing environment can be considered as a sequence of different stationary environments. The transition between one stationary environment to another can be handled by the re-planning method that detects the change of the environment and recalculates a plan in the new environment. This can be seen in

the example of dynamic programming solution in section 3.1.2, where a change of network statistics (e.g., latencies and CPU load) results in a new plan based on the new network statistics.

Thus, conventional dynamic programming is employed as the solution to the mobile agent planning problem and overcoming its scalability problem is one of the main topics of this thesis.

Chapter 4

Theory and Algorithm

Dynamic programming is suitable technique for the mobile agent planning problems, in uncertain environments, as mentioned in the previous chapter. However, dynamic programming cannot be applied to the planning problem without solving several challenging subproblems, such as those of scalability and complexity.

In this chapter, the sequencing problem which arises naturally in the planning of mobile agents is proposed. We name the problem the *Traveling Agent Problem*(TAP). It is NP-complete in its general formulation. However, by making simplifying assumptions, we have proved that the complexity of the problem can be reduced to be polynomial or pseudo-polynomial. This chapter is devoted to the study of several simplified subproblems along with proofs of their complexity and solution algorithms that use dynamic programming.

This chapter consists of 5 sections. The first section proposes TAP with a proof of its NP-complete complexity. The second section introduces assumptions such as (1) constant latencies between machines in the network and (2) constant latencies between machines in the same subnetwork, which successfully reduce the complexity of TAP. The third section deals with the Traveling Multiple Agent Problem where multiple agents cooperate with each other to complete a same task in the network. The fourth section introduces the deadlines after which resources in the network become unavailable. The final section concludes this chapter by solving TAP with deadlines where multiple agents are involved.

4.1 Traveling Agent Problem

Suppose you are shopping for a specific item known to be sold in $n+1$ stores, s_i where $0 \leq i \leq n$. The probability that store i has the item is known and given by p_i. Moreover, it takes a known time t_i to navigate through store i to the section where the item is stocked thereby determining whether the item is available or not. Going from store i to store j requires travel time l_{ij}. Starting and ending at store s_0, with $0 = t_0 = p_0$, what is the minimal expected time to find the item or conclude that it is not available?

As this shopping analogy suggests, once the item is found, you are done and can return to s_0 by the most expedient route which may or not may be taking the direct path requiring l_{i0} travel time if the item was found at s_i. With probability

$$\prod_{i=1}^{n}(1-p_i) = \prod_{i=0}^{n}(1-p_i)$$

all sites must be visited. Moreover, once a store is visited and found not to have the item in stock, there is no reason to go back to that store. Probabilities for success at different sites are assumed independent. Site s_0 is the start and end of the shopping task and can be considered as "home." Since $0 = p_0 = t_0$, s_0 only contributes to the problem through latencies.

This problem arises when planning the actions of mobile software "agents" [27, 30, 59]. Using the shopping metaphor from above, the "stores" are information servers such as databases or web servers. The probabilities of success, p_i can be thought of as estimated from relevance scores given by search engines such as Altavista, Infoseek and others. Computation times, t_i, and latencies, l_{ij}, are obtained from the network sensing module.

In this framework, an agent has a specific information request to satisfy, such as finding a topographical map of a given region. The search engine or directory service identifies locations, s_i for $i = 1, ..., n$, together with probabilities (relevance scores), p_i, for finding the required data at the corresponding sites.

The agent then queries the network status monitor to find latencies and estimated computation times, l_{ij} and t_i, for those sites. Based on this information, an autonomous agent must plan its itinerary to minimize expected time for successfully completing the task. In addition to this mobile agent application, there are numerous other planning and scheduling problems which can be

formulated in these terms [42].

Formally, the Traveling Agent Problem is defined as follows:

> *The Traveling Agent Problem* – There are $n+1$ sites, s_i with $0 \leq i \leq n$. Each site has a known probability, $0 \leq p_i \leq 1$, of being able to successfully complete the agent's task, and a time $t_i > 0$, required for the agent to attempt the task at s_i regardless of whether it is successful or not. These probabilities are independent of each other. Travel times or latencies for the agent to move between sites are also known and given by $l_{ij} \geq 0$ for moving between site i and site j. When the agent's task has been successfully completed at some site, the agent must return to the site from which it started (i.e., site 0). For site 0, $p_0 = t_0 = 0$. The *Traveling Agent Problem* is to minimize the expected time to successfully complete the task.

Several comments are appropriate.

- The latencies, l_{ij}, can be assumed to be the minimal travel time between nodes i and j as they would typically be used in network routing. This observation avoids the situation where an indirect path between nodes, without "stopping" at the sites along the indirect path, might be shorter that the direct path.

- The probabilities, p_i, can be thought of as conditioned on attempting the task at site i. That is

$$p_i = \text{Prob(success at site } i \mid \text{site } i \text{ not visited yet)}.$$

  Accordingly,

$$\text{Prob(success at site } i \mid \text{site } i \text{ has been visited)} = 0 \text{ or } 1.$$

  This formally handles the site revisiting issue.

- This problem can be formulated as a Markov Decision Problem or discrete stochastic control problem [5, 28] in which the state space consists of vectors indexed by sites with coordinate values indicating whether a site has been visited already or not. Standard dynamic programming algorithms could be used on this formulation but since the state space is exponentially large in the number of sites, this formulation is not scalable. However, we will see that in certain

cases, the state space can be simplified leading to efficient dynamic programming solutions. This will be shown in the following sections.

- If all sites must be visited (so that p_i is irrelevant), then the problem reduces to the classical Traveling Salesman Problem (TSP) which is known to be NP-Complete.

A solution to the Traveling Agent Problem consists of specifying the order in which to visit the sites, namely a permutation $< i_1, i_2, ..., i_n >$ of 1 through n. Such a permutation will be called a *tour* in keeping with the tradition for such problems.

The expected time to complete the task or visit all sites in failure, for a tour $T =< i_1, i_2, ..., i_n >$ is

$$C_T = l_{0i_1} + t_{i_1} + p_{i_1} l_{i_1 0} + \sum_{k=2}^{n} \left\{ \left(\prod_{j=1}^{j=k-1} (1 - p_{i_j}) \right) \right\} (l_{i_{k-1} i_k} + t_{i_k} + p_{i_k} l_{i_k 0}) + \prod_{j=1}^{n} (1 - p_j) l_{n0}. \quad (4.1)$$

This formula can be understood as follows. The first site, s_{i_1}, on the tour is always visited and requires travel time l_{0i_1} to be reached. Upon arrival, time t_{i_1} must be spent there regardless of success or failure. With probability p_{i_1} the task is successfully completed in which case the agent can return to site 0 with time cost $l_{i_1 0}$. However, with probability $(1 - p_{i_1})$ there was failure and the agent proceeds to site i_2. The expected value of the contribution involving moving from site i_1 to site i_2 and succeeding there is

$$(1 - p_{i_1})(l_{i_1 i_2} + t_{i-2} + p_{i_2} l_{i_2 0}).$$

Here the factor $(1 - p_{i_2})$ is the probability of failing at site i_2. The third term comes from failing at sites i_1 and i_2 and so has the probability $(1 - p_{i_1})(1 - p_{i_2})$ which is multiplied by the expected time for success at site i_3. The general term then has the form:

(probability of failure at the first $k - 1$ sites) $\times$ (expected time for success at site i_k).

Finally, the last term arises when failure occurs at all nodes and we must return to the originating site. We have used independence of the various probabilities here. Not surprisingly, this problem is NP-complete [31] as will be shown below. Note that in the proof, the question is altered so that we are asking whether there is a tour whose cost, as above, is smaller than or equal to some total length B. This can be used in a binary search method to find the minimum.

Theorem 1 *The Traveling Agent Problem (TAP) is NP-Complete.*

Proof – We start by showing that TAP belongs to NP. Given a tour, T, we can verify if the total expected length C_T is smaller than or equal to B by merely using the formula. This verification can clearly be performed in polynomial time (O(n^2) steps specifically). Thus, TAP belongs to NP.

Next, we show that the problem is NP-Hard, by proving that the Hamiltonian Cycle Problem [25] can be reduced to TAP. A Hamiltonian Cycle in graph $G =< V, E >$ is a simple circuit that includes all the vertices V. Thus, a cycle is expressed as an ordering of the vertices $< v_1, v_2, ..., v_k >$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k$ and $\{v_k, v_1\} \in E$.

Define TAP with probabilities strictly between 0 and 1 and

$$l_{ij} + t_j = \begin{cases} 0 & \text{if} \quad i, j \in E \\ 1 & \text{if} \quad i, j \notin E, \end{cases}$$

so that $t_j = 0$ for any vertex on an edge in E and $l_{ij} = 0$ for any edge in E. This formation can be done in polynomial time.

The graph G has a Hamiltonian cycle if and only if the corresponding TAP has a tour with expected cost of 0. To see this, assume that the graph G has a Hamiltonian cycle H. The corresponding tour T in TAP will have cost 0 because all the time costs for T are 0. On the other hand, if the tour T has cost 0, the latencies and site times must all be 0 along this tour by construction. Here we use the fact that the probabilities are strictly between 0 and 1 so that the only way for the cost to be 0 is for the times to be 0 along the tour. Thus graph G has a Hamiltonian cycle since all the edges in the tour T have to belong to E by construction again. Q.E.D.

4.2 Single Agent

4.2.1 Constant Latencies

The complexity of the problem can be reduced when latencies between nodes are equal. For example, if the processing time at each node is extremely large (compared to the latency between the nodes), differences among the latencies could be ignored, or even taken to be zero. Alternately, if no information about internodal latencies is known, we might assume all of them to be constant. The constant latency assumption is reasonable in the case of a single subnetwork as well. Accordingly, the assumption that we employ in this section is:

Assumption 1 *Latencies between nodes are all the same.*

Under this assumption, it turns out that TAP can be solved in polynomial time, as we will see below.

Theorem 2 *Assume that in TAP* $l = l_{ij} = l_{km} \geq 0$ *for all* i, j, k, m*, namely Assumption 1. Computation times,* t_i*, and probabilities,* p_i*, can be arbitrary. Then the optimal tour for TAP is attained if the nodes are visited in decreasing order of* $p_i/(t_i + l)$*.*

Proof – The proof uses an interchange argument commonly used in finance and economics. See [4] for example. For notational convenience and without loss of generality consider the specific tour $T = <1, 2, 3, ..., n>$. The total expected cost for the tour T is as in equation (1) with notational changes:

$$\begin{aligned} C_T &= l_{01} + t_1 + p_1 l_{10} + \sum_{k=2}^{n} \{(\prod_{j=1}^{k-1}(1-p_j))\}(l_{k-1k} + t_k + p_k l_{k0}) + \prod_{j=1}^{n}(1-p_j) l_{n0} \\ &= l + t_1 + p_1 l + \sum_{k=2}^{n} \{(\prod_{j=1}^{k-1}(1-p_j))\}(l + t_k + p_k l) + \prod_{j=1}^{n}(1-p_j) l \\ &= 2l + t_1 + \sum_{k=2}^{n} \{\prod_{j=1}^{k-1}(1-p_j)\}(t_j + l) \end{aligned}$$

where we have used the fact that

$$p_k \cdot l \cdot \prod_{j=1}^{k-1}(1-p_j) + l \cdot \prod_{j=1}^{k}(1-p_j) = l \cdot \prod_{j=1}^{k-1}(1-p_j).$$

This also eliminates the last term so note that p_n does not explicitly arise in the final expression because regardless of the value of p_n, we return to s_0 after visiting s_n.

Now consider the effect of switching the order of two adjacent sites on the tour, say i and $i+1$ for some $1 \leq i \leq n-1$. Call this new tour, T'.

In the above expression for the expected cost, only the ith and $(i+1)$st terms are affected by the switch. The terms appearing before the ith term do not contain anything which involves i or $i+1$. On the other hand, terms that follow the $(i+1)$st term all contain $(1-p_i) \cdot (1-p_{i+1})$ in precisely

the same way but no other ingredients that depend on either i or $i+1$. Note that for $i+1=n$ there are no terms following the two terms we are isolating so we can handle that case as well by the following argument.

The difference in expected cost between T and T' can be calculated by comparing only these two terms. The difference in the expected cost is therefore:

$$\begin{aligned} C_T - C_{T'} &= (t_i + l)\prod_{j=1}^{i-1}(1-p_j) + (t_{i+1}+l)\prod_{j=1}^{i}(1-p_j) \\ &\quad - (t_{i+1}+l)\prod_{j=1}^{i-1}(1-p_j) - (t_i+l)(1-p_{i+1})\prod_{j=1}^{i-1}(1-p_j) \\ &= (t_i + l + (t_{i+1}+l)(1-p_i) - t_{i+1} - l - (t_i+l)(1-p_{i+1}))\prod_{j=1}^{i-1}(1-p_j) \\ &= (p_{i+1}(t_i+l) - p_i(t_{i+1}+l))\prod_{j=1}^{i-1}(1-p_j). \end{aligned}$$

Thus, T is a better tour (has smaller expected cost) if

$$p_{i+1}(t_i+l) < p_i(t_{i+1}+l)$$

or equivalently,

$$\frac{p_i}{t_i+l} > \frac{p_{i+1}}{t_{i+1}+l}.$$

This shows that when two adjacent sites have the above ratios out of order (that is the ith site on the tour has a smaller ratio than the $(i+1)$st site), then we can decrease the expected cost by switching them.

So we can, for example, perform a Bubble Sort on any initial tour ordering and every swap in the Bubble Sort will decrease the expected time for the tour. The optimal value is then the sequence with decreasing ratios as above. Q.E.D.

Since it is possible for some of the p_i and/or (t_i+l) to be zero, we should handle that case as well. Clearly, any site for which $p_i = 0$ should not be visited at all, that is, it should be dropped from any prospective tour. Moreover, it might be that $t_i + l = 0$ for some of the remaining sites. In that case, we can modify the final steps of the above proof so that we visit the sites in order of increasing $(t_i+l)/p_i$ since $p_i \neq 0$.

Theorem 3 *Suppose that all latencies are the same and that some $p_i = 0$ and $t_i + l = 0$. Then the optimal tour consists of:*

- *First dropping sites with $p_i = 0$;*
- *Secondly, sorting the ratios for the remaining sites,*

$$\frac{t_i + l}{p_i},$$

into increasing order and visiting the sites in that order.

An important observation is the fact that $t_i + l$ is the expected time to reach site i and process there after failing at site $i-1$. We would obtain the same results if we used the expected time after site i but before reaching sites $i+1$ or the home sites, s_0. That expected time is

$$t_i + (1 - p_i)l + p_i l$$

which is equal to $t_i + l$ and does not change the above result.

In fact, the expected time, $t_i + l$ can be replaced by any expected time which is *independent* of the position of the site within the tour. This will be of key importance in the next subsection.

This observation and a small construction allows us to solve the following modified problem.

Theorem 4 *Suppose the latency from the home site, s_0, to all the other sites is a constant, $l' \neq l$, where l is the latency between sites i and j for $1 \leq i, j \leq n$. Then the optimal tour is still obtained by:*

- *Dropping sites with $p_i = 0$;*
- *Sorting the sites into increasing order of*

$$\frac{t_i + l}{p_i}.$$

Proof – Create a new site, s^*, whose latency to all sites s_i for $i > 0$ is l and whose latency to site s_0 is $l' - l$ (this might be negative but it does not affect the argument). Consider any tour, T, using this site, s^*, as the home. Call the cost of the tour using s^* as home, D_T. Let the cost of the tour for the original problem be C_T.

Then the relationship between the costs of the tours is

$$D_T = C_T - 2(l' - l)$$

because any tour must start and end at either s_0 or s^*, even if $l' - l < 0$. Because of this relationship between costs, the minimal expected time tour for the two problems are the same. The best tour using s^* can be computed using Theorem 1 and by the above, is the best tour for this modified problem as well. Q.E.D.

We will use a simple technical lemma based on the proof of Theorem 1. For this lemma, suppose that we have a general TAP with arbitrary latencies. This means that the expected time to visit site s_i is measured from the time an agent arrives at s_i until it either successfully finishes or travels to the next site. If site s_i is followed by site s_j, then that time is

$$t_i^* = t_i + p_i l_{i0} + (1 - p_i) l_{ij}.$$

In general, this time is variable but the lemma addresses the case when a subsequence of a tour can be rearranged without affecting these expected times.

Theorem 5 *Suppose that we have a general TAP with a tour,* $T =< s_1, s_2, s_3, ..., s_n >$. *Suppose that for a subsequence of the tour,* $< s_i, ..., s_j >$ *for* $i < j$, *any permutation of the sites* $s_i, ..., s_j$ *results in the same expected execution times,* t_i^*, *for each of those sites, then the optimal ordering for the subsequence (that is, the permutation of the subsequence that minimizes expected time) is obtained by the permutation* $< s_{k_i}, s_{k_{i+1}}, ..., s_{i_j} >$ *in which*

$$\frac{t_{k_i}^*}{p_{k_i}} \leq \frac{t_{k_{i+1}}^*}{p_{k_{i+}}} \leq ... \leq \frac{t_{k_j}^*}{p_{k_j}}.$$

Proof – As in the proof of Theorem 2, consider switching two adjacent sites, say $s_{k_m}, s_{k_{m+1}}$, in any ordering of the subsequence in question. Call the original tour T and the tour with switched sites T'. Since the expected times for these sites are independent of their ordering in the subsequence, we as above see that

$$C_T - C_{T'} = P(p_{k_{m+1}} t_{k_m}^* - p_{k_m} t_{k_{m+1}}^*) \leq 0$$

if and only if

$$\frac{t_{k_{m+1}}^*}{p_{k_{m+1}}} \geq \frac{t_{k_m}^*}{p_{k_m}}.$$

Thus this sorted order minimizes the expected time for the subsequence of sites. Q.E.D.

4.2.2 The Multiple Subnetwork Case

Assumption 1 and Theorem 2 address the case of completely constant latencies. Theorem 3 offers a small generalization in which latencies to the home node can be different but still constant. However, many situations can be modeled by variable latencies which are constant within subnetworks and across subnetworks. Specifically, consider the case of two subnetworks (one in Japan and one in the US). Latencies between any two nodes within the same subnetwork are constant as are latencies across the two subnetworks. That is, for sites in Japan, latencies are a constant, l_J, and in the USA they are l_U. Latencies between two nodes, one in Japan and one in the USA, are a third constant, l_{JU}. Formally, we define the Two Subnetwork Traveling Agent Problem (TSTAP) as follows.

Assumption 2 *The relevant sites belong to two subnetworks, S_1 and S_2. Sites in S_i are s_{ij} where $1 \leq j \leq n_i$. There are three latencies: $L_1, L_2, L_{12} \geq 0$. For $s_{1j} \in S_1, s_{2k} \in S_2$, $l_{1j2k} = l_{2k1j} = L_{12}$ while for $s_{1j}, s_{1k} \in S_1$, we have $l_{1j1k} = l_{1k1j} = L_1$. Similarly, for $s_{2j}, s_{2k} \in S_2$, we have $l_{2j2k} = l_{2k2j} = L_2$. Probabilities, $p_{mj} > 0$ are nonzero and independent as before. Computation times $t_{mj} \geq 0$ are arbitrary but nonnegative. Latencies between the home site, s_0, and sites in S_i are L_{0i}. We assume that $L_{0i}, L_{i2} \geq L_i$. That is, latencies within a subnetwork are smaller than latencies across networks and to the home sites.*

Under Assumption 2, the Two Subnetwork Traveling Agent Problem (TSTAP) can be solved in polynomial time using the results of Theorem 5 and dynamic programming. The result can be generalized in several ways but we defer that discussion until after the basic case is handled. Formally, we will show:

Theorem 6 *The optimal (minimal expected time) sequence for the TSTAP can be computed in polynomial time, $O(n_1 \log n_1 + n_2 \log n_2 + (n_1 + 1)(n_2 + 1))$.*

Outline of the proof – The proof consists of two steps. The major difference between this problem and the previous cases of all constant latencies is that now the optimal solution requires making choices about whether to stay in the same subnetwork or to cross over to the other subnet after each site is visited.

We first show that the order in which sites are visited in one of the subnets is given by the ordering specified by Theorems 2, 3 and 4. This greatly reduces the number of choices necessary – after visiting a site, we only need to decide which of the eligible sites, one per subnetwork at most, should be visited next. The sorting requires $O(n_1 \log n_1 + n_2 \log n_2)$ steps.

This sorted ordering is used in the second step where a dynamic programming algorithm is used to compute the optimal solution. Even though the problem is stochastic, it can be solved by a deterministic dynamic programming algorithm in roughly $O((n_1+1)(n_2+1))$ steps.

Proof – As before, we eliminate all sites with $p_{ij} = 0$ since they contribute time but no possibility of solution.

ASSERTION 1 – Suppose that the optimal tour is

$$< s_{i_r j_r} >=< s_{i_1 j_1}, s_{i_2 j_2}, ..., s_{i_M j_M} >$$

where $M = n_1 n_2$. Without loss of generality, let the sites in S_i be visited in this order:

$$s_{i1}, s_{i2}, s_{i3}, ..., s_{in_i}.$$

Then

$$\frac{t_{i(j-1)} + p_{i(j-1)} L_{i0} + (1 - p_{i(j-1)}) L_i}{p_{i(j-1)}} \leq \frac{t_{i(j)} + p_{i(j)} L_{i0} + (1 - p_{i(j)}) L_i}{p_{i(j)}}$$

for $1 \leq j - 1 \leq n_i - 1$.

We will show the result for subnetwork 1 and then see that it applies by symmetry to subnetwork 2 as well.

First of all, note that if a tour consists of consecutive visits to sites within S_1, then those sites within S_1 must be ordered according to the claim of the theorem by the interchange argument of Theorems 2, 3 and 4. That is, switching any two adjacent sites, s_{1j} and $s_{1(j+1)}$, within S_1 (without a intervening trip to S_2) leads to a difference in the expected time that is precisely of the form seen before. This means that the ordering has to follow

$$t^*_{1j}/p_{1j} \leq t^*_{1(j+1)}/p_{1(j+1)}.$$

Notice that although the last site within S_1 before crossing over to S_2 has a latency $L_{12} > L_1$ but this latency is independent of which S_1 is the last.

The only remaining case is when two sites within S_1 are separated on a tour by a visit to some sites within S_2. This case establishes the claim of the theorem by using aggregated sites and works only for the optimal tour, not any valid tour. We will point out where optimality of the tour is used.

Define *metasites* to be aggregated sites within subnetwork 2 that are visited *between* visits to subnetwork 1. Using the above notation, suppose that between visiting s_{1j} and $s_{1(j+1)}$ we visit only sites within subnetwork 2, say $s_{2k}, ..., s_{2m}$. We will treat the subnetwork 2 sites $s_{2k}, ..., s_{2m}$ as if they were a single site.

The expected time from starting at s_{1j} and arriving at s_{2k} is technically $t_{1j} + p_{1j}L_{10} + (1-p_{1j})L_{12}$ and the probability of success is p_{1j}. However, let us define a modified site, s^*_{1j} with

$$t^*_{1j} = t_{1j} + p_{1j}L_{10} + (1 - p_{1j})L_1$$

as the new expected time and probability of success $p^*_{1j} = p_{1j}$ as before.

For the *metasite* $ms_{2k:2m}$, define the expected time to be

$$\begin{aligned} t_{ms_{2k:2m}} \quad = \quad & L_{12} - L_1 + t_{2k} + p_{2k}L_{20} + (1 - p_{2k})L_2 \\ & + (1 - p_{2k})(t_{2(k+1)} + p_{2(k+1)}L_{20} + (1 - p_{2(k+1)})L_2 + ... + (1 - p_{2m})L_{12}..)))). \end{aligned}$$

The probability of success for this metasite is

$$p_{ms_{2k:2m}} = p_{2k} + (1 - p_{2k})p_{2(k+1)} + (1 - p_{2k})(1 - p_{2(k+1)})p_{2(k+2)} + ... \prod_{i=k}^{m-1}(1 - p_{2i})p_{2m}.$$

This time is merely the expected time to start at s_{2k} and either finish successfully or travel and start at site s_{1k} together with the difference $L_{12} - L_1$, which is leftover from the s_{1j} to s_{2k} travel time that is not accounted for in t^*_{1j}.

By splitting the expected time for site s_{1j} and the metasite $ms_{2k:2m}$ in this way, we see that the expected time to complete visiting s_{1j} and $ms_{2k:2m}$ is independent of whether another S_1 node follows s_{1j} or such a metasite follows s_{1j}. Similarly, the time for $ms_{2k:2m}$ is independent of which S_1 site precedes it.

To reiterate, we have redefined the expected time for s_{1j} so that it is as if the next site were an S_1 site instead of $ms_{2k:2m}$. Moreover, the excess latency we removed from s_{1j} has been added to the time for $ms_{2k:2m}$.

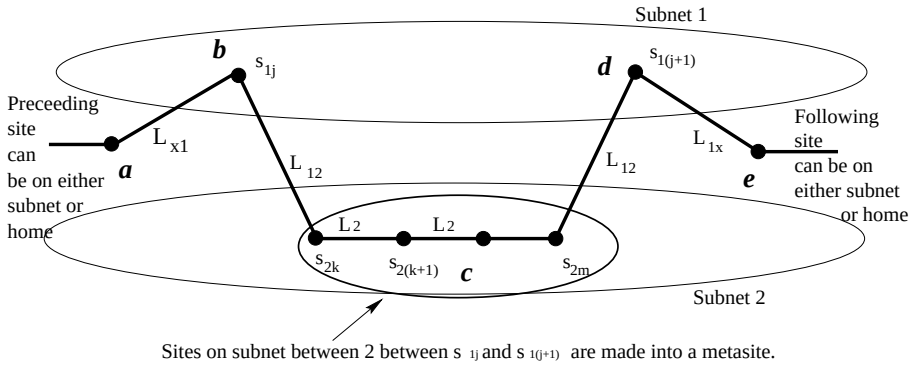

Figure 4.1: Definition of sites a, b, c, d, e.

This means that as long as either another S_1 site follows s^*_{1j} or $ms_{2k:2m}$ follows s^*_{1j}, the expected time for s^*_{1j} is constant. Similarly, as long as the site preceding $ms_{2k:2m}$ has a latency of L_1 to reach the site following it (that is, it is an S_1 site), the expected time for $ms_{2k:2m}$ is constant.

For notational simplicity, we will use the following names:

- The site preceding s^*_{1j} in S_1 is called a where an agent makes a decision whether to stay in $S-1$ or move to another subnetwork;
- Sites $s_{1j},\ ms_{2k:2m},\ s_{1(j+1)}$ are called $b,\ c,\ d$ respectively;
- The site following d is called e.

This situation is depicted in Figure 4.1.

Now suppose that the five sites, a, b, c, d, e, form part of the optimal tour. Our goal is to show that

$$\begin{aligned} p_b/t_b &= \frac{p_{1j}}{t_{1j} + p_{1j}L_{10} + (1-p_{1j})L_1} = p_{1j}/t^*_{1j} \\ &\leq p_d/t_d = \frac{p_{1(j+1)}}{t_{1(j+1)} + p_{1(j+1)}L_{10} + (1-p_{1(j+1)})L_1} = p_{1(j+1)}/t^*_{1(j+1)}. \end{aligned}$$

We do this in two steps:

1. Showing that $p_b/t_b \leq p_c/t_c$;
2. Showing that $p_c/t_c \leq p_d/t_d$.

Step 1 – Let the cost of the optimal tour with the subsequence a, b, c, d, e be C_{abcde} and consider the cost of the tour with sites b and c switched. Let the cost of the tour with b and c switched be

C_{acbde}. Then by optimality

$$\begin{aligned} 0 &\geq C_{abcdef} - C_{adcbef} \\ &= P \cdot (L_{x1} - L_{x2} + (t_b + p_b L_{10} + (1-p_b)L_{12}) + (1-p_b)(t_c + p_c L_{20} + (1-p_c)(L_{21}) \\ &\quad - (t_c + p_c L_{20} + (1-p_c)L_{21}) - (1-p_c)(t_b + p_b L_{10} + (1-p_b)L_1)) \\ &= X \geq X - P \cdot (L_{x1} - L_{x2} + L_{21} - L_1). \end{aligned}$$

Here $P > 0$ is the probability that failure has occurred at all nodes preceding b in the tour so that we in fact visit b. L_{x1} and L_{x2} are the latencies from a to S_1 and S_2 respectively since a can be either in S_1, S_2 or s_0, the home site. The other terms arise from the parts of the tours between b and d. Note that once we arrive at d, the remaining costs are identical for both tours and therefore cancel each other in the difference. The term $L_{12} - L_1$ arises as the difference in latencies in going from a to b versus a to c.

Finally, we note that $L_{x1} - L_{x2} + L_{21} - L_1$ is 0, $2L_{12} - L_1 - L_2$ or $L_{01} - L_{02} + L_{12} - L_1 = L_{12} - L_1$ depending on whether x is 1, 2 or 0 which are all nonnegative by assumptions on the latencies. This explains the last inequality.

Continuing, we have

$$\begin{aligned} 0 &\geq X - P \cdot (L_{x1} - L_{x2} + L_{12} - L_1) \\ &= P \cdot (t_b + p_b L_{10} + (1-p_b)L_1 + (1-p_b)(t_c + L_{21} - L_1 + p_c L_{20} + (1-p_c)L_{12}) \\ &\quad - (t_c + p_c L_{20} + (1-p_c)L_{21} + L_{12} - L_1) - (1-p_c)(t_b + p_b L_{10} + (1-p_b)L_1)) \\ &= P \cdot (p_c t_b - p_b t_c) \end{aligned}$$

which shows that $t_b/p_b \leq t_c/p_c$ as claimed. This shows that $C_{abcde} \leq C_{acbde}$ implies $t_b/p_b \leq t_c/p_c$.

Step 2 – We now switch c and d in an optimal tour as above, and with the same notational conventions, we have

$$\begin{aligned} 0 &\geq C_{abcde} - C_{abdce} = P \cdot (L_{12} - L_1 + t_c + p_c L_{20} + (1-p_c)L_{12} + (1-p_c)(t_d + p_d L_{10} + (1-p_d)L_{1x}) \\ &\quad - (t_d + p_d L_{10} + (1-p_d)L_{12}) - (1-p_d)(t_c + p_c L_{20} + (1-p_c)L_{2x}) \\ &= X \geq X - P \cdot (1-p_c)(1-p_d)(L_{1x} - L_{2x} + L_{12} - L_1) \\ &= P \cdot (t_c + (1-p_c)t_d - t_d - (1-p_d)t_c) \\ &= P \cdot (p_d t_c - p_c t_d) \end{aligned}$$

so that $t_c/p_c \leq t_d/p_d$. Again, $L_{1x} - L_{2x} + L_{12} - L_1$ is nonnegative as in step 1.

The two inequalities derived from Steps 1 and 2 combine to show that if *abcde* is part of the optimal tour then $t_b/p_b \leq t_c/p_c \leq t_d/p_d$ as claimed.

ASSERTION 2 – TAP of Theorem 6 can be solved in $O((n_1+1)(n_2+1))$ time using dynamic programming after the nodes are sorted as specified in Step 1 which requires $O(n_1 \log n_1 + n_2 \log n_2)$ steps.

By Assertion 1, we can assume that the nodes have been ordered into the required sequence which dictates the order in which they are visited in each subnetwork. We define a Markov Decision Problem (MDP) with states

$$\mathcal{S} = \{(i,j,k) | 0 \leq i \leq n_1, 0 \leq j \leq n_2, k = 0 \text{ or } 1\} \bigcup \{F\}$$

where state (i,j,k) means that an agent has already visited sites $s_{11}, ..., s_{1i}$ and $s_{21}, ..., s_{2j}$ and is presently on subnetwork k. The terminal state is F. The states $(0,0,1)$ and $(0,0,2)$ are the same initial state.

For instance, $(0,3,2)$ means that sites s_{21}, s_{22}, s_{23} have been visited and the agent is at subnetwork 2 while no sites from subnetwork 1 have yet been visited.

In the MDP framework, we need to describe actions for each state, corresponding transition probabilities and immediate costs. For the state $(i,j,1)$ there are two possible actions: G_1 and G_2, meaning attempts to go to the next site in subnet 1 or subnet 2. For state $(i,j,1)$ and action G_1 the next state is $(i+1,j,1)$ with probability $(1-p_{1i})$ and F with probability p_{1j}. The expected immediate cost is

$$t_{1j} + p_{1j}L_{10} + (1-p_{1j})L_1.$$

For state $(i,j,1)$ and action G_2, the next state is $(i,j+1,2)$ with probability $(1-p_{1j})$ and F with probability p_{1j}. The expected immediate cost is

$$t_{1j} + p_{1j}L_{10} + (1-p_{1j})L_{12}.$$

The same definitions apply to states $(i,j,2)$ with actions G_1 and G_2 as above with appropriate changes.

For states (n_1, j, k) the only allowable action is G_2 and for states (i, n_2, k) the only allowable actions are G_1. For state (n_1, n_2, k) there is only one action, to go to the terminal state F with appropriate costs.

This MDP has no cycles and so the optimal cost-to-go values can be computed using backtracking from the terminal node in time proportional to the total number of states which is $(n_1+1)(n_2+1)$. Q.E.D.

The above algorithm and results apply to situations with multiple subnetworks providing that internetwork latencies and latencies to the home site are larger that internetwork latencies which are constant. In that case, the algorithm requires time

$$(n_1+1)(n_2+1)...(n_m+1)$$

for m subnetworks. In the case where $n_i = 1$, this reduces to the general case of TAP which is NP-Hard and the algorithm is exponential.

We can use these results as approximation methods for general TAP's by organizing subnetworks with constant latencies that approximate the original latencies.

We have shown that TAP is NP-complete in the general formulation. However, by clustering multiple sites and approximating latencies among them to be constant, the complexity of the problem is decreased into a polynomial time computation.

The problems dealt with in this paper assumes that a single agent executes a task. In many cases, a task can be finished in a shorter time by multiple agents rather than a single agent. TAP should be extended to the case where multiple agents are involved in the same task.

4.3 Multiple Agents

Consider a planning problem where multiple agents cooperate to complete a task, for example, to search for some information. Intuitively, the total expected time should be smaller than in the single agent case. In this section, we consider the *Traveling Multiple Agent Problem* (TMAP). If the multiple agents are viewed as multiple processors and the sites to be visited by agents are the tasks to be assigned to the processors, TMAP becomes equivalent to the multiple processor scheduling problem with probabilistic success. In TMAP, we assume that the agents (processors) communicate

with each other and as soon as one of them completes the task successfully, the other agents stop execution. Formally, we define the Traveling Multiple Agent Problem (TMAP) as :

> *The Traveling Multiple Agent Problem* - There are $n+1$ sites, s_i with $0 \leq i \leq n$. Each site has a probability, $0 \leq p_i \leq 1$, of being able to successfully complete the agent's task and a computation time, $t_i > 0$, required for the agent to attempt the task at s_i regardless of whether successful or not. These probabilities are independent of each other. Travel time or latency is constant, i.e., $l = l_{ij} = l_{kq} \in Z^+$ for all sites i, j, k, q. There are m agents that cooperate each other complete a task. When one of the agents successfully completes the task, the agent must return to site 0 from which it started and at the same time, the rest of agents stop the execution of their tasks. For site 0, $p_0 = t_0 = 0$. The *Traveling Multiple Agent Problem* is to minimize the expected time to successfully complete the task.

A solution of the Traveling Multiple Agent Problem is give in a form of a multiple agent schedule as shown in Fig.4.2. In the diagram, σ_i represents the starting time of the task at site i. $t_i + l$ represents the sum of the computation time at task i and the travel time to move from the previous site to the current site.

The expected time to complete the task or visit all sites in failure for a schedule T (a set of agents' tours) is

$$C_T = (\sigma_{i_1} + l + t_{i_1}) + \sum_{k=2}^{n} \{ (\prod_{j=1}^{k-1} (1 - p_{i_j})) \cdot \{ (\sigma_{i_k} + l + t_{i_k}) - (\sigma_{i_{k-1}} + l + t_{i_{k-1}}) \} \} \qquad (4.2)$$

where i_k is the kth task to finish according to the schedule T (refer to Fig.4.2).

Please note that σ_{i_j} is equivalent to

$$\sum_{k \in A_j} t_k$$

where A_j is a set of sites visited before the site i_j by the same agent that visits the site i_j.

Unfortunately, this Traveling Multiple Agent Problem is NP-complete. The proof of TMAP's NP-completeness is shown as follows:

Theorem 7 *The Traveling Multiple Agent Problem (TMAP) is NP-complete*

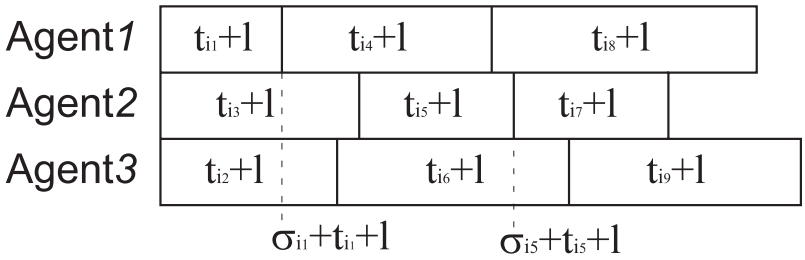

Figure 4.2: Schedule T for multiple agents

Proof – First, we show that TMAP belongs to NP. Given the m-agent schedule T as a certificate, the verification checks if the schedule T satisfies the deadline D by calculating C_T in (4.2) and comparing its value with D. This verification can be performed in polynomial time. Thus, this problem belongs to NP.

Next, we show TMAP belongs to NP-hard by proving that PARTITION [31] problem can be reduced to TMAP. Let the finite set of sites be A and a given size s_a for each $a \in A$ be an instance of PARTITION. Let $B = \sum_{a \in A} s_a$. Then, we let each site $a \in A$ with its size s_a correspond to a site a' in TMAP with its time $t_{a'} = s_a$ and $p_{a'} = 0$. Let $m = 2$ be the number of agents. Note that $t_{a'}$ includes both the constant latency and the computation time at the site a'. This TMAP instance can be easily constructed in polynomial time from the PARTITION instance.

To complete the proof, we show that there is a subset $A" \subseteq A$ for PARTITION such that

$$\sum_{a \in A"} s_a = \sum_{a \in A - A"} s_a = \frac{B}{2} \tag{4.3}$$

if and only if there is a schedule T that meets the overall deadline $D = B/2$.

Suppose that all sites a which correspond to $a \in A"$ are assigned to the first agent and the rest of the sites a' for $a' \in A - A"$ are assigned to the second agent. If the site number is given in increasing order of finishing time of a task at the site, the expected overall length C is calculated as:

$$\begin{aligned} C &= (\sigma_2 - \sigma_1) + \sum_{i=2}^{|A|} (\sigma_{i+1} - \sigma_i) \\ &= \sigma_{|A|+1} - \sigma_1 = \sigma_{|A|+1} \end{aligned}$$

where $\sigma_1 = 0$.

$\sigma_{|A|+1}$ is the time when the agent leaves the last site and it can be calculated by taking the largest value in the overall time for each agent, i.e.,

$$\sigma_{|A|+1} = \max\{\sum_{i \in A} t_i, \sum_{j \in A - A"} t_j\}$$

Since $t_a = s_a$ for $a \in A$, when (4.3) is satisfied, the schedule T that can meet the overall deadline $D = B/2 = \sigma_{|A|+1}$. On the other side, if there is a schedule T that can meet the overall deadline $D = B/2$, it is easy to observe

$$\sum_{i \in A"} t_i = \sum_{j \in A - A"} t_j = \frac{B}{2}$$

from the argument leading to (4.3).

Thus, the desired subset $A"$ exists for the instance PARTITION if and only if there is a schedule T for TMAP that meets the overall deadline $D = B/2$. Q.E.D.

We now explore optimal solutions for TMAP when we impose some assumptions. Ideally, we seek realistic assumptions that reduce the complexity of the problem.

4.3.1 Probability of Success = 0

Assumption 3 *Probabilities of success are all zero.*

Theorem 8 *Under the Assumption 3, the Traveling Multiple Agent Problem is exactly the same as the Multiple Processor Scheduling Problem[25], and the optimal solution can be obtained using dynamic programming in pseudo-polynomial time for a fixed number of agents. Pseudo-polynomial means polynomial in the values of input time, not the number of bits required to represent them.*

The proof is given as follows:

Proof – Suppose that the probability of success is $p_i = 0$ for site i, namely, Assumption 3 and that the number of agents m is fixed. Let n be the number of sites and $B = \sum_{i=1}^{n} t_i$ where t_i is the sum of the latency to move from the previous site and the computation time at the current site i. For integer $1 \leq i \leq n$ and $1 \leq j_k \leq B$ where $1 \leq k \leq m-1$, let $e(i,\ j_1,\ j_2, ...,\ j_{m-1})$ denote the function which is true if there is a partition $\{A_1,\ A_2, ...,\ A_{m-1}\}$ of the set of sites $< 1,\ 2, ...,\ i >$ that consists

of disjoint subsets $A_1,\ A_2, ...,\ A_{m-1}$ whose total tour times are $j_1,\ j_2, ...,\ j_{m-1}$, respectively. Note that the order of a site in the tour at each subset does not affect the total time because $p_i = 0$ for all i.

The value of $e(i,\ j_1,\ j_2, ...,\ j_{m-1})$ is obtained easily by dynamic programming. The number of combinations of possible parameter values in the function e are $n \times B^{m-1}$. The value of $e(i,\ j_1,\ j_2, ...,\ j_{m-1})$ is computed for each combination. We start with $i = 1$ and then increase the value of i. For $i = 1$, $e(i,\ j_1,\ j_2, ...,\ j_{m-1})$ is true if and only if either $j_1 = j_2 = ... = j_{m-1} = 0$ or only one j_k for $1 \leq k \leq m-1$ equals t_1 and the rest of j_k's are all 0. For $i \geq 2$, the value of $e(i,\ j_1,\ j_2, ...,\ j_{m-1})$ is true if and only if either $e(i-1,\ j_1,\ j_2, ...,\ j_{m-1})$ is true or $t_i \leq j_k$ and $e(i-1,\ j_1,\ j_2, ...,\ j_k - t_i, ...,\ j_{m-1})$ is true for some $1 \leq k \leq m-1$. It is easy to see that the calculation of values for each possible combination of parameters can be performed in polynomial time, i.e., $n \times B^{m-1}$ in terms of the number of sites (n). Recall that $B = \sum_{i=0}^{n} t_i$.

The subset A_k corresponds to the set of sites assigned to the agent k. The sites assigned to agent m are the sites that were not assigned to any agent k for $1\ \leq k\ \leq m-1$. The maximum time of the agent tours is the overall time of the schedule for TMAP. During the calculation of the values of $e(i, j_1,\ j_2, .., j_{m-1})$, the overall time can be obtained and it can be verified whether the overall time does not exceed the deadline D. This verification can be done in polynomial time. Thus, TMAP can be solved in polynomial time, provided it satisfies the following condition. The condition is that the sum of time spent at all the sites, B, and the number of agents, m, should be bounded in advance. Because of this condition, this problem is called as a pseudo-polynomial time problem instead of a polynomial time problem. Q.E.D.

If probabilities of success are unknown, it is reasonable that the probability at each site is set to be 0.

4.3.2 Probability of Success = 1

Next, we assume that all probabilities of success are 1, which means that the mobile agent can certainly complete its task at the first site to visit. The problem becomes a sorting problem which can be solved in polynomial time even for an arbitrary number of agents m.

Theorem 9 *If a probability of success p_i is 1 for all sites i, this problem can be solved in polynomial time even for arbitrary m.*

Proof – Suppose that $p_i = 1$ for all the sites i. Then, a task is completed when the agent leaves the first site. There is no need of scheduling for the rest of the sites. Hence, scheduling for the TMAP is to find a task i^* such as $t_{i^*} = \min_{i=0}^{n} t_i$ where n is the number of sites and t_i is the computation time at site i. This process can be done easily in a polynomial time. Q.E.D.

4.3.3 Constant Probability ≥ 0.5

The complexity of the original TMAP problem can be reduced to polynomial time under the assumption that the probabilities of success are all the same and are none less than 0.5. This assumption is not unrealistic. For example, if our only knowledge about sites is that they all have relatively large probabilities, we might be able to assume that the probabilities of all the sites are same and are larger than or equal to 0.5.

Accordingly, the assumption that we employ here is:

Assumption 4 *Probabilities of success p_i are constant and $p_i = p \geq 0.5$ for all i, while the sum of latency and computation time, i.e., $t_i + l$ can be arbitrary. There are m agents that cooperate with each other to complete a task.*

Under this assumption, it turns out that TMAP can be solved in polynomial time as shown below. We will start by proving a preliminary theorem (Theorem 10) that states that the sites visited by the same agent should be sorted in increasing order of $p/(t_i + l)$. Then, we prove a main theorem (Theorem 11) which states that the optimal schedule of TMAP under the above assumption can be obtained in polynomial-time.

Theorem 10 *Under Assumption 4, the optimal sequence of sites visited by the same agent is obtained by sorting the sites in increasing order of $p/(t_i + l)$, i.e., in decreasing order of $t_i + l$.*

Proof – The proof begins with obtaining the formula for the total expected cost of a schedule for an m agent problem, which is used in the rest of the proof.

The expected total time for an m-agent scheduling S can be obtained by setting $p_i = p$ in (4.2), and it is then given by:

$$C_S = (\sigma_1 + t_1 + l) + \sum_{i=2}^{n} \{ (\prod_{j=1}^{i-1} (1-p))(\sigma_i + t_i + l - \sigma_{i-1} - t_{i-1} - l) \} \tag{4.4}$$

where site i is the ith site in the list of sites sorted in the increasing order of its task's finishing time. σ_i is a time when a task starts at site i, and n is the total number of sites. $\sigma_i + t_i + l - \sigma_{i-1} - t_{i-1} - l$ stands for the time difference between the completion times of a task at site $i-1$ and at site i.

The above equation (4.4) simplifies as:

$$\begin{aligned} C_S &= (1-(1-p)) \cdot (\sigma_1 + t_1 + l) + \sum_{i=2}^{n-1} (\prod_{j=1}^{i-1} (1-p)) \cdot (1-(1-p)) \cdot (\sigma_i + t_i + l) \\ &\quad + (\prod_{j=1}^{n-1} (1-p))(\sigma_n + t_n + l) \\ &= p \cdot (t_1 + l) + \sum_{i=2}^{n-1} (1-p)^{i-1} \cdot p \cdot (\sigma_i + t_i + l) + (1-p)^{n-1} (\sigma_n + t_n + l) \end{aligned}$$

Please note that σ_i is equal to $\sum_{k \in T_i} (t_k + l)$ where T_i is a set of sites visited by the same agent that visits site i before site i.

Next, we show that the optimal schedule of sites visited by a single agent is obtained by sorting sites in decreasing order of $t_i + l$.

Suppose that there is an m-agent schedule, $S =< s_1, s_2, ..., s_n >$, where the sites are in increasing order of the agent's leaving time at each site as shown in Fig.4.3. Thus, $s_i < s_k$ if $\sigma_i + t_i \leq \sigma_k + t_k$. Now, we exchange the order of two adjacent sites s_i and s_{i+k}, both of which are visited by the agent 1. Assume that $t_i < t_{i+k}$. Note that the total expected time for the original schedule S is:

$$C_S = p \cdot (t_1 + l) + \sum_{i=2}^{n-1} (1-p)^{i-1} \cdot p \cdot (\sigma_i + t_i + l) + (1-p)^{n-1} (\sigma_n + t_n + l). \tag{4.5}$$

Swapping the two sites may affect the order of finishing time of the rest of the sites. Because $t_i < t_{i+k}$, some sites between the sites s_i and s_{i+k} in the original schedule S may finish before the swapped site t_{i+k} in the new schedule S'. For example, assume that we swap two sites s_i and s_{i+k}

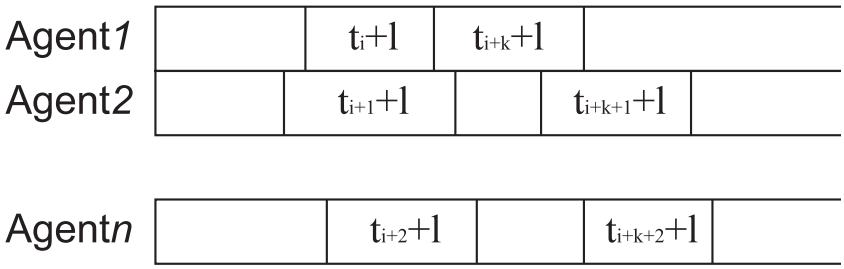

Figure 4.3: A schedule for the m agent problem

in the original schedule S where the sites are sorted in increasing order of the finishing time as

$$s_1, s_2, .., \mathbf{s_i}, .., \mathbf{s_{i+k}}, ..., s_n$$

. Then the new schedule S' is

$$s_1, s_2, ..., s_{i-1}, s_{i+1}, ..., s_{i+h}, \mathbf{s_{i+k}}, s_{i+h+1}, ..., s_{i+k-1}, \mathbf{s_i}, s_{i+k+1}, ..., s_n$$

where $0 < h < k$. These sites are sorted in increasing order of finishing time. Thus the order of the unswapped sites from s_{i+1} to s_{i+h} is shifted earlier by one.

If swapping s_i and s_{i+k} affects the order of the unswapped sites, as in the above case, the total expected time of the new schedule S' is:

$$\begin{aligned} C_{S'} &= p \cdot (t_1 + l) + (\sum_{j=2}^{i-1} (1-p)^{j-1} \cdot p \cdot (\sigma_j + t_j + l)) + (1-p)^{i-1} \cdot p \cdot (\sigma_{i+1} + t_{i+1} + l) \\ &+ .... + (1-p)^{i+h-2} \cdot p \cdot (\sigma_{i+h} + t_{i+h} + l) + (\mathbf{1-p})^{\mathbf{i+h-1}} \cdot \mathbf{p} \cdot (\sigma_{\mathbf{i}} + \mathbf{t_{i+k}} + \mathbf{l}) \\ &+ (1-p)^{i+h} \cdot p \cdot (\sigma_{i+h+1} + t_{i+h+1} + l) + ... + (1-p)^{i+k-2} \cdot p \cdot (\sigma_{i+k-1} + t_{i+k-1} + l) \\ &+ (\mathbf{1-p})^{\mathbf{i+k-1}} \cdot \mathbf{p} \cdot (\sigma_{\mathbf{i+k}} + \mathbf{t_i} + \mathbf{l}) + (1-p)^{i+k} \cdot p \cdot (\sigma_{i+k+1} + t_{i+k+1} + l) \\ &+ ... + (1-p)^{n-1} \cdot p \cdot (\sigma_n + t_n + l). \end{aligned}$$

.

Otherwise, the total expected time of the new schedule S' is:

$$C_{S'} = p \cdot (t_1 + l) + (\sum_{j=2}^{i-1} (1-p)^{j-1} \cdot p \cdot (\sigma_j + t_j + l)) + (\mathbf{1-p})^{\mathbf{i+h-1}} \cdot \mathbf{p} \cdot (\sigma_{\mathbf{i}} + \mathbf{t_{i+k}} + \mathbf{l})$$

$$+(1-p)^{i+h} \cdot p \cdot (\sigma_{i+1} + t_{i+1} + l)$$

$$+... + (1-p)^{i+k-2} \cdot p \cdot (\sigma_{i+k-1} + t_{i+k-1} + l)$$

$$+(\mathbf{1-p})^{\mathbf{i+k-1}} \cdot \mathbf{p} \cdot (\sigma_{\mathbf{i+k}} + \mathbf{t_i} + \mathbf{l})$$

$$+(1-p)^{i+k} \cdot p \cdot (\sigma_{i+k+1} + t_{i+k+1} + l)$$

$$+... + (1-p)^{n-1} \cdot p \cdot (\sigma_n + t_n + l).$$

The difference between schedules S and S' is:

- In the case where the order of unswapped sites is changed:

$$\begin{aligned} C_S - C_{S'} &= (1-p)^{i-1} \cdot p \cdot (\sigma_i + t_i - \sigma_{i+1} - t_{i+1}) \\ &\quad +(1-p)^{i} \cdot p \cdot (\sigma_{i+1} + t_{i+1} - \sigma_{i+2} - t_{i+2}) + .. + \\ &\quad (1-p)^{i+h-2} \cdot p \cdot (\sigma_{i+h-1} + t_{i+h-1} - \sigma_{i+h} - t_{i+h}) \\ &\quad +(1-p)^{i+h-1} \cdot p \cdot (\sigma_{i+h} + t_{i+h} - \sigma i - t_{i+k}) \\ &\quad +(1-p)^{i+k-1} \cdot p \cdot (\sigma_{i+k} + t_{i+k} - \sigma_{i+k} - t_i); \end{aligned} \tag{4.6}$$

- In the case where the order of unswapped sites is not changed:

$$\begin{aligned} C_S - C_{S'} &= (1-p)^{i-1} \cdot p \cdot (\sigma_i + t_i - \sigma_i - t_{i+k}) \\ &\quad (1-p)^{i+k-1} \cdot p \cdot (\sigma_{i+k} + t_{i+k} - \sigma_{i+k} - t_i) \\ &= (1-p)^{i-1} \cdot p \cdot (t_i - t_{i+k}) + (1-p)^{i+k-1} \cdot p \cdot (t_{i+k} - t_i) \end{aligned} \tag{4.7}$$

Note that all the terms in the first case are non-positive except for the last one. In order to prove that the sum of all the terms are non-positive. If we increase the power of $(1-p)$ in each of non-positive terms, which increases each value of these terms. Then the difference $C_S - C_{S'}$ in the first case can be expanded as follows:

$$\begin{aligned} C_S - C_{S'} &\leq (1-p)^{i+h-1} \cdot p \cdot (\sigma_i + t_i - \sigma_{i+1} - t_{i+1}) \\ &\quad +(1-p)^{i+h-1} \cdot p \cdot (\sigma_{i+1} + t_{i+1} - \sigma_{i+2} - t_{i+2}) + .. + \\ &\quad (1-p)^{i+h-1} \cdot p \cdot (\sigma_{i+h-1} + t_{i+h-1} - \sigma_{i+h} - t_{i+h}) \\ &\quad +(1-p)^{i+h-1} \cdot p \cdot (\sigma_{i+h} + t_{i+h} - \sigma i - t_{i+k}) \end{aligned}$$

$$
\begin{aligned}
& +(1-p)^{i+k-1} \cdot p \cdot (\sigma_{i+k} + t_{i+k} - \sigma_{i+k} - t_i) \\
= \quad & (1-p)^{i+h-1} \cdot p \cdot (t_i - t_{i+k}) \\
& +(1-p)^{i+k-1} \cdot p \cdot (t_{i+k} - t_i) \qquad (4.8)
\end{aligned}
$$

Since in the above $t_i - t_{i+k}$ is non-positive and $(1-p)^{i+h-1} \cdot p > (1-p)^{i+k-1} \cdot p$, $C_S - C_{S'}$ in the first case is non-positive. In the same way, $C_S - C_{S'}$ in the second case is non-positive, too. Therefore, the original schedule S has a smaller total expected time.

Before we reach a conclusion, we have to consider the case where the site s_{i+k} in the above argument is the last site, i.e., s_n because the last term in the form of the expected time for a schedule is different from the rest of the terms. Setting $i+k=n$ requires minor changes in C_S and C'_S. For example, the form of C_S has to be changed to:

$$
C_S = p \cdot (t_1 + l) + \sum_{i=2}^{i+k-1} (1-p)^{i-1} \cdot p \cdot (\sigma_i + t_i + l) + (1-p)^{i+k-1} (\sigma_n + t_{i+k} + l).
$$

and the form of $C_{S'}$ in the first case has to be changed to:

$$
\begin{aligned}
C_{S'} \quad = \quad & p \cdot (t_1 + l) + (\sum_{j=2}^{i-1} (1-p)^{j-1} \cdot p \cdot (\sigma_j + t_j + l)) + (1-p)^{i-1} \cdot p \cdot (\sigma_{i+1} + t_{i+1} + l) \\
& +.... + (1-p)^{i+h-2} \cdot p \cdot (\sigma_{i+h} + t_{i+h} + l) + \mathbf{(1-p)^{i+h-1} \cdot p \cdot (\sigma_i + t_{i+k} + l)} \\
& +(1-p)^{i+h} \cdot p \cdot (\sigma_{i+h+1} + t_{i+h+1} + l) + ... + (1-p)^{i+k-2} \cdot p \cdot (\sigma_{i+k-1} + t_{i+k-1} + l) \\
& +\mathbf{(1-p)^{i+k-1} \cdot (\sigma_{i+k} + t_i + l)}
\end{aligned}
$$

When $i+k=n$, it should not be difficult to understand that the form (4.6) (in the first case) becomes:

$$
\begin{aligned}
C_S - C_{S'} \quad \leq \quad & (1-p)^{i+h-1} \cdot p \cdot (t_i - t_{i+k}) \\
& +(1-p)^{i+k-1} \cdot (t_{i+k} - t_i) \\
\leq \quad & (1-p)^{i+k-2} \cdot (p - (1-p)) \cdot (t_i - t_{i+k}) \\
= \quad & (1-p)^{i+k-2} \cdot (2p-1) \cdot (t_i - t_{i+k})
\end{aligned}
$$

, which is non-positive if $p \leq 0.5$. In the same way, the value of the modified version of the form (4.7) (in the second case) can be derived to be non-positive.

Thus, $C_S - C_{S'}$ is non-positive if $p \geq 0.5$, which implies that sites visited by the same agent should be sorted in increasing order of $t_i + l$. Q.E.D.

The result of Theorem 10 is necessary as a preliminary theorem in the proof of the following theorem, Theorem 11.

Theorem 11 *Assume that in TMAP* $l = l_{ij} = l_{kq} \geq 0$ *for all* i, j, k, q, *namely Assumption 3. Probabilities* $p_i = p \geq 0.5$ *for all* i, *while computation times* t_i *can be arbitrary (Assumption 4). There is a sorted list of sites in increasing order of* $t_i + l$. *The sites in the list is assigned to* m *agents where* $m > 1$. *The optimal schedule for the* m *agent TMAP is attained by the greedy algorithm based on* $t_i + l$ *(i.e., by assigning a first unassigned site in the list to the agent that becomes available first).*

Proof – This proof consists of 4 steps. In the first 3 steps, we arbitrarily choose 2 agents from the m agents and consider the optimal schedule of sites visited by these 2 agents. The last step expands the result from the previous step into the m-agent optimal schedule.

The summary of each step is as follows: The first step shows that the last two sites in the optimal 2-agent schedule should be visited in increasing order of $t_i + l$ by different agents. The second step shows that the third site from the last in the optimal schedule has to be visited just before the second last site by a different agent assuming the last two sites are optimally scheduled. In order to use the induction reasoning to prove this Theorem 11, the third step shows that assuming the last $k-1$ sites are optimally scheduled by the greedy method (i.e., they are visited in increasing order of $t_i + l$ by different agents alternatively), the kth site from the last should be visited just before the lst $k-1$th site by a different agent. The last step generalizes the 2-agent optimal schedule into m-agent optimal schedule.

Step 1 – We show that the last two sites in the sorted list should be visited in the same order by different agents. This proof consists of two steps. The first step shows that the last two sites has to be visited in a same order of the sorted list if they can be visited in sequence by different agents. The second step shows that the last two sites have to be visited in sequence by different agents.

There are three possible configurations for the last two sites if an agent can leave these sites in sequence, as shown in Fig.4.4. The sites in the first and second rows in the schedules are visited by agent1 and agent2, respectively. Sites in the last row are visited by the rest of the agents, i.e., $m-2$

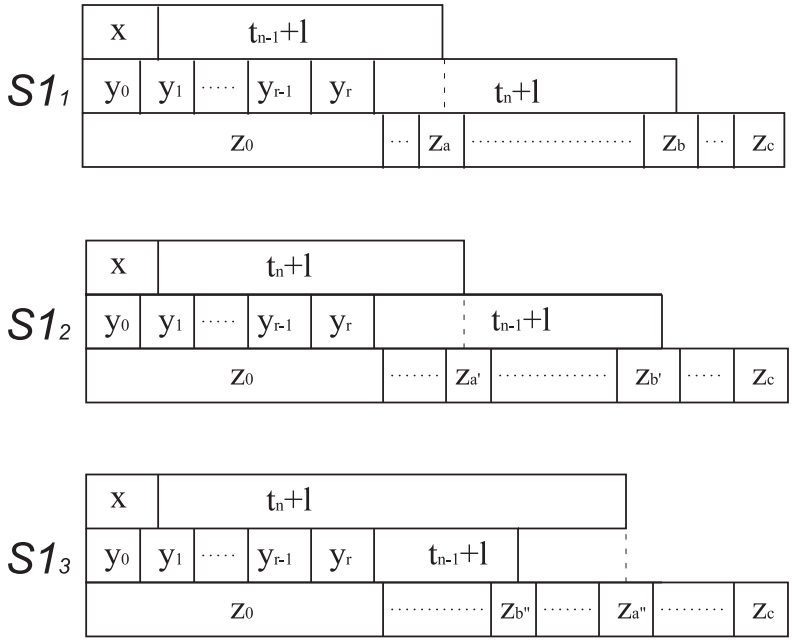

Figure 4.4: The optimal assignment of the last two sites

agents. Sizes of y_i and z_i vary, while $t_i > t_j$ if $i > j$. x and y_0 stand for the total time of sites visited by each agent before site S_{n-1} is visited. In the second case of Fig.4.4, $x_0 + t_n + l < \sum_{i=0}^{r} y_i + t_{n-1} + l$, while in the third case of Fig.4.4, $x_0 + t_n + l \geq \sum_{i=0}^{r} y_i + t_{n-1} + l$.

Assume that $\sum_{i=0}^{r} y_i - x < t_{n-1} + l$. The total costs of the two configurations are as follows:

- Case 1

$$\begin{aligned} C_{S1_1} &= py_0 + p(1-p)x + p(1-p)^2(y_0+y_1) + p(1-p)^3(y_0+y_1+y_2) \\ &\quad + \ldots + p(1-p)^{r+1}(\sum_{i=0}^{r} y_i) + p(1-p)^{r+2} z_0 \\ &\quad + \ldots + p(1-p)^{r+a+1}(\sum_{i=0}^{a-1} z_i) + p(1-p)^{r+a+2}(x + t_{n-1} + l) \\ &\quad + p(1-p)^{r+a+3}(\sum_{i=0}^{a} z_i) + \ldots + p(1-p)^{r+b+2}(\sum_{i=0}^{b-1} z_i) \\ &\quad + p(1-p)^{r+b+3}(\sum_{i=0}^{r} y_i + t_n + l) + p(1-p)^{r+b+4}(\sum_{i=0}^{b} z_i) \end{aligned}$$

$$+... + p(1-p)^{r+c+3}(\sum_{i=0}^{c-1} z_i)$$

$$+(1-p)^{r+c+4}(\sum_{i=0}^{c} z_i)$$

- Case 2

$$\begin{aligned} C_{S1_2} &= py_0 + p(1-p)x + p(1-p)^2(y_0+y_1) + p(1-p)^3(y_0+y_1+y_2) \\ &+... + p(1-p)^{r+1}(\sum_{i=0}^{r} y_i) + p(1-p)^{r+2}z_0 \\ &+... + p(1-p)^{r+a'+1}(\sum_{i=0}^{a'-1} z_i) + p(1-p)^{r+a'+2}(x+t_n+l) \\ &+p(1-p)^{r+a'+3}(\sum_{i=0}^{a'} z_i) + ... + p(1-p)^{r+b'+2}(\sum_{i=0}^{b'-1} z_i) \\ &+p(1-p)^{r+b'+3}(\sum_{i=0}^{r} y_i + t_n + l) + p(1-p)^{r+b'+4}(\sum_{i=0}^{b'} z_i) \\ &+... + p(1-p)^{r+c+3}(\sum_{i=0}^{c-1} z_i) \\ &+(1-p)^{r+c+4}(\sum_{i=0}^{c} z_i) \end{aligned}$$

- Case 3

$$\begin{aligned} C_{S1_3} &= py_0 + p(1-p)x + p(1-p)^2(y_0+y_1) + p(1-p)^3(y_0+y_1+y_2) \\ &+... + p(1-p)^{r+1}(\sum_{i=0}^{r} y_i) + p(1-p)^{r+2}z_0 \\ &+... + p(1-p)^{r+a''+1}(\sum_{i=0}^{a''-1} z_i) + p(1-p)^{r+a''+2}(x+t_{n-1}+l) \\ &+p(1-p)^{r+a''+3}(\sum_{i=0}^{a''} z_i) + ... + p(1-p)^{r+b''+2}(\sum_{i=0}^{b''-1} z_i) \\ &+p(1-p)^{r+b''+3}(\sum_{i=0}^{r} y_i + t_n + l) + p(1-p)^{r+b''+4}(\sum_{i=0}^{b''} z_i) \end{aligned}$$

$$+\ldots + p(1-p)^{r+c+3}(\sum_{i=0}^{c-1} z_i)$$

$$+(1-p)^{r+c+4}(\sum_{i=0}^{c} z_i)$$

Note that because $t_n > t_{n-1}$ $a' \geq a$ and $b' \leq b$ in the second case, and $a'' \geq a$ and $b'' \leq b$ in the third case.

The difference of C_{S1_1} and $C1_{S_2}$ is:

$$\begin{aligned}
C_{S1_1} - C_{S1_2} &= p(1-p)^{r+a+2}\{(x + t_{n-1} + l) - (\sum_{i=0}^{a} z_i)\} \\
&\quad +(1-p)^{r+a+3}\{(\sum_{i=0}^{a} z_i) - (\sum_{i=0}^{a+1} z_i)\} + \ldots + (1-p)^{r+a'+2}\{(\sum_{i=0}^{a'-1} z_i) - (\sum_{i=0}^{a'} z_i)\} \\
&\quad +p(1-p)^{r+a'+3}\{(\sum_{i=0}^{a'} z_i) - (x + t_n + l)\} \\
&\quad +p(1-p)^{r+b'+3}\{(\sum_{i=0}^{b'} z_i) - (\sum_{i=0}^{r} y_i + t_{n-1} + l)\} \\
&\quad +p(1-p)^{r+b'+4}\{(\sum_{i=0}^{b'+1} z_i) - (\sum_{i=0}^{b'} z_i)\} + \ldots + p(1-p)^{r+b+2}\{(\sum_{i=0}^{b-1} z_i) - (\sum_{i=0}^{b-2} z_i)\} \\
&\quad +p(1-p)^{r+b+3}\{(\sum_{i=0}^{r} y_i) + t_n + l) - (\sum_{i=0}^{b-1} z_i)\} \\
&= p(1-p)^{r+a+2}(x + t_{n-1} + l - \sum_{i=0}^{a} z_i) \\
&\quad +(1-p)^{r+a+3}(-z_{a+1}) + \ldots + (1-p)^{r+a'+2}(-z_{a'}) \\
&\quad +p(1-p)^{r+a'+3}(\sum_{i=0}^{a'} z_i - x - t_n - l) \\
&\quad +p(1-p)^{r+b'+3}(\sum_{i=0}^{b'} z_i - \sum_{i=0}^{r} y_i - t_{n-1} - l) \\
&\quad +p(1-p)^{r+b'+4}(z_{b'+1}) + \ldots + p(1-p)^{r+b+2}(z_{b-1})
\end{aligned}$$

$$+p(1-p)^{r+b+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{b-1} z_i)$$

(4.9)

Using the same technique to derive (4.8) in the proof of Theorem 10, the above difference $C_{S1_1} - C_{S1_2}$, i.e., (4.9) continues as:

$$\begin{aligned}
C_{S1_1} - C_{S1_2} \quad \leq \quad & p(1-p)^{r+a'+3}(x + t_{n-1} + l - \sum_{i=0}^{a} z_i) \\
& +p(1-p)^{r+a'+3}(-z_{a+1}) + ... + (1-p)^{r+a'+3}(-z_{a'}) \\
& +p(1-p)^{r+a'+3}(\sum_{i=0}^{a'} z_i - \sum_{i=0}^{r} y_i - t_n - l) \\
& +p(1-p)^{r+b'+3}(\sum_{i=0}^{b'} z_i - x - t_{n-1} - l) \\
& +p(1-p)^{r+b'+3}(z_{b'+1}) + ... + p(1-p)^{r+b'+3}(z_{b-1}) \\
& +p(1-p)^{r+b'+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{b-1} z_i) \\
= \quad & p(1-p)^{r+a'+3}(t_{n-1} - t_n) + p(1-p)^{r+b'+3}(t_n - t_{n-1}) \qquad (4.10)
\end{aligned}$$

(4.10) is non-positive because $t_n \geq t_{n-1}$ and $p(1-p)^{r+a'+3} \geq p(1-p)^{r+b'+3}$. Thus, the difference $C_{S1_1} - C_{S1-2}$ is non-positive.

In the same way, the difference $C_{S1_1} - C_{S1-3}$ is

$$\begin{aligned}
C_{S1_1} - C_{S1_3} \quad = \quad & p(1-p)^{r+a+2}(x + t_{n-1} + l - \sum_{i=0}^{a} z_i) \\
& +(1-p)^{r+a+3}(-z_{a+1}) + ... + (1-p)^{r+a''+2}(-z_{a''}) \\
& +p(1-p)^{r+a''+3}(\sum_{i=0}^{a''} z_i - \sum_{i=0}^{r} y_i - t_{n-1} - l) \\
& +p(1-p)^{r+b''+3}(\sum_{i=0}^{b''} z_i - x - t_n - l) \\
& +p(1-p)^{r+b''+4}(z_{b''+1}) + ... + p(1-p)^{r+b+2}(z_{b-1})
\end{aligned}$$

$$+p(1-p)^{r+b+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{b-1} z_i)$$

$$\leq \quad p(1-p)^{r+a''+3}(x - \sum_{i=0}^{r} y_i) + p(1-p)^{r+b''+3}(\sum_{i=0}^{r} y_i - x) \tag{4.11}$$

(4.11) is also non-positive because $\sum_{i=0}^{r} y_i \geq x$ and $p(1-p)^{r+a'+3} \geq p(1-p)^{r+b'+3}$. Thus, the difference $C_{S1_1} - C_{S1-3}$ is non-positive.

Because both of (4.10) and (4.11) are non-positive, the first schedule $S1_1$ has the smallest expected time.

In the above, we assume that the total time in the last row, i.e., $\sum_{i=0}^{c} z_i$ is larger than the total time in both of the first and second rows. Either site s_n or s_{n-1} was not the last site in the tour. However, if either one is the last site, the expected time of a schedule changes a little, although the result (C_{S_1} is the best schedule) remains same, as shown below.

If a task in both sites s_n and s_{n-1} finishes earlier than $\sum_{i=0}^{c} z_i$ in the above three schedules in Fig.4.4, i.e., $c = 0$, the difference of the expected time of schedules $C_{S1_1} - C_{S1-2}$ and $C_{S1_1} - C_{S1-3}$ become as follows:

$$\begin{aligned} C_{S1_1} - C_{S1_2} &= p(1-p)^{r+3}(t_{n-1} - t_n) + (1-p)^{r+4}(t_n - t_{n-1}) \\ &= (1-p)^{r+3}(2p-1)(t_{n-1} - t_n) \end{aligned} \tag{4.12}$$

and

$$\begin{aligned} C_{S1_1} - C_{S1_3} &= p(1-p)^{r+3}(x - \sum_{i=0}^{r} y_i) + (1-p)^{r+4}(\sum_{i=0}^{r} y_i - x) \\ &= (1-p)^{r+3}(2p-1)(x - \sum_{i=0}^{r} y_i) \end{aligned} \tag{4.13}$$

Thus, both (4.12) and (4.13) are non-positive if $p \geq> 0.5$.

If a task only one of sites s_n and s_{n-1} finishes earlier than $\sum_{i=0}^{c} z_i$ in Fig.4.4, i.e., if $x_0 + t_n + l < \sum_{i=0}^{c} z_i < \sum_{i=0}^{r} y_i + t_{n-1} + l$ or if $\sum_{i=0}^{r} y_i + t_{n-1} + l < \sum_{i=0}^{c} z_i < x_0 + t_n + l$, the situation changes a little. It is not difficult to understand that the differences $C_{S1_1} - C_{S1_2}$ and $C_{S1_1} - C_{S1_3}$ becomes

$$C_{S1_1} - C_{S1_2} \quad = \quad p(1-p)^{r+a+2}(x + t_{n-1} + l - \sum_{i=0}^{a} z_i)$$

$$
\begin{aligned}
& +(1-p)^{r+a+3}(-z_{a+1}) + \ldots + (1-p)^{r+a'+2}(-z_{a'}) \\
& +p(1-p)^{r+a'+3}(\sum_{i=0}^{a'} z_i - x - t_n - l) \\
& +(1-p)^{r+c+3}(t_n - t_{n-1}) \\
\leq\ & p(1-p)^{r+a'+3}(t_{n-1} - t_n) + (1-p)^{r+c+3}(t_n - t_{n-1}) \\
=\ & (1-p)^{r+a'+3}(p - (1-p)^{c-a'})(t_{n-1} - t_n)
\end{aligned}
$$

and

$$
\begin{aligned}
C_{S1_1} - C_{S1_3} =\ & p(1-p)^{r+a+2}(x + t_{n-1} + l - \sum_{i=0}^{a} z_i) \\
& +(1-p)^{r+a+3}(-z_{a+1}) + \ldots + (1-p)^{r+a'+2}(-z_{a'}) \\
& +p(1-p)^{r+a'+3}(\sum_{i=0}^{a'} z_i - \sum_{i=0}^{r} y_i - t_{n-1} - l) \\
& +(1-p)^{r+c+3}(\sum_{i=0}^{r} y_i - x) \\
\leq\ & p(1-p)^{r+a'+3}(x - \sum_{i=0}^{r} y_i) + (1-p)^{r+c+3}(\sum_{i=0}^{r} y_i - x) \\
=\ & (1-p)^{r+a'+3}(p - (1-p)^{c-a'})(x - \sum_{i=0}^{r} y_i)
\end{aligned}
$$

Thus, if $p \geq 0.5$, both of the above differences are non-positive.

As a summary, if $p \geq 0.5$, schedule C_{S_1} has the smallest expected time, which implies that t_n has to be visited later than t_{n-1}.

In the above, we have assumed that $\sum_{i=0}^{r} y_i < x + t_{n-1} + l$ according to Fig.4.4. There were no other sites in the agent 2's tour whose tasks are finished between site s_{n-1} and s_n.

Now, we will take a look at the case where $\sum_{i=0}^{r} y_i > x + t_{n-1} + l$ where a task of some sites in the agent 2's tour can be finished between site s_{n-1} and s_n. Then we show that tasks of the last two sites in the optimal schedule have the largest computation time (i.e., they have to be t_n and t_{n-1}), and these sites have to be visited by different agents in increasing order of computation time (i.e., in the order of s_{n-1} and s_n).

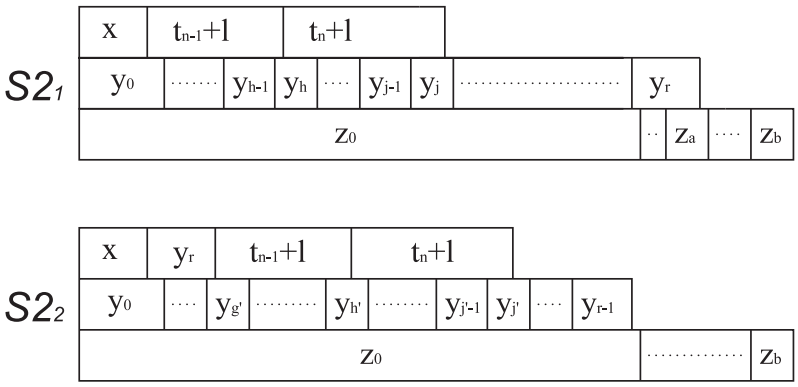

Figure 4.5: The optimal assignment of the last two sites (2)

In the following three cases, we assume that there are $r+1$ sites visited by agent 2 before site n, as shown in Fig.4.4, Fig.4.5 and Fig.4.6. In the first case (Fig.4.5), $\sum_0^r y_i > x + t_{n-1} + t_n + 2l$. In the second case (Fig.4.6), $x + t_{n-1} + l < (\sum_{i=0}^r y_i) < x + t_{n-1} + t_n + 2l$. In the third case (Fig.4.7), $(\sum_{i=0}^r y_i) > x + t_{n-1} + l$ and $(\sum_{i=0}^{r-1} y_i) < x + y_r + t_{n-1} + l$. Basically what we are going to observe a result of these three cases is that site t_n and t_{n-1} have to be the last two sites to be visited in the optimal schedule.

Note that in Fig.4.5, $g' \leq h \leq h' \leq i$. The first schedule in Fig.4.5 has a longer expected time until agent2 finishes the entire tour after the finishing time of the last site n. This is shown as follows. The expected time for respective schedules in Fig.4.5 are:

- The first schedule:

$$\begin{aligned}
C_{S2_1} \quad = \quad & px + p(1-p)y_0 + p(1-p)^2(y_0 + y_1) + ... + p(1-p)^h(y_0 + ... + y_{h-1}) \\
& + p(1-p)^{h+1}(x + t_{n-1} + l) + p(1-p)^{h+2}(y_0 + ... + y_h) \\
& + ... + p(1-p)^{j+1}(y_0 + ... + y_{j-1}) + p(1-p)^{j+2}(x + t_{n-1} + t_n + 2l) \\
& + p(1-p)^{j+3}(y_0 + .. + y_j) + ... + (1-p)^{r+2}(y_0 + ... + y_r - 1) \\
& + p(1-p)^{r+3}z_0 + p(1-p)^{r+4}(z_0 + z_1) \\
& + ... + p(1-p)^{r+a+2}(z_0 + ... + z_{a-1}) + p(1-p)^{r+a+3}(y_0 + ... + y_r) \\
& + p(1-p)^{r+a+4}(z_0 + ... + z_a) + ... + p(1-p)^{r+b+3}(z_0 + ... + z_{b-1})
\end{aligned}$$

$$+(1-p)^{r+b+4}(z_0+...+z_b)$$

- The second schedule:

$$\begin{aligned}
C_{S2_2} &= px+p(1-p)y_0+p(1-p)^2(y_0+y_1)+...+p(1-p)^{g'}(y_0+...+y_{g'-1}) \\
&+p(1-p)^{g'+1}(x+y_r)+p(1-p)^{g'+2}(y_0+...+y'_g) \\
&+...+p(1-p)^{h'+1}(y_0+...+y_{h'-1})+p(1-p)^{h'+2}(x+y_r+t_{n-1}+l) \\
&+p(1-p)^{h'+3}(y_0+...+y_{h'})+...+p(1-p)^{j'+2}(y_0+...+y_{j'-1}) \\
&+p(1-p)^{j'+3}(x+y_r+t_{n-1}+t_n+2l)+p(1-p)^{j'+4}(y_0+...+y_{j'}) \\
&+...+p(1-p)^{r+3}(y_0+...+y_{r-1})+p(1-p)^{r+4}z_0 \\
&+...+p(1-p)^{r+b+4}(z_0+...+z_b)
\end{aligned}$$

The difference of the expected time of the above two schedules is :

$$\begin{aligned}
C_{S2_1}-C_{S2_2} &= p(1-p)^{g'+1}(\sum_{i=0}^{g'}y_i-x-y_r) \\
&+p(1-p)^{g'+2}y_{g'+1}+...+p(1-p)^h y_{h-1} \\
&+p(1-p)^{h+1}(x+t_{n-1}+l-\sum_{i=0}^{h-1}y_i) \\
&+p(1-p)^{h'+2}(\sum_{i=0}^{h'}y_i-x-y_r-t_{n-1}-l) \\
&+p(1-p)^{h'+3}y_{h'+1}+...+p(1-p)^{j+1}y_{j-1} \\
&+p(1-p)^{j+2}(x+t_{n-1}+t_n+2l-\sum_{i=0}^{j-1}y_i) \\
&+p(1-p)^{j'+3}(\sum_{i=0}^{j'}y_i-x-y_r-t_{n-1}-t_n-2l) \\
&+p(1-p)^{j'+4}(y_{j'+1})+...+p(1-p)^{r+2}(y_{r-1}) \\
&+p(1-p)^{r+3}(z_0-y_0+...+y_{r-1})+ \\
&+p(1-p)^{r+4}(z_1)+..+p(1-p)^{r+a+2}(z_{a-1})
\end{aligned}$$

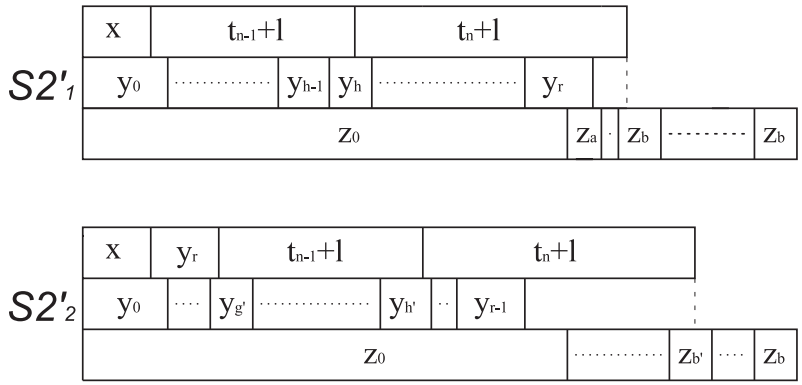

Figure 4.6: The optimal assignment of the last two sites (3)

$$+p(1-p)^{r+a+3}(y_0 + ... + y_r - z_0 - ... - z_{a-1}) \tag{4.14}$$

Since each term in the difference (4.14) is non-negative, the whole difference is non-negative, which implies that the second schedule in Fig.4.5 has a smaller expected time. If $\sum_{i=0}^{r} y_i > \sum_{i=0}^{b} z_i$ or if $\sum_{i=0}^{r-1} y_i > z_0$ and $b = 0$, $C_{S2-1} - C_{S2_2}$ is still non-negative as long as $p \geq 0.5$, which can be proved by adding a minor modification to the above difference (4.14).

It is easy to understand from the result that moving a site from the tour of agent 2 to the tour of agent 1 in the first schedule in Fig.4.5 to decrease agent 2's expect time after site n, reduces the total expected time of the schedule.

There may be a question as to which site should be moved from agent 1 to agent 2. We can have a smaller expected time by moving some other site rather than y_m from agent 1 to agent 2. This is valid because this movement still contributes to the reduction of the time difference between the tours of agent 1 and agent 2.

Next, in the schedule in Fig.4.5, we set $j-1$ to be r, which means that $\sum_{i=0}^{r} y_i < x+t_{n-1}+t_n+2l$. The diagram for this situation is described in Fig.4.6. The situation will be observed if we follows the result of the case in Fig.4.5 and move a site from the agent 2's tour to the agent 1's tour.

The total expected costs of $C_{S2'_1}$ and $C_{2S'_2}$ are:

- The first schedule:

$$
\begin{aligned}
C_{S2'_1} &= px + p(1-p)y_0 + p(1-p)^2(y_0+y_1) + ... + p(1-p)^h(\sum_{i=0}^{h-1} y_i) \\
&\quad + p(1-p)^{h+1}(x + t_{n-1} + l) + p(1-p)^{h+2}(\sum_{i=0}^{h} y_i) \\
&\quad + ... + p(1-p)^{r+1}(\sum_{i=0}^{r-1} y_i) + p(1-p)^{r+2} z_0 \\
&\quad + p(1-p)^{r+3}(z_0 + z_1) + .. + p(1-p)^{r+a+1}(\sum_{i=0}^{a-1} z_i) \\
&\quad + p(1-p)^{r+a+2}(\sum_{i=0}^{r} y_i) + p(1-p)^{r+a+3}(\sum_{i=0}^{a} z_i) \\
&\quad + ... + + p(1-p)^{r+b+2}(\sum_{i=0}^{b-1} z_i) + p(1-p)^{r+b+3}(x + t_{n-1} + t_n) \\
&\quad + p(1-p)^{r+b+4}(\sum_{i=0}^{b} z_i) + ... + p(1-p)^{r+c+4}(\sum_{i=0}^{c} z_i)
\end{aligned}
$$

- The second schedule:

$$
\begin{aligned}
C_{S2'_2} &= px + p(1-p)y_0 + p(1-p)^2(y_0+y_1) + ... + p(1-p)^g(\sum_{i=0}^{g-1} y_i) \\
&\quad + p(1-p)^{g+1}(x + y_r) + p(1-p)^{g+2}(\sum_{i=0}^{g} y_i) \\
&\quad + ... + p(1-p)^{h'+1}(\sum_{i=0}^{h'-1} y_i) + p(1-p)^{h'+2}(x + y_r + t_{n-1} + l) \\
&\quad + p(1-p)^{h'+3}(\sum_{i=0}^{h'} y_i) + ... + p(1-p)^{r+2}(\sum_{i=0}^{r-1} y_i) \\
&\quad + p(1-p)^{r+3} z_0 + ... + p(1-p)^{r+b'+2}(\sum_{i=0}^{b'-1} z_i) \\
&\quad + p(1-p)^{r+b'+3}(x + y_r + t_{n-1} + t_n + 2l) + p(1-p)^{r+b'+4}(\sum_{i=0}^{b'} z_i)
\end{aligned}
$$

$$+\ldots + p(1-p)^{r+c+4}(\sum_{i=0}^{c} z_i)$$

The difference of the total expected costs becomes:

$$\begin{aligned}
C_{S2'_1} - C_{S2'_2} &= p(1-p)^{g+1}(\sum_{i=0}^{g} y_i - x - y_r) \\
&\quad + p(1-p)^{g+2} y_{g+1} + \ldots + p(1-p)^{h} y_{h-1} \\
&\quad + p(1-p)^{h+1}(x + t_{n-1} + l - \sum_{i=0}^{h-1} y_i) \\
&\quad + p(1-p)^{r+2}(z_0 - \sum_{i=0}^{r-1} y_i) + p(1-p)^{r+3} z_1 \\
&\quad + \ldots + p(1-p)^{r+b+2} z_{b-1} + p(1-p)^{r+b+3}(x + t_{n-1} + t_n - \sum_{i=0}^{b} z_i) \\
&\quad + p(1-p)^{r+b+4} z_0 + \ldots + p(1-p)^{r+b'+2}(z_{b'}) \\
&\quad + p(1-p)^{r+b'+3}(\sum_{i=0}^{b'-1} z_i - x - y_r - t_{n-1} - t_n - 2l) \qquad (4.15)
\end{aligned}$$

The difference of the total expected times (4.15) is non-negative because all the terms in (4.15) is non-negative. If $x + y_m + t_{n-1} + t_n + 2l > \sum_{i=0}^{b'} z_i$ in Fig.4.6, $C_{S2'_1} - C_{S2'_2}$ is still non-negative as long as $p \geq 0.5$, which can be proved by adding a minor modification to the above difference (4.15). The above result implies that a site has to be moved from the tour of agent 2 to that of agent 1 to get smaller total expected time as long as $y_0 + \ldots + y_{r-1} - x < y_r + t_{n-1} + t_n + 2l$.

Next, we will observe what happens if $y_0 + \ldots + y_{r-1} - x \leq y_r + t_{n-1} + t_n + 2l$, which is the case in Fig.4.7. This case will be encountered if we move a site from the tour of agent 2 to the tour of agent 1 in order to decrease the total expected time in the previous case (Fig.4.6).

The total expected cost for each schedule is, respectively,

- The first schedule:

$$\begin{aligned}
C_{S3_1} &= px + p(1-p)y_0 + p(1-p)^2(y_0 + y_1) + \ldots + p(1-p)^r(y_0 + \ldots + y_{r-1}) \\
&\quad + p(1-p)^{r+1} z_0 + \ldots + p(1-p)^{(}r + a)(z_0 + \ldots + z_{a-1})
\end{aligned}$$

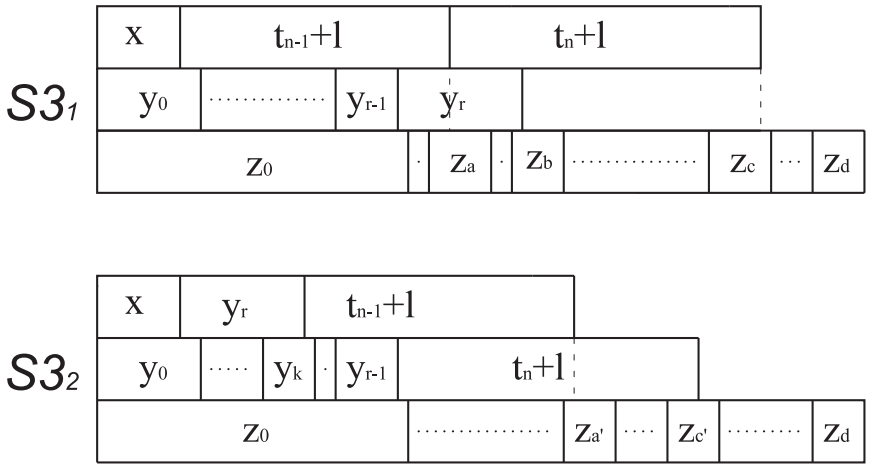

Figure 4.7: The optimal assignment of the last two sites (4)

$$+p(1-p)^{r+a+1}(x+t_{n-1}+l)+p(1-p)^{(}r+a+2)(z_0+..+z_a)$$

$$+...+p(1-p)^{r+b+1}(z_0+...+z_{b-1})+p(1-p)^{r+b+2}(y_0+...+y_r)$$

$$+p(1-p)^{r+b+3}(z_0+...+z_b)+...+p(1-p)^{r+c+2}(z_0+...+z_{c-1})$$

$$+p(1-p)^{r+c+3}(x+t_{n-1}+t_n+2l)+p(1-p)^{r+c+4}(z_0+...+z_c)$$

$$+...+p(1-p)^{r+d+4}(z_0+....+z_d)$$

- The second schedule:

$$\begin{aligned}
C_{S3_2} &= px+p(1-p)y_0+p(1-p)^2(y_0+y_1)+...+p(1-p)^h(y_0+...+y_{h-1}) \\
&\quad +p(1-p)^{h+1}(x+y_r)+p(1-p)^{h+2}(y_0+...+y_h) \\
&\quad +p(1-p)^{h+3}(y_0+...+y_{h+1})+....+p(1-p)^{r+1}(y_0+...y_{r-1}) \\
&\quad +p(1-p)^{r+2}z_0+p(1-p)^{r+3}(z_0+z_1) \\
&\quad +...+p(1-p)^{r+a'+1}(z_0+...+z_{a'-1}) \\
&\quad +p(1-p)^{r+a'+2}(x+y_r+t_{n-1}+l)+p(1-p)^{r+a'+3}(z_0+...+z_{a'}) \\
&\quad +...+p(1-p)^{r+c'+2}(z_0+...+z_{c'-1}) \\
&\quad +p(1-p)^{r+c'+3}(y_0+..+y_{r-1}+t_n+l)
\end{aligned}$$

$$+p(1-p)^{r+c'+4}(z_0+..+z_{c'})$$

$$+...+p(1-p)^{r+d+4}(z_0+...+z_d)$$

Then, the difference of the total expected cost between the above schedules is:

$$\begin{aligned}
C_{S3_1}-C_{S3_2} &= p(1-p)^{h+1}(\sum_{i=0}^{h} y_i - x - y_r) \\
&\quad +p(1-p)^{h+2}y_{h+1}+...+p(1-p)^{r}y_{r-1} \\
&\quad p(1-p)^{r+1}(z_0-\sum_{i=0}^{r-1} y_i) \\
&\quad +p(1-p)^{r+2}(z_1)+...+p(1-p)^{r+a}(z_{a-1}) \\
&\quad +p(1-p)^{r+a+1}(x+t_{n-1}+l-\sum_{i=0}^{a-1} z_i) \\
&\quad +p(1-p)^{r+a'+2}(\sum_{i=0}^{a'} z_i - x - y_r - t_{n-1} - l) \\
&\quad +p(1-p)^{r+a'+3}(z_{a'+1})+...+p(1-p)^{r+b+1}(z_{b-1}) \\
&\quad +p(1-p)^{r+b+2}(\sum_{i=0}^{r} y_i - \sum_{i=0}^{b-1} z_i) \\
&\quad +p(1-p)^{r+c'+3}(\sum_{i=0}^{c'} z_i - \sum_{i=0}^{r-1} y_i - t_n - l) \\
&\quad +p(1-p)^{r+c'+4}(z_{c'+1})+...+p(1-p)^{r+c+2}z_{c-1} \\
&\quad +p(1-p)^{r+c+3}(x+t_{n-1}+t_n+2l-\sum_{i=0}^{c-1} z_i) \qquad (4.16)
\end{aligned}$$

All the terms in the above equation (4.16) are non-negative. Thus the second schedule has a smaller total expected time. This result implies that the last two sites $n-1$ and n have to be finished at the end of each tour of agent 1 and agent 2. Thus, if $y_0+...+y_{r-1}-x \leq y_r+t_{n-1}+t_n+2l$, the site has to be moved from the tour of agent 2 to that of agent 1 and the last site n has to be appended at the end of the tour of agent 2 to get a smaller total expected time.

Note that in case where $\sum_{i=0}^{c} z_i < x+t_{n-1}+t_n+2l$ in Fig.4.7, the difference of the expected

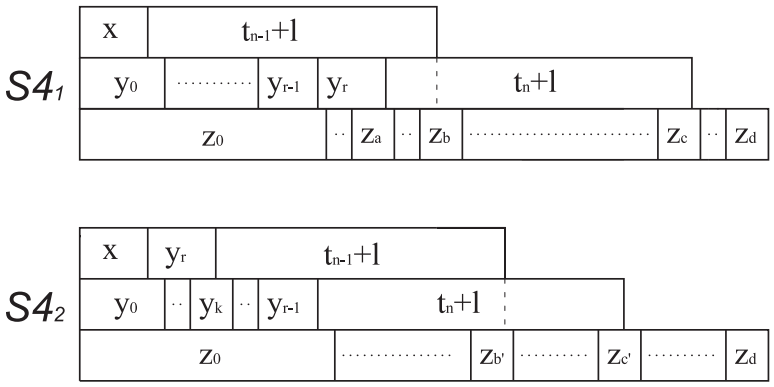

Figure 4.8: The optimal assignment of the last two sites (5)

time between the two schedule $C_{S3_1} - C_{S3_2}$ is still non-negative as long as $p \geq 0.5$. This can be verified with a minor modification in (4.16). Thus the above result holds in this case, too.

Next is the final case in the step1 proof. It will show that two sites s_{n-1} and s_n have to be the last two sites in the optimal tour and they have to visited by different agents. Now we considered the two assignments depicted in Fig.4.8. The case in Fig.4.8 is observed if we move a site from the tour of agent 2 to the tour of agent 1 in order to decrease the total expected time in the previous case (Fig.4.7).

The total expected cost of each assignment is, respectively,

$$\begin{aligned}
C_{S4_1} &= px + p(1-p)y_0 + p(1-p)^2(y_0+y_1) + ... + p(1-p)^r(y_0+...+y_{r-1}) \\
&\quad + p(1-p)^{r+1}z_0 + p(1-p)^{r+2}(z_0+z_1) \\
&\quad + ... + p(1-p)^{r+a}(z_0+...+z_{a-1}) + p(1-p)^{r+a+1}(y_0+...+y_r) \\
&\quad + p(1-p)^{r+a+2}(z_0+...+z_a) + ... + p(1-p)^{r+b+1}(z_0+...+z_{b-1}) \\
&\quad + p(1-p)^{r+b+2}(x+t_{n-1}+l) \\
&\quad + p(1-p)^{r+b+3}(z_0+...+z_b) + ... + p(1-p)^{r+c+2}(z_0+...+z_{c-1}) \\
&\quad + p(1-p)^{r+c+3}(y_0+...+y_r+t_n+l) + p(1-p)^{r+c+4}(z_0+...+z_c) \\
&\quad + ... + (1-p)^{r+d+4}(z_0+...+z_d)
\end{aligned}$$

and

$$
\begin{aligned}
C_{S4_2} &= px + p(1-p)y_0 + p(1-p)^2(y_0+y_1) \\
&\quad +\ldots + p(1-p)^k(y_0+\ldots+y_{k-1}) + p(1-p)^{k+1}(x+y_r) \\
&\quad +p(1-p)^{h+2}(y_0+..+y_k) + p(1-p)^{r+1}(y_0+\ldots+y_{r-1}) \\
&\quad +p(1-p)^{r+2}z_0 + \ldots p(1-p)^{r+b'+1}(z_0+\ldots z_{b'-1}) \\
&\quad +p(1-p)^{r+b'+2}(x+y_r+t_{n-1}+l) + p(1-p)^{r+b'+3}(z_0+\ldots+z_{b'}) \\
&\quad +\ldots + p(1-p)^{r+c'+2}(z_0+\ldots+z_{c'-1}) + p(1-p)^{r+c'+3}(y_0+\ldots+y_{r-1}+t_n+l) \\
&\quad +(1-p)^{r+c'+4}(z_0+\ldots+z_{c'}) + \ldots + (1-p)^{r+d+4}(z_0+\ldots+z_d)
\end{aligned}
$$

The difference of the above two total expected costs is:

$$
\begin{aligned}
C_{S4_1} - C_{S4_2} &= p(1-p)^{k+1}(\sum_{i=0}^{k} y_i - x - y_r) \\
&\quad +p(1-p)^{k+2}y_{k+1} + \ldots + p(1-p)^r y_{r-1} \\
&\quad +p(1-p)^{r+1}(z_0 - \sum_{i=0}^{r-1} y_i) \\
&\quad +p(1-p)^{r+2}z_1 + \ldots + p(1-p)^{r+a}z_{a-1} \\
&\quad +p(1-p)^{r+a+1}(\sum_{i=0}^{r} y_i - \sum_{i=0}^{a-1} z_i) \\
&\quad +p(1-p)^{r+b+2}(x + t_{n-1} + l - \sum_{i=0}^{b} z_i) \\
&\quad +p(1-p)^{r+b+3}(-z_{b+1}) + \ldots + p(1-p)^{r+b'+1}(-z_{b'-1}) \\
&\quad +p(1-p)^{r+b'+2}(\sum_{i=0}^{b'-1} z_i - x - y_r - t_{n-1} - l) \\
&\quad +p(1-p)^{r+c'+3}(\sum_{i=0}^{c'} z_i - \sum_{i=0}^{r-1} y_i - t_n - l) \\
&\quad +p(1-p)^{r+c'+4}z_{c'+1} + \ldots + p(1-p)^{r+c+2}z_{c-1}
\end{aligned}
$$

$$
\begin{aligned}
& +p(1-p)^{r+c+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{c-1} z_i) \\
\geq \quad & p(1-p)^{k+1}(\sum_{i=0}^{k} y_i - x - y_r) \\
& +p(1-p)^{k+2} y_{k+1} + ... + p(1-p)^{r} y_{r-1} \\
& +p(1-p)^{r+a+1}(z_0 - \sum_{i=0}^{r-1} y_i) \\
& +p(1-p)^{r+a+1} z_1 + ... + p(1-p)^{r+a+1} z_{a-1} \\
& +p(1-p)^{r+a+1}(\sum_{i=0}^{r} y_i - \sum_{i=0}^{a-1} z_i) \\
& +p(1-p)^{r+a+1}(x + t_{n-1} + l - \sum_{i=0}^{b} z_i) \\
& +p(1-p)^{r+a+1}(-z_{b+1}) + ... + p(1-p)^{r+a+1}(-z_{b'-1}) \\
& +p(1-p)^{r+b'+2}(\sum_{i=0}^{b'-1} z_i - x - y_r - t_{n-1} - l) \\
& +p(1-p)^{r+c'+3}(\sum_{i=0}^{c'} z_i - \sum_{i=0}^{r-1} y_i - t_n - l) \\
& +p(1-p)^{r+c'+4} z_{c'+1} + ... + p(1-p)^{r+c+2} z_{c-1} \\
& +p(1-p)^{r+c+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{c-1} z_i) \\
= \quad & p(1-p)^{k+1}(\sum_{i=0}^{k} y_i - x - y_r) \\
& +p(1-p)^{k+2} y_{k+1} + ... + p(1-p)^{r} y_{r-1} \\
& +p(1-p)^{r+a+1}(y_r + x_{n-1} + l - \sum_{i=0}^{b'-1} z_i) \\
& +p(1-p)^{r+b'+2}(\sum_{i=0}^{b'-1} z_i - x - y_r - t_{n-1} - l)
\end{aligned}
$$

$$+p(1-p)^{r+c'+3}(\sum_{i=0}^{c'} z_i - \sum_{i=0}^{r-1} y_i - t_n - l)$$

$$+p(1-p)^{r+c'+4} z_{c'+1} + ... + p(1-p)^{r+c+2} z_{c-1}$$

$$+p(1-p)^{r+c+3}(\sum_{i=0}^{r} y_i + t_n + l - \sum_{i=0}^{c-1} z_i) \tag{4.17}$$

All the terms in (4.17) are non-negative, thus the difference $C_{S4_1} - C_{S4_2}$ is non-negative. This implies that if there is a site which finishes its task between the starting time of the site s_{n-1} and s_n, as described in Fig.4.9, the site y_r in Fig.4.9 should be moved to agent 1 to get a smaller total expected cost. Note that even if $\sum_{i=0}^{c'} z_i < \sum_{y=0}^{r} + t_n + l$, we can obtain the same result by adding a monitor modification to (4.17).

From the results of the situations in Fig.4.4 through Fig.4.9, We have proved that sites s_{n-1} and s_n should be visited at the end of the optimal schedule in sequence by different agents

Step2– In this step, we will show that the last three sites in the optimal schedule are s_{n-2}, s_{n-1} and s_n, and these sites have to be visited in the sequence alternately by different agents.

In the three assignments in Fig.4.10, their total expected costs are:

- The first schedule:

$$\begin{aligned}
C_{S5_1} \quad = \quad & pz_0 + p(1-p)y_0 + p(1-p)^2 x \\
& +p(1-p)^3(z_0 + z_1) + ... + p(1-p)^{a+1}(z_0 + ... + z_{a-1}) \\
& +p(1-p)^{a+2}(y_0 + y_r) + p(1-p)^{a+3}(z_0 + ... + z_a) \\
& +... + p(1-p)^{b+2}(z_0 + ... + z_{b-1}) + p(1-p)^{b+3}(x + t_{n-2} + l) \\
& +p(1-p)^{b+4}(z_0 + ... + z_b) + ... + p(1-p)^{c+3}(z_0 + ... + z_{c-1}) \\
& +p(1-p)^{c+4}(y_0 + y_r + t_{n-1} + l) + p(1-p)^{c+5}(z_0 + ... + z_c) \\
& +... + p(1-p)^{d+4}(z_0 + ... + z_{d-1}) + p(1-p)^{d+5}(x + t_{n-2} + t_n + 2l) \\
& +p(1-p)^{d+6}(z_0 + ... + z_d) + ... + p(1-p)^{e+5}(z_0 + ... + z_{e-1}) \\
& +(1-p)^{e+6}(z_0 + ... + z_e)
\end{aligned}$$

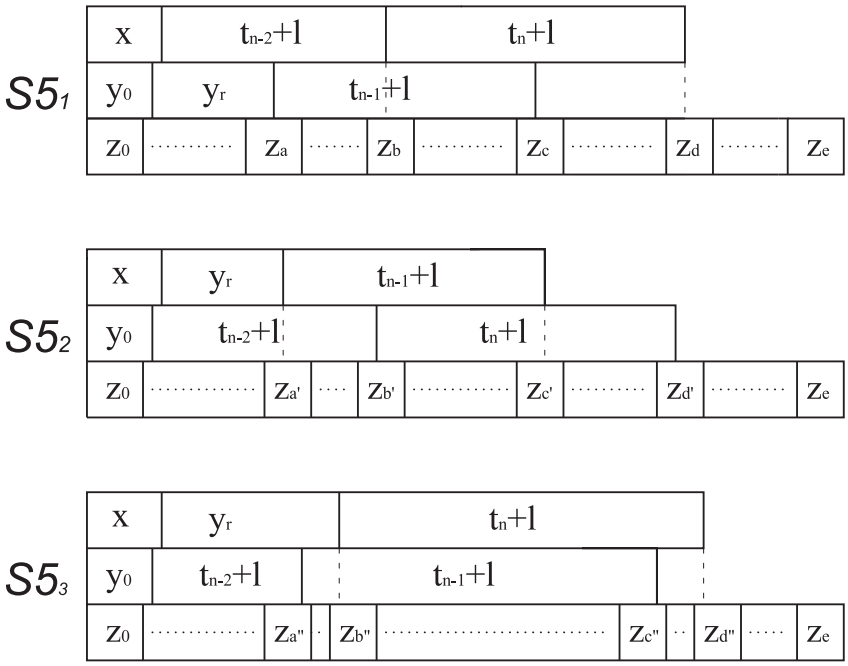

Figure 4.9: The optimal assignment of the last two sites (6)

- The second schedule:

$$\begin{aligned}
C_{S5_2} &= pz_0 + p(1-p)y_0 + p(1-p)^2 x + p(1-p)^3(z_0 + z_1) \\
&\quad + \ldots + p(1-p)^{a'+1}(z_0 + \ldots + z_{a'-1}) + p(1-p)^{a'+2}(x + y_r) \\
&\quad + p(1-p)^{a'+3}(z_0 + \ldots + z_{a'}) + \ldots + p(1-p)^{b'+2}(z_0 + \ldots + z_{b'-1}) \\
&\quad + p(1-p)^{b'+3}(y_0 + t_{n-2} + l) + p(1-p)^{b'+4}(z_0 + \ldots + z_{b'}) \\
&\quad + \ldots + p(1-p)^{c'+3}(z_0 + \ldots + z_{c'-1}) + p(1-p)^{c'+4}(x + y_r + t_{n-1} + l) \\
&\quad + p(1-p)^{c'+5}(z_0 + \ldots + z_{c'}) + \ldots + p(1-p)^{d'+4}(z_0 + \ldots + z_{d'-1}) \\
&\quad + p(1-p)^{d'+5}(y_0 + t_{n-2} + t_n + 2l) + p(1-p)^{d'+6}(z_0 + \ldots + z_{d'}) \\
&\quad + \ldots + p(1-p)^{e+5}(z_0 + \ldots + z_{e-1}) + (1-p)^{e+6}(z_0 + \ldots + z_e)
\end{aligned}$$

- The third schedule:

$$\begin{aligned}
C_{S5_3} &= pz_0 + p(1-p)y_0 + p(1-p)^2 x + p(1-p)^3(z_0 + z_1) \\
&\quad + \ldots + p(1-p)^{a''+1}(z_0 + \ldots + z_{a''-1}) + p(1-p)^{a''+2}(y_0 + t_{n-2} + l) \\
&\quad + p(1-p)^{a''+3}(z_0 + \ldots + z_{a''}) + \ldots + p(1-p)^{b''+2}(z_0 + \ldots + z_{b''-1}) \\
&\quad + p(1-p)^{b''+3}(x + y_r) + p(1-p)^{b''+4}(z_0 + \ldots + z_{b''}) \\
&\quad + \ldots + p(1-p)^{c''+3}(z_0 + \ldots + z_{c''-1}) + p(1-p)^{c''+4}(y_0 + t_{n-2} + t_{n-1} + 2l) \\
&\quad + p(1-p)^{c''+5}(z_0 + \ldots + z_{c''}) + \ldots + p(1-p)^{d''+4}(z_0 + \ldots + z_{d''-1}) \\
&\quad + p(1-p)^{d''+5}(x + y_r + t_n + l) + p(1-p)^{d''+6}(z_0 + \ldots + z_{d''}) \\
&\quad + p(1-p)^{e+5}(z_0 + \ldots + z_{e-1}) + (1-p)^{e+6}(z_0 + \ldots + z_e)
\end{aligned}$$

The differences of the total expected time $C_{S5_{-1}} - C_{S5_2}$ and $C_{S5_1} - C_{S5_3}$ are:

$$\begin{aligned}
C_{S5_1} - C_{S5_2} &= p(1-p)^{a+2}(y_0 + y_r - \sum_{i=0}^{a} z_i) \\
&\quad + p(1-p)^{a+3}(-z_{a+1}) + \ldots + p(1-p)^{a'+1}(-z_{a'-1}) \\
&\quad + p(1-p)^{a'+2}(\sum_{i=0}^{a'-1} z_i - x - y_r) + p(1-p)^{b'+3}(\sum_{i=0}^{b'} z_i - y_0 - t_{n-2} - l)
\end{aligned}$$

$$
\begin{aligned}
& +p(1-p)^{b'+4}(z_{b'+1}) + ... + p(1-p)^{b+2}(z_{b-1}) \\
& +p(1-p)^{b+3}(x + t_{n-2} + l - \sum_{i=0}^{b-1} z_i) \\
& +p(1-p)^{c+4}(y_0 + y_r + t_{n-1} + l - \sum_{i=0}^{c} z_i) \\
& +p(1-p)^{c+5}(-z_{c+1}) + ... + p(1-p)^{c'+3}(-z_{c'-1}) \\
& +p(1-p)^{c'+4}(\sum_{i=0}^{c'-1} z_i - x - y_r - t_{n-1} - l) \\
& +p(1-p)^{d'+5}(\sum_{i=0}^{d'} z_i - y_0 - t_{n-2} - t_n - 2l) \\
& +p(1-p)^{d'+6}(z_{d'+1}) + ... + p(1-p)^{d+4}(z_{d-1}) \\
& +p(1-p)^{d+5}(x + t_{n-2} + t_n + 2l - \sum_{i=0}^{d-1} z_i) \\
\leq\ & p(1-p)^{a'+2}(y_0 - x) + p(1-p)^{b'+3}(x - y_0) \\
& +p(1-p)^{c'+4}(y_0 - x) + p(1-p)^{d'+5}(x - y_0)
\end{aligned}
$$

and

$$
\begin{aligned}
C_{S5_1} - C_{S5_3} =\ & p(1-p)^{a+2}(y_0 + y_r - \sum_{i=0}^{z} z_i) \\
& +p(1-p)^{a+3}(-z_{a+1}) + ... + p(1-p)^{a''+1}(-z_{a''-1}) \\
& +p(1-p)^{a''+2}(\sum_{i=0}^{a''-1} z_i - y_0 - t_{n-2} - l) \\
& +p(1-p)^{b''+3}(\sum_{i=0}^{b''} z_i - x - y_r) \\
& +p(1-p)^{b''+4}(z_{b''+1}) + ... + p(1-p)^{b+2}(z_{b-1}) \\
& +p(1-p)^{b+3}(x + t_{n-2} + l - \sum_{i=0}^{b} z_i)
\end{aligned}
$$

$$
\begin{aligned}
& +p(1-p)^{c+4}(y_0+y_r+t_{n-1}+l-\sum_{i=0}^{c} z_i) \\
& +p(1-p)^{c+5}(-z_{c+1})+....+p(1-p)^{c''+3}(-z_{c''-1}) \\
& +p(1-p)^{c''+4}(\sum_{i=0}^{c''-1} z_i-y_0-t_{n-2}-t_{n-1}-2l) \\
& +p(1-p)^{d''+5}(\sum_{i=0}^{d''} z_i-x-y_r-t_n-l) \\
& +p(1-p)^{d''+6}(z_{d''+1})+...+p(1-p)^{d+4}(z_{d-1}) \\
& +p(1-p)^{d+5}(x+t_{n-2}+t_{n-1}+2l-\sum_{i=0}^{d-1}) \qquad (4.18) \\
\leq\ & p(1-p)^{a''+2}(y_m-t_{n-2}-l)+p(1-p)^{b''+3}(t_{n-2}+l-y_m) \\
& +p(1-p)^{c''+4}(y_m-t_{n-2}-l)+p(1-p)^{d''+5}(t_{n-2}+l-y_m)
\end{aligned}
$$

In the difference $C_{S5_1}-C_{S5_2}$, y_0-x is non-positive, $p(1-p)^{a'+2} \geq p(1-p)^{b'+3}$ and $p(1-p)^{c'+4} \geq p(1-p)^{c'+4}$. Thus $C_{S5_1}-C_{S5_2}$ is non-positive. In the same way, in $C_{S5_1}-C_{S5_3}$, $y_m-t_{n-2}-l$ is non-positive because the task of site s_{n-2} is larger than any other task except those of s_{n-1} and s_n. $p(1-p)^{a''+2} \geq p(1-p)^{b''+3}$ and $p(1-p)^{c''+4} \geq p(1-p)^{d''+5}$. Thus the difference $C_{S5_1}-C_{S5_3}$ is also non-positive.

If $\sum_{i=0}^{e} z_i < x+t_{n-2}+t_n+2l$, the last part of the equations of differences C_{S5_1} and C_{S5_2} change a little. However, both the differences can be proved easily to be non-positive as long as $p \geq 0.5$.

Thus, we have proved that the third site from the last in the optimal schedule should be the site s_{n-2} if the last two sites are s_{n-1} and s_n, and that those three sites are visited alternately by different agents as shown in the first schedule in Fig.4.9. In other words, we have proved that the optimal schedule of the last three sites is obtained by the greedy method based on the computation time of the task.

Step3 – We now assume that the schedule of the last $k-1$ sites is obtained by the greedy method based on the computation time of the task, as depicted in the first figure of Fig.4.10. Under such an assumption, we show that the kth site from the last also should be decided by the greedy method for the optimal schedule. This can be simply proved using the result of the previous proof. As we can

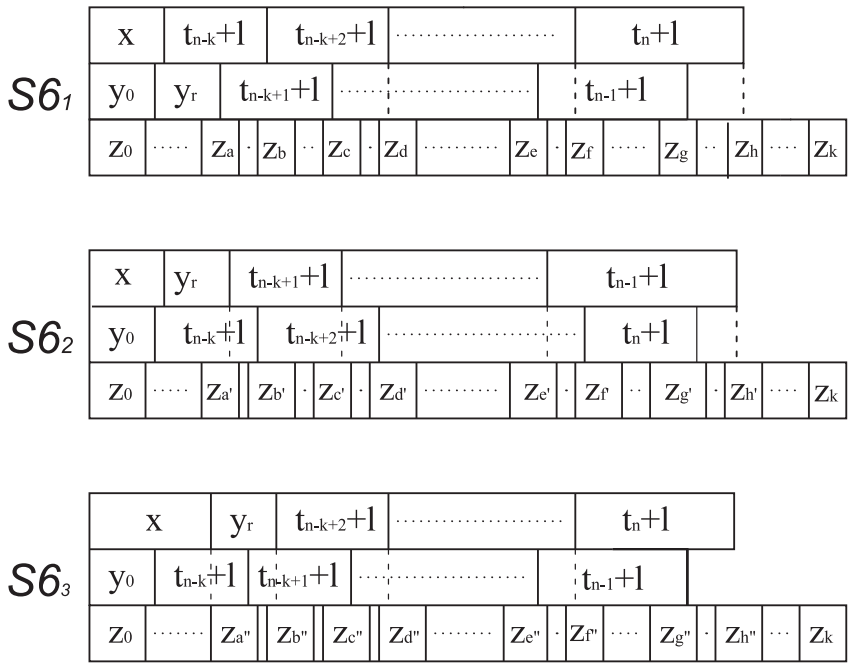

Figure 4.10: The optimal assignment of the last two sites (7)

see in Fig.4.10, the schedules of sites $< y_r, s_{n-k}, s_{n-k+1}, s_{n-k+2} >$ are identical to the schedules in Fig.4.9 where $k = 2$, and the first schedule has the smallest total expected cost in Fig.4.9. Since the schedule of sites s_{n-k+3} through s_n are decided by the greedy method based on the computation time of the task, the calculations of the differences of the expected times $C_{S6_1} - C_{S6_2}$ and $C_{S6_1} - C_{S6_3}$ will have the same pattern as that of the calculation of the sites s_{n-1} through s_n in the previous case (Fig.4.9), which shows the first schedule is the best. Thus both differences $C_{S6_1} - C_{S6_2}$ and $C_{S6_1} - C_{S6_3}$ are non-positive.

By using induction reasoning based on the results from the step1, step2 and step3, we can now prove that the optimal scheduling of 2 agents can be obtained by the greedy method based on the computation time of the task.

Step4 – As an extension of the previous step, the optimal schedule of the sites visited by multiple agents ($m \geq 2$) also should be obtained by the greedy method because this is the only schedule that allows the schedule of two arbitrary agents to be the same as the one that is obtained by the greedy method.

4.3.4 Constant Computation Time

Assuming that the computation time at all the sites are same, TMAP becomes the simple sorting problem that can be solved in polynomial time.

Theorem 12 *Computation time at each site is constant, i.e., $t_i = t$ for all i. Latencies between machines are constant, i.e., $l_{ij} = l$ for all i, j. Probabilities of success, p_i, are arbitrary. TMAP with m agents can be solved in polynomial time.*

proof – Let J_k be the number of sites visited by agent k. Let the jth site visited by the kth agent be i_{jk} where $1 \leq j \leq J_k$ and $1 \leq k \leq m$. Now we convert the m-agents into one agent that can visit m sites at one period of a stage. The length of the period is $t + l$. We can imagine that all the sites visited by the new agent in one stage are merged into one site. The probability of success at the merged site $p(i_j)$ is

$$p(i_j) = 1 - \prod_{k=1}^{m} (1 - p(i_{jk}))$$

where $p(i_{jk}) = 0$ if there is no jth site visited by the kth agent.

Note that $p(i_j)$ monotonically increases as any $p(i_{jk})$ increases.

If $m = 1$, the optimal schedule S will be achieved if sites are visited in decreasing order of probabilities of success. Moreover, the larger the probabilities are in an earlier stage, the smaller the expected total time of the schedule is. Thus, the optimal multiple agent schedule S is obtained if we assign the first m sites in the list of sites sorted in decreasing order of probabilities, into the next available stage. The assigned sites at each stage can be assigned into the m-agent arbitrarily. This assignment does not affect the expected total time.

It is easy to see that both construction of the above optimal scheduling and its verification can be performed in polynomial time. Q.E.D.

4.4 The Traveling Agent Problem with Deadlines

The above results are relevant to agent planning problems without deadlines. In this section, we address the important problem of handling a deadline for each site (The Traveling Agent problem with Deadlines). For example, agents encounter this situation when they have information regarding

when sites are going to shutdown or be isolated due to link disconnection. The goal of the problem is to obtain tours that finish an agent's task with the highest probability of success before machines become unusable. We present a pseudo-polynomial algorithm for finding such optimal tours below.

Assumption 5 *Each site i (including the home site s_0) has a deadline, D_i, by which an agent must visit the site or else the site will become unavailable.*

Theorem 13 *Assume in TAP that $l = l_{ij} = l_{km} \geq 0$ for all i, j, k, m, namely Assumption 1. Computation times, t_i, and probabilities, p_i, can be arbitrary. The agent must finish its task by visiting sites and returning to the home site before the respective deadlines. The optimal tour for this TAP with Deadlines maximizes the probability of success without exceeding a deadline at each site. (The deadline time D_i is measured from the time the agent leaves the home site.) Such an optimal tour can be computed in pseudo-polynomial time using a dynamic programming algorithm. Input times are assumed to be integers and pseudo-polynomial means polynomial in the values of input time, not the number of bits required to represent them.*

Proof – The proof consists of four steps. In the first step, we formalize the concept of task completion before deadlines. The second step shows that sites should be visited in increasing order of deadlines in order to obtain the optimal answer. The third step introduces two array $e(j, s)$ and $p(j, s)$. The first array is the boolean function which tells if there is a subset which consists of the first through jth sorted sites and which has the total time equal to s. The second array is the total probability of success $p(j, s)$ of the subset that makes $e(j, s) = true$. The fourth step constructs these arrays in pseudo-polynomial time using dynamic programming. The optimal tour is the sorted subset that maximizes $p(j, s)$.

Step 1 - In this step we formalize the meaning of task completion before deadlines. We assume that the agent start time is $\tau = 0$ and that the agent must visit site i before $\tau = D_i$. If an agent visits s_i, this means that it failed at every site it visited before. The time required to visit all sites before i_m in failure for a tour $T'(m) = < i_1, i_2, ..., i_m >$ where $m < n$ and return to the home site is:

$$C_{T'}(m) = l_{01} + t_{i_1} + \sum_{k=2}^{m} (l_{i_{k-1} i_k} + t_{i_k})$$

$$= \sum_{k=1}^{m} (l + t_{i_k})$$

$$= m \cdot l + \sum_{k=1}^{m} t_{i_k}$$

The above $C_{T'}(m)$ cannot be greater than the deadline D_{i_m} at the site i_m and, in addition, the total time for all tours $T'(k)$ for $k < m$ should not exceed the deadline D_{i_k}. Although the order for visiting sites does not affect the total time $C_{T'}(m)$ and the total probability of success for the tour, changing the order might violate the deadline constraints.

The tour which maximizes the probability of successful completion can be computed in pseudo-polynomial time using dynamic programming in the following steps.

Step 2 - We show that all sites should be visited in increasing order of deadline. Without loss of generality, let the sorted sequence be a tour $T =< 1, 2, ..., n >$. This tour has a minimum maximum lateness which for a tour $T^* =< i_1, i_2, ..., i_n >$ is defined to be $\min_{T^*}\{\max_{k=1}^{n}(C'_{T^*}(k) - D_{i_k})\}$ where $C'_{T^*}(k)$ represents the completion time at the i_kth site [29].

Step 3 - We introduce two arrays of size n (number of sites) by $\min(B, D_0 - l)$ where $B = \sum_{k=1}^{n} t_k + n \cdot l$ (total possible time for visiting all sites) and $D_0 - l$ represents the deadline by which an agent should leave the last site for its home machine.

For integer j where $1 \leq j \leq n$, let $e(j, s) = 1$, if there is a subset of $\{1, 2, .., j\}$ for which the total maximum time is exactly s and each site of which can be visited no later than its deadline. If such a subset does not exist, $e(j, s) = 0$ so e is essentially Boolean. We also define the array value $p(j, s) = 0$ if $e(j, s) = 0$ and $p(j, s) = 1 - \prod_{k \in T_{e(j,s)}}(1 - p_k)$ if $e(j, s) = 1$. Here $T_{e(j,s)}$ is the set of nodes whose time add up to s (thus making the deadline constraint transparent). This subset of sites has the maximum probability of success among all such subsets and each of its sites can be visited no later than the site's deadline. That probability is precisely what is stored in $p(j, s)$.

Step 4 - The values of $e(j, s)$ and $p(j, s)$ are obtained by dynamic programming. The dimensions of the arrays e and p are both n by $\min(B, D_0 - l)$. They are all initialized to be 0.

We start the algorithm with row $j = 1$ and proceed to calculate the following rows of e from the previous rows. For $j = 1$, $e(1, s) = 1$ if and only if either $s = 0$, or $s = l + t_1$ and $s \leq D_1$. If $e(1, s) = 0$, or $e(1, s) = 1$ and $s = 0$, $p(1, s)$ is set to be 0. If $e(j, s) = 1$ and $s = l + t_1$,

$p(j,s) = p_1 = 1 - (1 - p_1)$.

For $j \geq 2$, we set the value of $e(j,s) = 1$ if and only if $e(j-1,s) = 1$ is true, $l + t_j \leq s$ and $e(j-1, s-l-t_j) = 1$ where $s \leq D_j$, or $l + t_j = s$. This means that we can construct a tour with total cost s in the case of failure in one of three possible ways:

1. There was already a tour involving sites $1, 2, ..., j-1$ with total time s (the case $e(j-1,s) = 1$), each site of which can be visited no later than its deadline;

2. There is a nonempty tour with total time $s - t_j - l$ involving a subset of sites $1, 2, ..., j-1$ and we can add the site j to that tour if $s \leq D_j$ (this is the case, $e(j-1, s-t_j-l) = 1$ and $l + t_j \leq s$ where $s \leq D_j$) ;

3. $l + t_j = s$ and so we have a tour with only the site j on it if $s \leq D_j$.

In the second case, we check if we can add the site j to a tour by comparing s and D_j when $e(j-1, s-l-t_j) = 1$ and $l + t_j \leq s$.

It is obvious that a tour which satisfies $e(j-1, s-l-t-j) = 1$ has a maximum lateness of 0 since the tour does not violate the deadline constraints. Now if we append the site j at the end of the tour, the total time becomes s. The maximum lateness of sites in the new tour is $\max\{0, s-D_j\}$ because those sites are sorted in increasing order of their deadline. Since the maximum lateness is minimum among those of all possible permutations of sites in the tour, it is impossible to decrease the maximum lateness by changing the order of sites in the tour. Thus, site j cannot be added to the tour that satisfies $e(j,s) = 1$ if $s > D_j$.

We also define elements of the jth row of p as follows. If $e(j,s) = 0$, $p(j,s) = 0$. We handle the cases for which $e(j,s) = 1$ in the following way:

1. There is a tour already involving a subset of $1, 2, ..., j-1$ with a total time of s, each node of which can be visited no later than its deadline but j could not be added to any previous tour to get a new tour (case for which $e(j-1,s) = 1$ but $e(j-1, s-t_j-l) = 0$): $p(j,s) = p(j-1,s)$;

2. There is no tour with total time s based on sites $1, 2, ..., j-1$ but there is one when s_j is added to a tour (case when $e(j-1,s) = 0$ and $e(j-1, s-t_j-l) = 1$): $p(j,s) = p_j$ if $s \leq D_j$;

3. There is already a tour involving a subset of $1, 2, ..., j-1$ with a total time of s and another tour with total time $s-l-t_j$ using sites $1, 2, ..., j-1$ which can be visited no later than their

respective deadlines (the case when $e(j-1,s)=1$, and $e(j-1,s-t_j-l)=1$ and $s \leq D_j$): $p(j,s) = \max\{p(j-1,s), 1-(1-p_j)(1-p(j-1,s-l-t_j))\}$;

4. $s = l + t_j$, $e(j-1,s)=1$ and $s \leq D_j$ in which case $p(j,s) = \max\{p(j-1,s), p_j\}$.

This handles all cases and clearly requires only a single scan of each row, therefore about $O(\min(D_0 - l, B))$ operations per row for n rows yielding a complexity of $O(\min(D_0 - l, B) \cdot n)$ steps..

The subset which maximizes the probability of the agent's success and whose sites can be visited no later than their respective deadlines is the one that maximizes $p(n,s)$ for $s \leq D_0 - l$ (recall that $p(j,s) = 0$ if $e(j,s) = 0$). $D_0 - l$ is the deadline by which time an agent has to leave the last site for its home site 0. By visiting all the sites in that subset in increasing order of their deadline, no exceeding of deadlines will occur. However, this order cannot guarantee the minimum expected time in visiting all the sites prior to their respective deadlines.

It takes at most $n \log n$ to sort the subset in increasing order of deadlines. Therefore, the optimal solution can be obtained in time that is polynomial at most in n and B, $O(2 \cdot n \cdot \min\{\sum_{k=1}^{n} t_k + n \cdot l, D_0 - l\} + n \log n)$.

The only remaining issue is that of maintaining the list of sites on a tour corresponding to $e(j,s) = 1$ and the value of $p(j,s)$ recorded in the array. This can clearly be handled within the time bounds we have demonstrated. Q.E.D.

As mentioned above, this solution for TAP with deadlines optimizes the probability of success but does not necessarily minimize the expected time to visit all sites on the tour while satisfying the respective deadlines.

If there is only a global deadline, D, by which an agent must finish and return home, we can set the deadline of each site to be $D - l$ and use the algorithms introduced in Theorems 13 in order to obtain a solution with the highest probability of success.

4.4.1 Multiple Subnetwork Case

The above TAP with deadlines assumes that latencies are all the same. However, this assumption may not be reasonable if an agent visits sites in different subnetworks, such as the USA and Japan. This case can be modeled by variable latencies which are constant within subnetworks and across two subnetworks, as modeled in Assumption 2.

Theorem 14 *The relevant sites belong to two subnetworks, S_1 and S_2. Sites in S_i are s_{ij} where $1 \leq j \leq n_i$. There are three latencies: $L_1, L_2, L_{12} \in Z^+$. For $s_{1j} \in S_1, s_{2k} \in S_2$, $l_{1j2k} = l_{2k1j} = L_{12}$ while for $s_{1j}, s_{1k} \in S_1$, we have $l_{1j1k} = l_{1k1j} = L_1$. Similarly, for $s_{2j}, s_{2k} \in S_2$, we have $l_{2j2k} = l_{2k2j} = L_2$. Probabilities, $p_{mj} > 0$ are nonzero and independent as before. Computation times $t_{mj} \in Z^+$ are arbitrary and nonzero. Latencies between the home site, s_0, and sites in S_i are L_{0i}. The agent must finish its task by visiting sites and returning to the home site before the respective deadlines. The optimal tour for this Two Subnetwork Traveling Agent Problem (TSTAP) with deadlines maximizes the probability of success without exceeding a deadline D_i at each site i. (The deadline time D_i is measured from the time the agent leaves the home site.) Such an optimal tour can be computed in pseudo-polynomial time using a dynamic programming algorithm.*

Proof – Let the time required to visit all sites preceding and including $i_{n'}$ for a tour $T'(n') =< i_1, i_2, ..., i_{n'} >$ where $n' < n$ and to return to the home site be:

$$C_{T'(n')} = l_{01} + t_{i_1} + \sum_{k=2}^{n'} (l_{i_{k-1}i_k} + t_{i_k}) \tag{4.19}$$

where l_{ij} stands for the latency between nodes i and j.

The above $C_{T'(n')}$ can be no greater than the deadline $D_{i_{n'}}$ at the site $i_{n'}$ and, in addition, the total time for all tours $T'(k)$ for $k < n'$ should not exceed the deadline D_{i_k}. In fact, by contrast with Theorem 13, the order of visiting sites affects the total time $C_{T'}$ because the value of $l_{i_{k-1}i_k}$ depends on the subnetworks that sites i_{k-1} and i_k belong to. This means that the order of subnetworks that sites belong to will affect the total time $C_{T'(n')}$, not the order of sites within a subnetwork. The probability of success for the tour is constant for a set of sites.

The tour which maximizes the probability of successful completion while satisfying deadlines can be computed in pseudo-polynomial time using dynamic programming as follows.

First, we will see that sites in a subnetwork has to be visited in increasing order of their deadlines for the optimal answer. Now, all the sites in each subnetwork are sorted in increasing order of their deadlines. Without loss of generality, let the sorted sequence be $T_{S_m} =< m_1, m_2, ..., m_{n_m} >$ for subnetwork S_m. This sorted sequence will be merged with another sorted sequence from another subnetwork (We call this sequence the original sequence) when we construct a tour later. Suppose

that in the merged sequence, we can create other sequences by reordering sites from the same subnetwork without altering the interleaving pattern. Among all possible such sequences, the original sequence has the minimum maximum lateness, as explained in the proof of Theorem 13 [29].

Next, we construct four 3-dimensional arrays of size n_1 by n_2 by $\min(B, D_0 - L_{10}, D_0 - L_{20})$ where D_0 is the deadline at the home machine, $B = \sum_{k=1}^{n_1} t_{k1} + \sum_{k=1}^{n_2} + (n_1 + n_2 - 1) \cdot L_{12} + 2 \cdot \max(L_{01}, L_{02})$ (maximum time for visiting all sites in the worst case) and $D_0 - L_{m0}$ represents the deadline that an agent should leave the last site in the subnetwork S_m for its home machine.

For integer $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$, let $e_m(i, j, s) = 1$ in each subnetwork S_m if there is a subset of $\{1, 2, .., i\}$ and $\{1, 2, .., j\}$ (sorted in increasing order of the deadline for the minimum maximum lateness) for which the total maximum time (visiting sites and moving to the subnetwork S_m) is exactly s and each site of which can be visited no later than its deadline. If such a subset does not exist, $e_m(i, j, s) = 0$ so e_m is Boolean. We also define the array value $p_m(i, j, s) = 0$ if $e_m(i, j, s) = 0$ and $p_m(i, j, s) = 1 - \prod_{k \in T_{e_m(i,j,s)}} (1 - p_k)$ if $e_m(i, j, s) = 1$. Here $T_{e_m(i,j,s)}$ is the set of sites which can be visited in time s without violating their deadline constraints. $p_m(i, j, s)$ is the maximum probability of success among subsets that can be $T_{e_m(i,j,s)}$.

The values of $e_m(i, j, s)$ and $p_m(i, j, s)$ are obtained by dynamic programming. The dimensions of the arrays e_m and p_m are both n_1 by n_2 by $min(B, D_0 - L_{m0})$. They are all initialized to be 0.

We start the algorithm with row $i = 1$ and $j = 0$ for e_1 and $i = 1$ and $j = 0$ for e_2, and proceed to calculate the following rows of e_m from the previous rows. For $i = 1$ and $j = 0$, $e_1(1, 0, s) = 1$ if and only if either $s = 0$, or $s = L_{01} + t_{1_1}$ and $s \leq D_{1_1}$. If $e_1(1, 0, s) = 0$, or $e_1(1, 0, s) = 1$ and $s = 0$, $p_1(1, 0, s)$ is set to be 0. For $i = 0$ and $j = 1$, $e_2(0, 1, s) = 1$ if and only if either $s = 0$, or $s = L_{02} + t_{2_1}$ and $s \leq D_{2_1}$. If $e_2(0, 1, s) = 0$, or $e_2(0, 1, s) = 1$ and $s = 0$, $p_2(0, 1, s)$ is set to be 0.

For $i \geq 2$, we set the value of $e_1(i, j, s) = 1$ if and only if (a) $e_1(i-1, j, s) = 1$, (b) $e_1(i-1, j, s - L_1 - t_{1_i}) = 1$ where $s \leq D_{1_i}$ and $L_1 + t_{1_i} < s$, or (c) $e_2(i-1, j, s - L_{21} - t_{1_i}) = 1$ where $s \leq D_{1_i}$ and $L_{21} + t_{1_i} < s$. For $j \geq 2$, we set the value of $e_2(i, j, s) = 1$ if and only if (a) $e_2(i, j-1, s) = 1$ (b) $e_2(i, j-1, s - L_2 - t_{2_j}) = 1$ where $s \leq D_{2_j}$ and $L_2 + t_{2_j} < s$, or (c) $e_1(i, j-1, s - L_{12} - t_{2_j})$ where $s \leq D_{2_j}$ and $L_{12} + t_{2_j} < s$.

In the above, we construct a tour with total cost s in the case of failure as follows:
In case a tour finishes at the subnetwork 1:

1. There is already a tour involving sites $\{1, 2, .., i-1\}$ and $\{1, 2, ..., j\}$ and then finishing at the subnetwork 1 with total time s (the case $e_1(i-1, j, s) = 1$), each site of which has to be visited no later than its deadline;

2. There is a nonempty tour with total time $s - t_{1_i} - L_1$, involving a subset of sites $\{1, 2, ..., i-1\}$ and $\{1, 2, ..., j\}$ and finishing in the subnetwork 1. Then, we can add site i to that tour if $s \leq D_{1_i}$ (this is the case, $e(i-1, j, s - t_{1_i} - L_1) = 1$ and $t_{1_i} + L_1 \leq s$ where $s \leq D_{1_i}$) ;

3. There is a nonempty tour with total time $s - t_{2_j} - L_{21}$ involving a subset of sites $\{1, 2, ..., i-1\}$ and $\{1, 2, ..., j\}$ and finishing in the subnetwork 1. Then, we can add site i to that tour if $s \leq D_{2_i}$ (this is the case, $e(i, j-1, s - t_{2_j} - L_{21}) = 1$ and $t_{1_j} + L_{21} \leq s$ where $s \leq D_{2_j}$) ;

4. $L_{01} + t_{1_i} = s$ and so we have a tour with only site 1_i on it if $s \leq D_{1_i}$.

In case a tour finishes in the subnetwork 2:

1. There is already a tour involving sites $\{1, 2, .., i\}$ and $\{1, 2, ..., j-1\}$ and then finishing at subnetwork 2 with total time s (the case $e_2(i, j-1, s) = 1$), each site of which has to be visited no later than its deadline;

2. There is a nonempty tour with total time $s - t_{1_i} - L_{21}$ involving a subset of sites $\{1, 2, ..., i-1\}$ and $\{1, 2, ..., j\}$ and finishing in the subnetwork 2. Then, we can add the site i to that tour if $s \leq D_{1_i}$ (this is the case, $e(i-1, j, s - t_{1_i} - L_{21}) = 1$ and $t_{1_i} + L_{21} \leq s$ where $s \leq D_{1_i}$) ;

3. There is a nonempty tour with total time $s - t_{2_j} - L_2$ involving a subset of sites $\{1, 2, ..., i\}$ and $\{1, 2, ..., j-1\}$ and finishing in the subnetwork 2. Then, we can add the site i to that tour if $s \leq D_{2_j}$ (this is the case, $e(i, j-1, s - t_{2_j} - L_2) = 1$ and $t_{2_j} + L_2 \leq s$ where $s \leq D_{2_j}$) ;

4. $L_{02} + t_{2_j} = s$ and so we have a tour with only the site 2_j on it if $s \leq D_{2_j}$.

In the second and third cases for both subnetwork 1 and 2, we check if we can append site i or j at the end of a tour by comparing s and D_{1_i}, or s and D_{2_j}, respectively. It is obvious that a previous tour to which a new site i or j will be appended does not exceed any deadline of the sites in the tour, that is, the maximum lateness of the tour is 0. Now if we append site 1_i or 2_j at the end of the tour, the total time becomes s. The maximum lateness of sites in the new tour is $\max\{0, s - D_{1_i}\}$ or $\max\{0, s - D_{2_j}\}$ for the cases where the site 1_i is added or where the site 2_j is

added, respectively. Since the maximum lateness is the minimum among all possible permutations of sites in same subnetworks, it is impossible to decrease the maximum lateness by changing the order of sites in the same subnetworks. Thus, site 1_i cannot be added to the tour that satisfies $e_1(i,j,s) = 1$ if $s > D_{1_i}$. In the same way, 2_j cannot be added to the tour that satisfies $e_2(i,j,s) = 1$ if $s > D_{2_j}$.

We also define $p(i,j,s)$ as follows. If $e(i,j,s) = 0$, $p(i,j,s) = 0$. We handle the cases for which $e(i,j,s) = 1$ in the following way:

In case a tour finishes in subnetwork 1:

1. There is a tour already involving a subset of $\{1,2,...,i-1\}$ and $\{1,2,...,j\}$ and finishing at the subnetwork 1 with a total time of s, each node of which has to be visited no later than its deadline but i cannot be added to any previous tour to get a new tour (case for which $e_1(i-1,j,s) = 1$ but $e_1(i-1,j,s-t_{1_i}-L_1) = 0$): $p_1(i,j,s) = p_1(i-1,j,s)$;

2. There is no tour with total time s based on sites $\{1,2,...,i-1\}$ and $\{1,2,...,j\}$ but there is one when s_i is added to a tour (case when $e_1(i-1,j,s) = 0$ and $e_1(i-1,j,s-t_{1_i}-L_1) = 1$): $p(j,s) = p_i$ if $s \leq D_{1_i}$;

3. There is already a tour involving a subset of $\{1,2,...,i-1\}$ and $\{1,2,...,j\}$ with a total time of s and other tours with total time $s - L_1 - t_{1_i}$ or $s - L_{21} - t_{1_i}$ using sites $\{1,2,...,i-1\}$ and $\{1,2,...,j\}$ which has to be visited no later than their respective deadlines (the case when $e_1(i-1,j,s) = 1$, and $e_1(i-1,j,s-t_{1_i}-L_1) = 1$ and $s \leq D_{1_i}$ or the case when $e_2(i-1,j,s) = 1$, and $e_2(i-1,j,s-t_{2_j}-L_{12}) = 1$ and $s \leq D_{2_j}$): $p_1(i,j,s) = \max\{p_1(j-1,s), 1-(1-p_{1_i})(1-p_1(j-1,s-L_1-t_{1_i})), 1-(1-p_{1_i})(1-p_2(i-1,j,s-L_{21}-t_{1_i}))\}$;

4. $s = L_{01} + t_{1_i}$, $e_1(i-1,j,s) = 1$ and $s \leq D_{1_i}$ in which case $p_1(i,j,s) = \max\{p_1(i-1,j,s), p_{1_i}\}$.

In case a tour finishes in subnetwork 2:

1. There is a tour already involving a subset of $\{1,2,...,i\}$ and $\{1,2,...,j-1\}$ and finishing at the subnetwork 2 with a total time of s, each node of which has to be visited no later than its deadline but j cannot be added to any previous tour to get a new tour (case for which $e_2(i,j-1,s) = 1$ but $e_2(i,j-1,s-t_{2_i}-L_2) = 0$): $p_2(i,j,s) = p_2(i,j-1,s)$;

2. There is no tour with total time s based on sites $\{1,2,...,i\}$ and $\{1,2,...,j-1\}$ but there is one when s_j is added to a tour (case when $e_2(i,j-1,s)=0$ and $e_2(i,j-1,s-t_{2_j}-L_2)=1$): $p(i,s)=p_j$ if $s \le D_{2_j}$;

3. There is already a tour involving a subset of $\{1,2,...,i\}$ and $\{1,2,...,j-1\}$ with a total time of s and another tour with total time $s-L_2-t_{2_j}$ or $s-L_{12}-t_{2_j}$ using sites $\{1,2,...,i\}$ and $\{1,2,...,j-1\}$ which has to be visited no later than their respective deadlines (the case when $e_2(i,j-1,s)=1$, and $e_2(i,j-1,s-t_{2_j}-L_{22})=1$ and $s \le D_{2_j}$ or the case when $e_2(i,j-1,s)=1$, and $e_1(i,j-1,s-t_{2_j}-L_{12})=1$ and $s \le D_{2_j}$): $p_2(i,j,s) = \max\{p_2(i,j,s), 1-(1-p_{2_j})(1-p_2(i,j-1,s-L_2-t_{2_j})), 1-(1-p_{2_j})(1-p_1(i,j-1,s-L_{21}-t_{2_j}))\}$;

4. $s=L_{02}+t_{2_j}$, $e_2(i,j-1,s)=1$ and $s \le D_{2_j}$ in which case $p_2(i,j,s)=\max\{p_2(i,j-1,s), p_{2_j}\}$.

This handles all cases and clearly requires only a single scan of each row, therefore about $O(\min(D_0-L_{m0},B)$ operations per row for $n_1 \cdot n_2$ rows yielding a complexity of $O(\min(D_0-L_{m0},B)\cdot n_1 \cdot n_2)$ steps where m stands for the subnetwork S_m. D_0-L_{m0} is the deadline by which an agent has to leave the last site for its home site 0.

The subset which maximizes the probability of the agent's success and whose sites can be visited no later than their respective deadlines is the one that maximizes $p_m(n_1,n_2,s)$ for $s \le D_0-L_{m0}$ (recall that $p(j,s)=0$ if $e(j,s)=0$). Let $e_m(n_1,n_2,s)$ with the maximum probability be $e_m(n_1,n_2,s')$.

By tracing backward from $e_m(n_1,n_2,s')$ toward $e_m(0,0,0)$, we can obtain the tour that maximizes the probability of success without violating deadline constraints. However, this tour cannot guarantee the minimum expected time in visiting all the sites no later than their respective deadlines.

It takes at most $n_m \log n_m$ to sort the subset in the subnetwork S_m in increasing order of deadlines. Therefore, the optimal solution can be obtained in time that is polynomial at most in n_1, n_2 and B, $O(4\cdot n_1 \cdot n_2 \cdot \min\{\sum_{k=1}^{n_1} t_{1_k} + \sum_{k=1}^{n_2} t_{2_k} + (n_1+n_2-1)\cdot L_{12} + 2\cdot L_{m0}, D_0-L_{m0}\} + n_1 \log n_1 + n_2 \log n_2)$.

The only remaining issue is that of maintaining the list of sites on a tour corresponding to $e_m(i,j,s)=1$ and the value of $p_m(i,j,s)$ recorded in the array. This can clearly be handled within the time bounds we have demonstrated. Q.E.D.

4.4.2 Multiple Agents

As in TAP without deadlines, we consider a planning problem where multiple agents cooperate to complete a same task. Intuitively, the maximum probability of success will be larger than in the single agent case because the chance that agents can visit sites before their deadlines increases. In this subsection, we deal with the multiple agent planning problem with deadlines, i.e., the Traveling Multiple Agent Problem (TMAP) with Deadlines. Formally, we define the Traveling Multiple Agent Problem (TMAP) with Deadlines as follows:

Theorem 15 *Assume that TMAP with Deadlines has a the latency* $l = l_{ij} = l_{km} \in Z^+$ *for all* i, j, k, m. *Computation times* $t_i \in Z^+$ *are arbitrary but should be nonnegative integers. Probabilities of success* $p_i \geq 0$ *are arbitrary and independent. Each site* i *(including the home site* s_0*) has a deadline,* D_i*, by which an agent must visit the site or else the site becomes unavailable. There are two agents working on a same task. The optimal schedule for this TMAP maximizes the probability of success without exceeding a deadline at each site. Such an optimal schedule can be computed in pseudo-polynomial time using a dynamic programming algorithm.*

Proof – First, we recall the second step of proof for Theorem 13, which shows that the sites should be visited in increasing order of deadline. Since the proof assumes a single agent, more precisely, we should restate here that the sites visited by the same agent should be sorted in increasing order of deadline.

Next, we introduce two arrays $e(i, s_1, s_2)$ and $p(i, s_1.s_2)$ of size n (number of sites) by $\min(B, D_0 - l)$ where $B = \sum_{k=1}^{n} t_k + n \cdot l$ (total time for visiting all sites in the worst case) and $D_0 - l$ represents the deadline that an agent should leave the last site for its home machine.

For integer i, where $1 \leq i \leq n$, let $e(i, s_1, s_2) = 1$ if there is a subset of $\{1, 2, ..., i\}$ for which the total maximum time by agent 1 and agent 2 are exactly s_1 and s_2, respectively, and each site of which can be visited no later than its deadline. If such a subset does not exist, $e(i, s_1, s_2) = 0$. Another array $p(i, s_1, s_2)$ is defined as follows: $p(i, s_1, s_2) = 0$ if $e(i, s_1, s_2) = 0$ and $p(i, s_1, s_2) = 1 - \prod_{k \in T_{e(i,s_1,s_2)}} (1 - p_k)$ if $e(i, s_1, s_2) = 1$ where $T_{e(i,s_1,s_2)}$ is the set of sites whose times spent by agent 1 and agent 2 are s_1 and s_2 respectively. This subset of sites has to be the maximum probability of success among all such subsets and each of its sites can be visited no later than the site's deadline. That probability is precisely what is stored in $p(i, s_1, s_2)$.

The values of $e(i, s_1, s_2)$ and $p(i, s_1, s_2)$ are obtained by dynamic programming. They are all initialized to be 0.

We start the algorithm with row $i = 1$ and proceed to calculate the following rows of e from the previous rows. For $i = 1$, $e(1, s_1, s_2) = 1$ if and only if

1. $s_1 = 0$ and $s_2 = 0$;
2. $s_1 = l + t_1$, $s_1 \leq D_1$ and $s_2 = 0$;
3. $s_2 = l + t_1$, $s_2 \leq D_2$ and $s_1 = 0$.

$p(i, s_1, s_2) = 0$ in the first case. $p(i, s_1, s_2) = p_1 = 1 - (1 - p_1)$ in the second and third cases.

For $i \geq 2$, we set the value of $e(i, s_1, s_2) = 1$ if and only if

1. $e(i-1, s_1, s_2) = 1$;
2. $l + t_i \leq s_1$ and $e(i-1, s_1 - l - t_i, s_2) = 1$ where $s_1 \leq D_i$;
3. $l + t_i \leq s_2$ and $e(i-1, s_1, s_2 - l - t_i) = 1$ where $s_2 \leq D_i$.

We also define elements of the ith row of p as follows. If $e(i, s_1, s_2) = 0$, $p(i, s_1, s_2) = 0$. We handle the cases for which $e(i, s_1, s_2) = 1$ in the following way:

1. If $e(i-1, s_1, s_2) = 1$, but $e(i-1, s_1 - t_i - l, s_2) = 0$ or $e(i-1, s_1, s_2 - t_i - l) = 0$: $p(i, s_1, s_2) = p(i-1, s_1, s_2)$;
2. If $e(i-1, s_1, s_2) = 0$, $e(i-1, s_1 - t_i - l, s_2) = 1$, and $s_1 \leq D_i$: $p(i, s_1, s_2) = p_i$;
3. If $e(i-1, s_1, s_2) = 0$, $e(i-1, s_1, s_2 - t_i - l) = 1$, and $s_2 \leq D_i$: $p(i, s_1, s_2) = p_i$;
4. If $e(i-1, s_1, s_2) = 1$, $e(i-1, s_1 - t_i - l, s_2) = 1$, and $s_1 \leq D_i$: $p(i, s_1, s_2) = \max\{p(i-1, s_1, s_2), 1 - (1 - p_i)(1 - p(i-1, s_1 - l - t_j, s_2))\}$;
5. If $e(i-1, s_1, s_2) = 1$, $e(i-1, s_1, s_2 - t_i - l) = 1$, and $s_2 \leq D_i$: $p(i, s_1, s_2) = \max\{p(i-1, s_1, s_2), 1 - (1 - p_i)(1 - p(i-1, s_1, s_2 - t_i - l))\}$;
6. If $s_1 = l + t_i$, $e(i-1, s_1, s_2) = 1$ and $s_1 \leq D_i$: $p(i, s_1, s_2) = \max\{p(i-1, s_1, s_2), p_i\}$;
7. If $s_2 = l + t_i$, $e(i-1, s_1, s_2) = 1$ and $s_2 \leq D_i$: $p(i, s_1, s_2) = \max\{p(i-1, s_1, s_2), p_i\}$.

This handles all cases and clearly requires only a single scan of each row whose size is $(\min(D_0 - l, B))^2$, therefore about $O((\min(D_0 - l, B))^2)$ operations per row for n rows yielding a complexity of $O((\min(D_0 - l, B))^2 \cdot n)$ steps.

The subset which maximizes the probability of the agent's success and whose sites can be visited no later than their respective deadlines is the one that maximizes $p(n, s_1, s_2)$ for $s_1, s_2 \leq D_0 - l$ (recall that $p(i, s_1, s_2) = 0$ if $e(i, s_1, s_2) = 0$). $D_0 - l$ is the deadline by which time an agent has to leave the last site for its home site 0.

By visiting all the sites in that subset in increasing order of deadline, no deadline is violated. However, as discussed before, this order cannot guarantee the minimum expected time to visit all sites no later than their respective deadlines.

It takes at most $n \log n$ to sort the subset in increasing order of deadline. Therefore, the optimal solution can be obtained in time that is polynomial at most in n and B, $O(2 \cdot n \cdot (\min\{\sum_{k=1}^{n} t_k + n \cdot l, D_0 - l\})^2 + n \log n)$. Q.E.D.

The above algorithm can be applied to the Traveling Multiple Agent Problem with Deadlines even when the number of agents m is larger than 2. The size of the two arrays introduced in the proof will become $n \cdot (\min\{\sum_{k=1}^{n} t_k + n \cdot l, D_0 - l\})^m$, while the construction of the arrays is the same as the two agent case. Thus, the complexity of the algorithm for TMAP with Deadlines where $m \geq 2$ is:

$$O(2 \cdot n \cdot (\min\{\sum_{k=1}^{n} t_k + n \cdot l, D_0 - l\})^m + n \log n).$$

Chapter 5

Implementation

In the previous chapter, we developed algorithms and theories for a specific planning problem that is often observed in mobile-agent applications such as information retrieval [47] and data-mining [20, 21]. This chapter presents the architecture of our implemented mobile-agent planning system, where these algorithms and theories are applied.

The mobile-agent system we employ is *D'Agents*, which was developed at Dartmouth College[26, 27]. The current version of D'Agents supports Tcl/Tk, Java, Scheme and Python. The main attractive features of D'Agents are its ease of use (e.g., simple and high-level) and its effective security mechanisms.

Our planning system is built as one of the utilities used in D'Agents. With the planning system, the mobile agents in D'Agents can perform some tasks with minimum execution time. Thus, although the planning system will be transparent to an end user, its contribution to the performance of mobile agents will be significant.

This section begins with an overview of the mobile-agent planning system, which has four main components: the Planning module, the Network Sensing module, the Clustering module and the Directory Service module. Detailed explanations of each component follow in the rest of the section.

5.1 Overview of Mobile Agent Planning System

Since the mobile-agent planning system is built on top of D'Agents, we start with a description of D'Agents before explaining the mobile-agent planning system. D'Agents is a mobile-agent system that supports agents written in Tcl/Tk, Java, Scheme and Python. The system provides migrations,

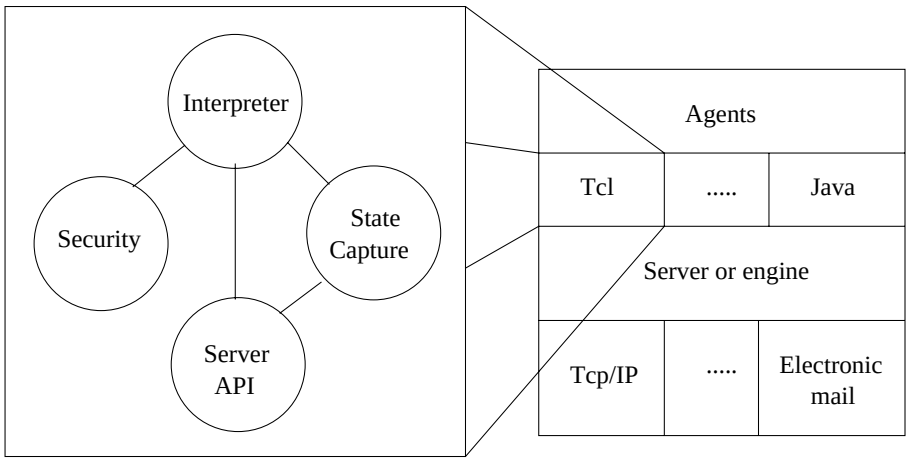

Figure 5.1: The D'Agents' architecture

low-level communication, and security mechanisms. The main component of D'Agents is a server that runs on each machine and executes mobile-agent commands. Its most important function is to support migration, specifically, the command *agent_jump* that allows an agent to migrate to a new machine during its execution. The sever on the agent's current machine captures the complete state of the agent and sends the information to the server on the destination machine. The destination server starts up an execution environment, loads the received state information into this environment, and resumes the agent's execution from the point where its execution was suspended on the previous machine. The server also handles low-level communication (message passing and binary streams between agents) and security mechanism. All the migration and communication commands interpreted by the server are simple and high-level, and the security mechanisms are transparent to the user.

The D'Agents' architecture is depicted in Fig.5.1. The lowest level of the architecture is an interface to each available transport mechanism. The second level is the server that performs several tasks: (1) command interpreter, (2) state capture during migration, and (3) authentication of incoming agents for security. The third level of the architecture consists of the execution environments for each supported agent language. The last level are the agents themselves.

The biggest advantages of mobile agents versus conventional distributed-computing paradigms are their efficient use of bandwidth and their robust performance in the presence of unreliable

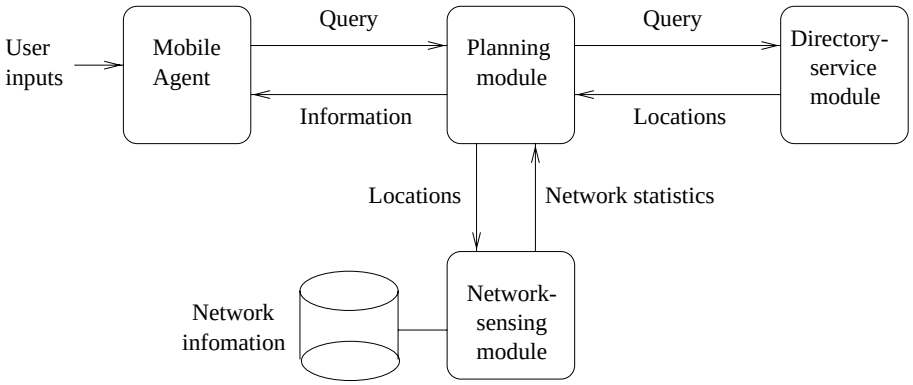

Figure 5.2: The architecture of the planning system

network connections. By sending a mobile agent to a remote database and getting only a small result back after its execution there, a significant amount of data transmission can be avoided. Since the mobile agent does not require a constant connection with the remote host during its execution, it can perform robustly even when periods of network disconnection occur frequently. Because of such advantages, mobile agents are suitable for applications, such as information retrieval and data-mining, that require access to a large amount of remote data in a possibly unreliably connected network such as a wireless network.

In information retrieval and data-mining applications, a mobile agent needs a planning system. In those applications, it often happens that a mobile agent cannot complete its task, e.g., locate the desired information, in the first database that it happens to visit. So the mobile agent has to try several databases in the network until it finds its target information. Random migration to these database will lead to inefficient performance. Instead we need a planning system that can decide where an agent should go next so that the agent can perform better.

The architecture of our planning system for mobile agents is depicted in Fig.5.2. The planning system consists of four main components: planning module, network-sensing module, clustering module and directory-service module. In our system, when a mobile agent is given a task, for example, searching for information, it consults with the planning module first. The planning module asks a directory-service module for possible locations where the mobile agent might complete its task, e.g., where it can find the desired information. The directory-service module returns a list of locations with relevance rankings, just like Internet search engines such as *Infoseek* and *Altavista*.

The relevance ranking is treated as the probability of success, i.e., the probability that the mobile agent will find the desired information, thus completing its task. Although we have not provided a function to measure this probability in the implemented system, we assume that this probability is measurable. For example, the probability can be the ratio of data cached at a proxy server over the full amount of data available at the actual server.

After obtaining the list of machines and their relevance rankings, the planning module passes the list to the network-sensing module, which captures the latencies between those machines and their CPU load. The network-sensing module keeps track of these network statistics by probing the network at fixed intervals.

As soon as the network statistics are returned to the planning module, the sequence in which agents should visit machines (to minimize its total expected execution time) is calculated from the network statistics and probabilities of success. The calculation is done using the algorithms and theorems described in the previous chapter. The clustering module forms subnetworks based on the latencies collected by the network-sensing module for the approximations used in the algorithms and theorems.

Each module is described in detail in the following subsections.

5.2 Planning Module

The planning module is the main component of the planning system. It is directly consulted by the mobile agents and after collecting the necessary information from the other modules, it decides the sequence in which machines should be visited.

The planning module is implemented as an Agent Tcl script so that it can communicate easily with its client agents. Pseudo-code for this planning agent is in Fig.5.3.

The planning agent is idle until it receives a query from a mobile agent for an optimal machine-visit sequence. As soon as receives a query, it forwards the query to the directory-service agent and waits for the reply containing the list of machines where the mobile agent might be able to find the desired information. The directory service also returns the probability of success for each machine. The list of machines is forwarded to the network-sensing agent to get the relevant network statistics (such as latencies between the machines and their CPU usage.)

```
# create a planning agent
agent _begin
agent _name Planning _agent

while (1)  {
   # receive a query from a mobile agent
   set Mobile_Agent [agent_receive query -blocking]

   # send the query to the Directory-service Agent
   agent_send Directory_Service query

   # receive the locations and probability of success
   # from the Directory Service Agent
   agent_receive locations probability -blocking

   # send the locations to the Network Sensing Agent
   # to get the current network statistics
   agent_send Network_Sensing locations

   # receive the network statistics
   agent_receive statistics -blocking

   # decide which  planning algorithm to use
   # (based on the probabilities and network statistics)
   set type [Decide_type probability statistics]

   # execute the chosen planning algorithm to obtain
   # the best sequence in which to visit the machines
   switch type {
     1 : set sequence [Constant_latency locations probability statistics]
     2 : set sequence [Subnetwork locations probability statistics]
         .
         .
   }

   # send the sequence of machines to the requesting mobile agent
   agent_send Mobile_Agent sequence
}
```

Figure 5.3: Pseudo-code for the planning agent

Once the network statistics are returned, the planning agent selects the most suitable planning algorithm depending on the deviation in and the average of the latencies, the probabilities of success, and the estimated computation time of the task at each machine. Note that the estimated computation time is calculated based on the measured CPU load, the benchmarked machine performance, and the size of a task. Algorithm selection is determined by whether the probabilities, latencies, and estimated computation time can be considered relatively constant, since the algorithms developed in the previous chapter assume constant latency, constant probabilities or constant estimated computation time.

Once a suitable algorithm is chosen, the algorithm executes on the latencies, probabilities and estimated computation time to produce the sequence of machine visits that minimizes the agent's expected total execution time. In this calculation, the benchmarked performance of the machines and the size of the mobile agent's task are assumed to be known in advance so that the computation time at each machine can be calculated from its given CPU load.

The sequence of machines is returned to the mobile agent that requested it.

5.3 Directory Service Module

The job of the directory-service module is to maintain an index information resources in the network and tell the planning agent where the agent's desired information can be found. This requires indexing the information, keeping track of the location of information, updating its tables if the location changes and so on, similar to what Internet search engines do for example. The second task is to return the list of machines where the desired information might be located, along with a probability of success for each machine. Since the request from the planning agent can be abstract, e.g., keywords related to the desired information, the directory service cannot specify the exact location of the information with certainty. Thus, it provides the probability of success to rank the possible locations, just as some Internet search engines show the relevance rankings of web pages. Each probability can be regarded as the directory's degree of confidence that the desired information can be found there.

The directory service that we have implemented is simple and handles only the second task above (returning the list of locations with probabilities). Moreover, mechanisms to measure the

```
while (1) {
   # wait until receiving a query from a planning agent
   set Planning_Agent [agent_receive keywords -blocking]

   # open the file that contains the location table
   # for the query (the keywords)
   set fd [open location_file]

   # search for the entry for the keywords in the file
   set locations_with_prob [find keywords $fd]
   close $fd

   # send a list of locations with probabilities of success
   agent_send Planning_Agent locations_with_prob
}
```

Figure 5.4: Pseudo-code for the directory service agent

probabilities of success are assumed to be available but are not implemented yet. One simple approach is to take the ratio of the amount of information at the site (e.g. number of bytes) over the the total amount of information in the network. Anyway, our directory service needs significant extensions before it can be used in a production-environment, but it is sufficient for testing the planning services.

Our directory service is implemented in Agent Tcl as a stationary agent. Pseudo-code for the directory service agent is shown in Fig.5.4.

The directory service agent waits until it receives a query, e.g., keywords, from the planning agent, which needs the list of locations where a mobile agent can find the desired information. As soon as the directory service agent receives the query, it opens the file that contains the location table, which associates locations with keywords. The location table also contains the probabilities of success, i.e., the probabilities that the agent can find the desired information at the associated locations. The list of locations for the keywords given by the planning agent are retrieved from the file and returned to the planning agent after closing the file.

Note that, as mentioned above, our directory service lacks the ability to create and manage the location table. Thus, the file that contains the table is created by hand in advance.

5.4 Network-Sensing Module

The network-sensing module periodically collects network statistics such as latency, available bandwidth, current CPU load, and packet loss rate, accepts queries about these statistics from the planning agent, and answers those queries.

Collecting network statistics or, in other words, *network sensing*, is a very important topic for distributed applications and mobile-agent applications. Whether the network statistics are accurate or not significantly affects the performance of those applications. For example, if a mobile agent tries to migrate to a machine that has been shutdown, it will waste a large amount of time. However, accurate network sensing is very difficult to achieve without introducing a significant amount of traffic into the network. Several different network-sensing schemes have been proposed in the literature to overcome the difficulty.

This section overviews the state-of-the-art in network sensing, and then presents the architecture of our accurate and efficient network sensing system along with a description of its implementation.

5.4.1 Literature of Network Monitoring

Availability of accurate network statics will be necessary for the robust and efficient performance of distributed and mobile-agent applications. However, achieving accurate network monitoring is difficult without transmitting a huge amount of additional network traffic. The more updated and accurate information a network-sensing system collects, the more traffic it has to generate.

Many researchers have responded to this problem by proposing various network-sensing systems [2, 39, 52]. The goal of all these existing network-sensing systems is to minimize the extra network traffic generated due to the network-sensing. These systems can be classified based on three criteria [7].

One criteria is the amount of traffic generated during the sensing process. In a passive sensing process, the network sensors piggy-back status information onto existing messages, while in an active-sensing process, network measurements are done by sending additional control messages [45]. Although the first process generates less traffic, it may not be able to collect comprehensive network status information if messages are not being sent between some hosts.

The second criteria is the frequency of the network-sensing process. In an on-demand sensing

process, sensing occurs only when an application requests status information about a specific resource, while in the continuous-sensing process, sensing occurs continuously and applications are informed of changes in the network status information as soon as those changes occur [39]. The first scheme can not provide the network status information on demand when it is requested like the second scheme, though the first scheme is more efficient because it collects information only when necessary.

The third criteria is how the sensing process is controlled. In the centralized sensing process, all of the network status information is collected and stored by one host, and this information is requested by other hosts. In the decentralized sensing process, distributed monitors collect only local network information and request non-local information on demand from other monitors on other hosts [2]. The first scheme has robustness and scalability problems. When the machine where the network sensing process is running is down, the network statistics become unavailable. Moreover, the information managed by the centralized monitor grows quadratically as the number of hosts in network increases, simply because the number of latency and bandwidth values is n^2 where n is the number of hosts. The replication of the stored information to other machines may be the solution to the first problem, but it causes a larger scalability problem because of having duplicated information in the network. The second scheme does not suffer from these problems, but complex control for collaboration between distributed monitors is required if an application needs to access non-local network status information or status information for the entire network.

5.4.2 Architecture of the Network Sensing System

The network-sensing module (network monitoring system) that we implemented is classified as an active, continuous and centralized system according to the above three criteria. Our system collects network status information on a periodic basis by sending control messages to measure (1) the latencies between hosts and (2) the current CPU load of hosts in the network. All the network status information is managed by the central network sensing server, which is assisted by a network sensing slave on each host. Each slave collects local information and reports it to the central server when it is requested.

This active, continuous and centralized network-sensing system should be suitable to our planning system. In the planning system, access to the network status information by the planning agent

could be comprehensive in some case or, at least, non-local in most of the cases, and furthermore, the information should be sent to the agent on demand.

Our network-sensing system has a special feature, which other existing systems do not, to solve the scalability problem when collecting latencies between hosts in a network. Usually, latency is measured using mechanism similar to that of the standard latency measurement tool *ping*. However, with ping measurement of all latencies between n hosts in the network requires n^2 packet transmissions, leading to a huge amount of traffic if n is large. Our system successfully reduces this traffic using multicast packets. In the best case, the order of the amount of the traffic in our system is $O(n)$, rather than the $O(n^2)$ of the conventional systems. The following subsection explains this in detail.

Latency Measurement with Multicast Packet

Our network sensing system uses multicast packets to reduce the network traffic when measuring latencies. This subsection explains the mechanism in detail. It starts with an explanation of the conventional method to measure latencies, and then explains our method and analyzes the performance gain.

The conventional method for measuring the latency between two machines is *ping*. *Ping* sends an Internet Control Messaging Protocol (ICMP) message to the destination machine. This message includes a timestamp, i.e., the time at which the message was sent. When the packet is returned from the destination, *ping* calculates the round trip time by taking the difference between the receive time and the time of the timestamp. Half of this round trip time is the latency.

If latencies are measured among n machines in the network, in a conventional system, each machine must send the ping packet to the rest of the machines ($n-1$ machines). Thus, the amount of messages sent for measuring latencies in the entire network will be $n \times (n-1)$, which quadratically grows as n becomes larger.

Although we can send a multicast packet with *ping* to measure latencies between a given source and multiple destinations, its usage in the network sensing system will not significantly reduce the amount of traffic. The multicast packet is sent only in one way direction, from the source to the destination. So the number of returning packets will be still $n-1$, which is the same as with a normal ping.

Our method takes more step to reduce the network traffic. Our method measures only one way trip time instead of the round trip time measured by ping or multicast ping.

Each machine sends a multicast packet to the rest of the machines in the network, timestamped with the time at which it left the source. When the destination machine receives the packet, it calculates the one-way trip time, taking the difference between the packet's arrival time and the time on the timestamp. Note that the calculated one way time is not accurate unless the clocks on the destination and source machines are synchronized. The adjustment of the time warp due to unsynchronized clocks is explained later.

Since each machine sends a multicast packet at the same time, it also receives $n-1$ packet from the rest of the machines, and thus, it gets $n-1$ one-way trip times. All these one-way trip times with their source machine addresses are sent in one packet to the central network-sensing server that manages the network-status information.

Then the central network sensing server looks for the pair of one-way-trip times between two machines. Obviously, this pair consists of two one-way-trip times, the direction of which are opposite each other. Adding these two one-way-trip times to create the round-trip time removes time warp due to unsynchronized clocks on the two terminal machines of the trip, which is explained as follows:

> Assume that a one-way-trip time of one direction is the actual one-way-trip time l_{ij} plus the time difference between two clocks δ, that is, $l_{ij}+\delta$. Then another one-way-trip time of the opposite direction will be $l_{ji}-\delta$. The addition of these one-way-trip time makes $l_{ij}+l_{ji}$ where δ disappears.

The latency between the two machines is half of the obtained round-trip time. In the same way, the central network-sensing server can obtain all the latencies between the machines in the network.

All the packets sent to measure latencies in the above scheme are one-way-multicast packets, the number of which is only n. By considering the number of packets sent from each machine to the central network sensing server, i.e., 1, the order of the number of total packets sent in the network is only $O(1 \times n + n) = O(n)$ in the best case, which outperforms the conventional network-sensing systems.

```
while (1) {

   # receive a request of network statistics about "machines"
   set Planning_Agent [agent_receive machines -blocking]

   # open "dbm" file that contains network statistics
   set fd1 [open_dbm $Latency_File]
   set fd2 [open_dbm $CPU_File]
   set fd3 [open_dbm $performance_File

   # search for the statistics of $machines
   set latency [find_dbm $fd1 $machines]
   set CPU [find_dbm $fd2 $machines]
   set perform [find_dbm $fd3 $machines]

   # return the statistics to the planning agent
   agent_send $Planning_Agent $latency $CPU $perform

}
```

Figure 5.5: Pseudo-code of the stationary network sensing agent

5.4.3 Implementation of the Network Sensing System

Our network sensing system consists of a stationary agent written in Agent Tcl and a central network sensing server daemon with network sensing slave daemons running on each machine in the network. Those daemons are written in C. The stationary agent handles all queries from planning agents for network-status information. The central network-sensing server and slave daemons collaborate to periodically collect network-status information.

Pseudo-code of the stationary network sensing agent is described in Fig.5.5.

The task of the stationary network-sensing agent is fairly simple. When the agent receives a request from a planning agent for network-status information about certain machines, it searches for the information in the files that contain latencies, current CPU loads, and benchmarked performance. Then the retrieved network-status information is returned to the planning agent.

The latency, CPU loads, and benchmarked performance information are stored in UNIX *dbm* files by the central network-sensing server. The database of latencies is indexed by a combination of the addresses of two computers, while the databases of current CPU load and performance are indexed by the address of a single machine.

Pseudo code of network sensing server is described in Fig.5.6.

```
#define Freq 30   /* the monitoring frequency */
#define latency_file ``latency.dbm''
#define CPU_file ``CPU.dbm''
#define cluster_file ``cluster.dbm''
#define Machines_List "xxx.xxx.xxx.xxx"  /* Multicast group address */

while (1) {

  /* wait until the next collecting time */
  sleep(Freq);

  /* send a trigger to the network sensing slaves running
                                       on all the machines */
  sendto(sockfd, trigger_packet, Machine_List);

  while (until timeout) {
     /* receive packets from the network sensing slaves */
     recvfrom(sockfd, mesg);

     /* extract the one-way-trip time and CPU load from the packet */
     one_way_time=getOneWayTime(mesg);
     CPU=getCPU(mesg);

     /* store the one-way-trip time in OneWayTime_list */
     addList(OneWayTime_list, one_way_time);

     /* store the CPU load in a database */
     store_db(CPU_file, CPU);
  }

  for (i=0; i < the_number_of_machines; ++i)
    for (j=0; j < the_number_of_machines; ++j) {
      /* calculate the latency using latency pointer list */
      latency=GetRoundTrip(OneWayTime_list, GetAddress(Machines_list,i), \
                                            GetAddress(Machines_list,j));
      /* store the result in the dbm file */
      index=MakeIndex(GetAddress(Machines_list,i), GetAddress(Machines_list,j));
      store_db(tmpLatency_file, index, latency);
    }
  }

  /* Cluster machines based on latencies */
  MakeCluster(tmpLatency_file, Latency_file);
}
```

Figure 5.6: Pseudo-code of the network sensing server

The central network sensing server sends a trigger to the slaves using a multicast packet. The triggers are sent at a frequency defined by the administrator and cause the slaves to collect the status information. When the slaves report back the one-way-trip times and current CPU loads, the one-way-trip times are stored in a pointer list so that it can be scanned to find a matched pair of one-way-trip times later. The current CPU loads are stored right away in the UNIX *dbm* database file indexed by a machine address. If all expected packets arrive, or a certain time has passed, the central network sensing server proceeds to find pairs of one-way-trip times in the pointer list in order to calculate the latencies between machines. Those results are stored in the dbm database file. Then the cluster module is called to create the clusters based on the newly collected latencies. Those clusters are used as subnetworks by the planning algorithms developed in the previous section. Modified latencies based on the cluster information are stored in a latency file referred to by the network-sensing agent.

Pseudo-code of the network sensing slave is given in Fig.5.7.

First, the network sensing slave joins itself to the multicast group using the Internet Group Management Protocol (IGMP) [14], which basically lets a multicast host talk to a multicast router about multicast group membership. Refer to RFC1112 for detail. By sending a packet to the address of the multicast group, a machine can send the packet to all the members in the group.

As soon as the network sensing slave receives a trigger for collecting network status information from the server, it puts a current timestamp in a packet and sends it to the rest of the machines as a UDP packet using the multicast address. Meanwhile, it starts receiving UDP packets from the rest of the machines. Those packets contain their departure time from their sources in their timestamp. By collecting the arriving time of the packets on the local machine, the slave obtains the one-way-trip time of the packets and appends it at the end of the packet that will be reported to the network-sensing server. When all the expected packets arrive from the rest of machines or a certain time (defined by a user) is up, the slave proceeds to measure the CPU load of the local machine and append it to the end of the packet and returns it to the server. Note that if some of the expected packets does not arrived at the slave within the certain time, the one-way-trip time of the packets are recorded as infinity.

```
#define MultGroup "xxx.xxx.xxx.xxx" /* Multicast group address */

main() {

  /* join the multicast group */
  JoinMultGroup(MultGroup, local_address);

  while (1) {

    /* wait until receiving a trigger from NSsever */
    recvfrom(sockfd, packet, &NSserver);

    /* get a current time */
    gettimeofday(&current_time);

    /* send a multicast packet to the rest of machines */
    sendto(sockfd, current_time, MultGroup);

    while (until timeout) {

      /* receive a packet from other machines */
      recvfrom(sockfd, send_time, &sender);

      /* get the receiving time */
      gettimeofday(&recv_time);

      /* calculate the one-way-trip time */
      one_way_time=recv_time-send_time;

      /* append it to the end of the report that will be sent to NSserver */
      AppendPacket(report, one_way_time, sender);
    }

    /* measure the CPU load of the local machine */
    GetCPUload(CPU);

    /* append it to the end of the report */
    AppendPacket(report, CPU, local_machine);

    /* send it to NSserver */
    sendto(sockfd, report, NSserver);
   }
}
```

Figure 5.7: Pseudo-code of the network sensing slave

5.5 Clustering Module

The clustering module is activated by the network-sensing server to create clusters based on newly collected latencies. The created clusters are subnetworks that are used in the planning algorithms as shown in subsection 4.2.2 and 4.3.1. Thus, a cluster is a group of machines where all latencies between the machines are approximately constant. Latencies between machines belonging to different clusters are variable. This subsection shows the formulation of the clustering method that we employed, and then the implementation of the clustering module.

The clustering problem in our case is defined as follows:

> *Assume that there is a square matrix M indexed by machines, each entry of which is a latency l_{ij} between machine i and machine j. There is another square matrix M' with the same size. Latencies l'_{ij} between machines in the same cluster, say k, are replaced by the average of all the latencies between machines in the cluster k, i.e.,*
>
> $$l'_{ij} = \sum_{a,b \in C_k,\ a \neq b} l_{ab} / N_{C_k}$$
>
> *where N_{C_k} is the number of latencies between machines in the cluster k.*
>
> *Latencies l'_{ij} between machines located in two different clusters, say cluster k and h, are replaced by the average of all the latencies between those machines as follows:*
>
> $$l'_{ij} = \sum_{a \in C_k,\ b \in C_h} l_{ab} / N_{C_{kh}}$$
>
> *where $N_{C_{kh}}$ is the number of latencies between machines that belong to different clusters k and h.*
>
> *The clustering problem is to minimize the sum of squared differences between entries in those two matrices, i.e.,*
>
> $$S = \sum_{i,j \in N} (l_{ij} - l'_{ij})^2$$
>
> *where N is a set of machines in the network.*

Note that the replacement of a latency by the average value of a latency between machines in the same cluster or across different clusters is derived from the least square error method. The

replacement minimizes the least square differences between entries whose machines belong to a same subnetwork.

The above clustering problem is different from those that have often been seen in the literature. Most of the conventional clustering algorithms minimize the least square difference between only entries whose machines belong to a same cluster, while our clustering approach minimizes the least square difference between all entries in two matrices.

Due to the different character of our clustering problem from the typical clustering problem in the literature, conventional clustering methods [34] cannot be applied to our clustering problem. Instead of treating our problem as a clustering problem, we treat it as an optimization problem where the function S should be minimized by employing *genetic algorithms*[8, 35] or *simulated annealing* [33, 32]. In these two optimization methods, the cost function is expressed as the square difference S between two matrices. The state is expressed as the cluster configuration, and the action is to move a machine from one cluster to another cluster.

Experiments with these algorithms were done in *matlab* [46] for randomly created latency matrixes. The results showed that the genetic algorithm converged faster than the simulated annealing method and obtained solutions of similar quality, which leads us to employ the genetic algorithm in the clustering module.

The clustering module that has the genetic algorithm at its heart is implemented as a daemon in the C language. This module is called by the network sensing server with a parameter, i.e., the name of the file that contains the latencies, after the server collects the new network status information. The clustering method (genetic algorithm) is executed on these latencies in the file, and the clustering result is stored in another file that will be accessed by the network sensing agent when it receives a query from the planning agent.

Chapter 6

Experimental Results

Experimental results demonstrate that the planning algorithms developed in Chapter 3 can be actually implemented as a part of the D'Agents package and are useful in agent planning.

Three series of experiments are described. In these experiments, information retrieval mobile agents must search for information in the network with the assistance of the planning (module) agent. In the first experiment, a single information retrieval agent consults the planning agent (that uses a directory service agent and a network sensing agent) for its itinerary so that it can find the information in the minimum expected time. In the second experiment, multiple information retrieval agents cooperate to locate information in the network with assistance from the planning agent. The last experiment imposes deadlines on the access time of machines in the network. The planning agent assists the information retrieval agent in order to maximize the probability of successfully finding the desired information without violating deadline constraints.

For a sake of comparison with planning algorithms derived in this thesis, two greedy algorithms are employed, one of which is based on the probability of success and the other on the estimated computation time at each machine. Note that the estimated computation time at each machine is obtained based on its current CPU load, its benchmarked CPU performance, and the estimated size of a task.

6.1 Experimental Setup

6.1.1 Overview of the Experiments

In three sets of experiments, information retrieval agents are executed and their execution times are measured. The task of these agents is to open a certain text file (the size is 234KB) in a text database on a machine and parse the file entirely to satisfy a given query by a user. The success of finding the information on a machine is determined by a random generator so that the probability of success is the same as that given by the directory service agent. Note that the result of parsing the text while looking for a given query does not affect the success of the task, which is in fact decided by the random generator. If the search at a machine is successful, the information retrieval agent returns to the home machine where it was launched. Otherwise, it migrates to the next unvisited machine.

Experiments were run among seven laptop computers distributed in three subnetworks, as shown in Fig.6.1. Subnetwork 1 has three identical Toshiba laptop computers and one Gateway laptop computer, which serves as the home subnetwork to run the planning agent, the network sensing agent, and the directory service agent. Subnetwork 2 contains a same Toshiba laptop computer, while subnetwork 3 contains 2 identical Toshiba laptop computers. These three subnetworks are connected by a switch.

6.1.2 Physical Configuration

The configuration of these seven laptops are as follows:

- 6 Toshiba Tecra 500CS
  - Processor: Pentium 133MHz
  - RAM size: 16MB
  - Linux partition size: 600MB
  - Swap partition size: 65MB
  - Linux Kernel version: 2.0.30
  - Card Manager version: 2.9.11

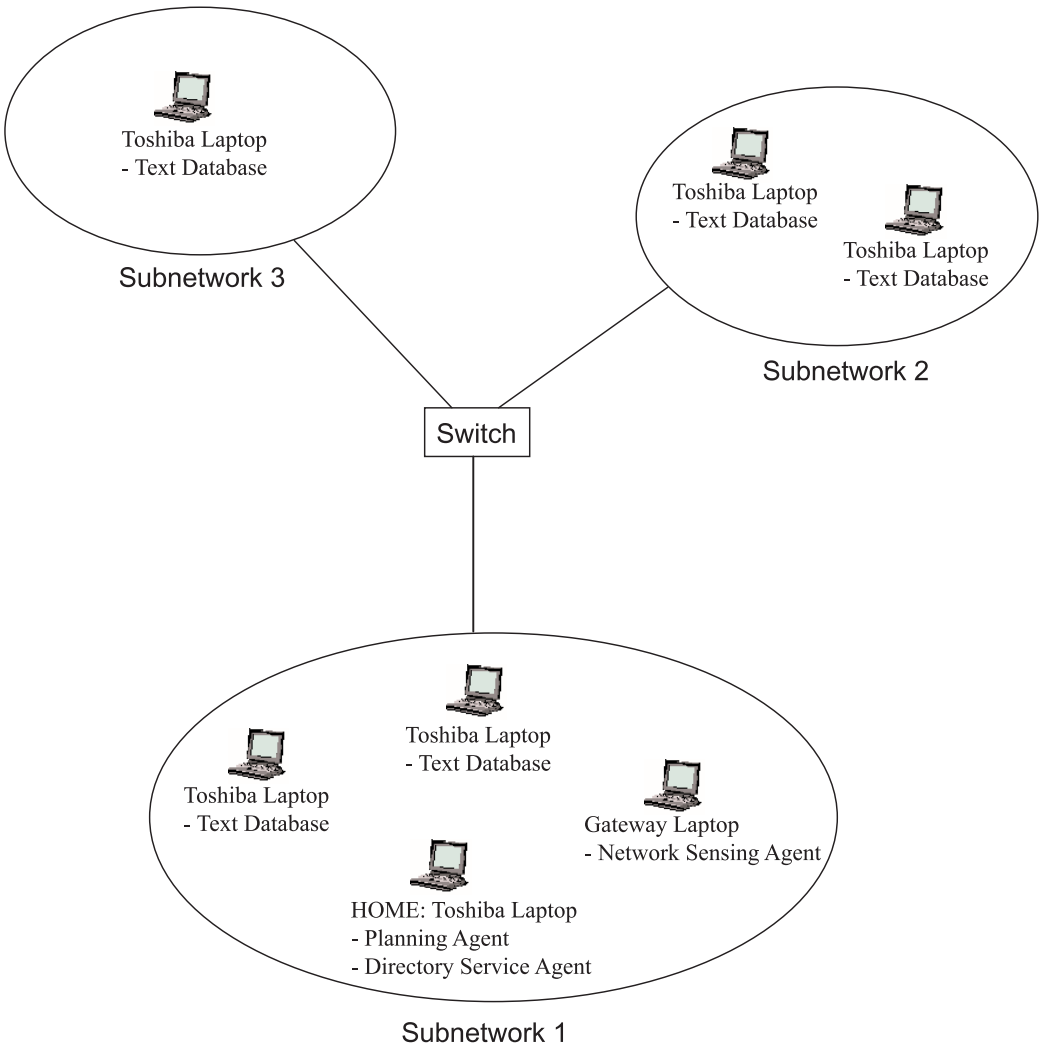

Figure 6.1: Network setup for the experiments

  - Ethernet Card: 3Com EtherLink 3 (10Base-T)

- 1 Gateway Solo 2300
  - Processor: Pentium II 200MHz
  - RAM size: 32MB
  - Linux partition size: 1.7GB
  - Swap partition size: 133MB
  - Linux Kernel version: 2.0.30
  - Card Manager version: 2.9.11
  - Ethernet Card: 3Com EtherLink 3 (10Base-T)

6.1.3 Software Configuration

One of the Toshiba laptops in subnetwork 1 serves as the home machine where an information retrieval information agent is launched and returns after completing its task. Both the planning agent and the directory service agent run on the home machine. The remaining Toshiba laptops in subnetwork 1 serve to hold the text database where the agents look for information. The Gateway laptop in subnetwork 1 is dedicated to the network sensing processes; the network sensing agent, the central network sensing daemon, and the clustering module are permanently located there. All the Toshiba laptop computers in subnetwork 2 and 3 serve to hold the text database.

Two network sensing related processes are constantly running on all seven machines. One process measures one-way latency from the other machines and the other process collects the CPU load of the local machine every 30 seconds and then reports it back to the central network sensing daemon.

In order to vary CPU loads on the 5 Toshiba laptops, different numbers of computing processes are run on each of them during the experiments. The number of running processes on each machine is decided randomly between a given range before each experiment.

The inter-subnetwork and intra-subnetwork latencies are differentiated by adding a virtual latency to an actually measured latency between nodes belonging to the different subnetworks.

As mentioned above, machines where agents search for information are 5 identical Toshiba laptops. This setting can eliminate the efforts requiring for adjusting the difference of CPU performance

among those machines to estimate their computation time based on the measured CPU load.

Configuration of the planning related software is as follows:

- The home machine: The Toshiba Tecra 500CS laptop
- The Planning agent: On the home machine
- The Directory service agent: On the home machine
- The Network Sensing agent: The Gateway Solo laptop
  - The Central Network Sensing Daemon: On the Gateway laptop
  - The Client Network Sensing Daemon: On the 6 Toshiba laptops
    - The network sensing frequency: 30 seconds
    - The range of added CPU load on each machine (%): [0 10]
    - The latencies matrix between three subnetworks (in ms)

      · Single-Agent Case

      $$\begin{pmatrix} 0 & 526 & 236 \\ 526 & 0 & 470 \\ 236 & 470 & 0 \end{pmatrix}$$

      · Multiple-Agent Case

      $$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

  - The Cluster Module: On the Gateway laptop
    - The number of clusters: 3
    - The size of pools in genetic algorithm: 20
    - The probability of mutation: 0.01
- Information Retrieval Agent (parses the text file): on each Toshiba laptop
  - The size of text file: 234MB
  - The computation time on Toshiba machine(5% CPU load): 21ms

6.2 Experiments and results

This section overviews three different series of experiments and provides the experimental results along with their short analysis.

6.2.1 A Single Agent without Deadlines

The scenario of the first experiment is that of a single information retrieval agent which looks for information with the assistance from the planning agent.

The experiments are executed on 50 differently-configured environments where the probability of success (for an agent to find the information) and the added CPU load on each machine are decided randomly. The range of the randomly generated probability of success is [0, 1], while the range of the added CPU load is [0% 10%].

In each test environment, an agent consults three different planning agents for the itinerary and follows each of the itineraries to find the information. The execution times for each itinerary are measured and compared. The first planning agent uses the planning algorithm developed for the subnetwork planning problem of a single agent in section 4.2, while the second and the third planning agents use the greedy methods based on the probability of success and on the estimated computation time at each machine (calculated based on its current CPU load, benchmarked CPU performance, and the estimated size of a task) respectively.

The results of the first experiment are shown in Fig.6.2 through Fig.6.7 and Table 6.1. Fig.6.2 and Fig.6.3 describe the performance comparisons of this thesis's algorithm (the optimal algorithm) versus the greedy planning method on the probability of success in terms of elapsed time (the execution time + the time spent for planning) and in terms of the execution time only. Fig.6.4 and Fig.6.5 are the performance comparisons between the optimal algorithm and the greedy planning method based on the computation time. Note that experimental runs are sorted in the increasing order of times of the optimal algorithm. Fig.6.6 and Fig.6.7 are the histograms of the normalized performance of the greedy methods on the probability of success and on the computation time, respectively, versus the optimal algorithm.

The summary performance statistics reported in the Table 6.1 are defined as follows:

- First place finishes : a count of runs where a method has the best performance among all

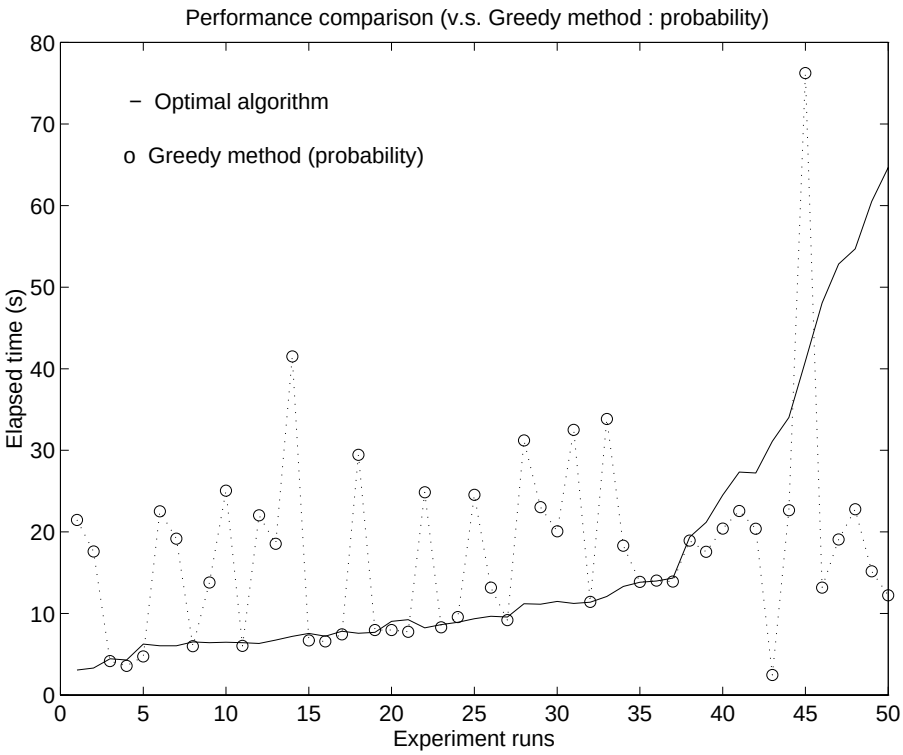

Figure 6.2: Performance comparison (Greedy on Prob): Elapsed time

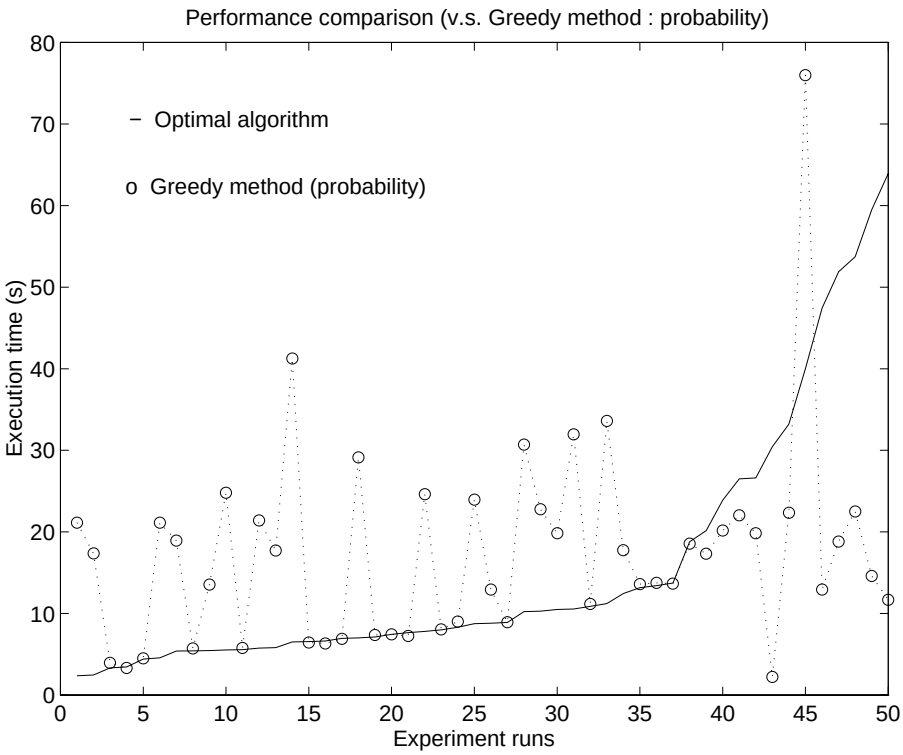

Figure 6.3: Performance comparison (Greedy on Prob): Execution time

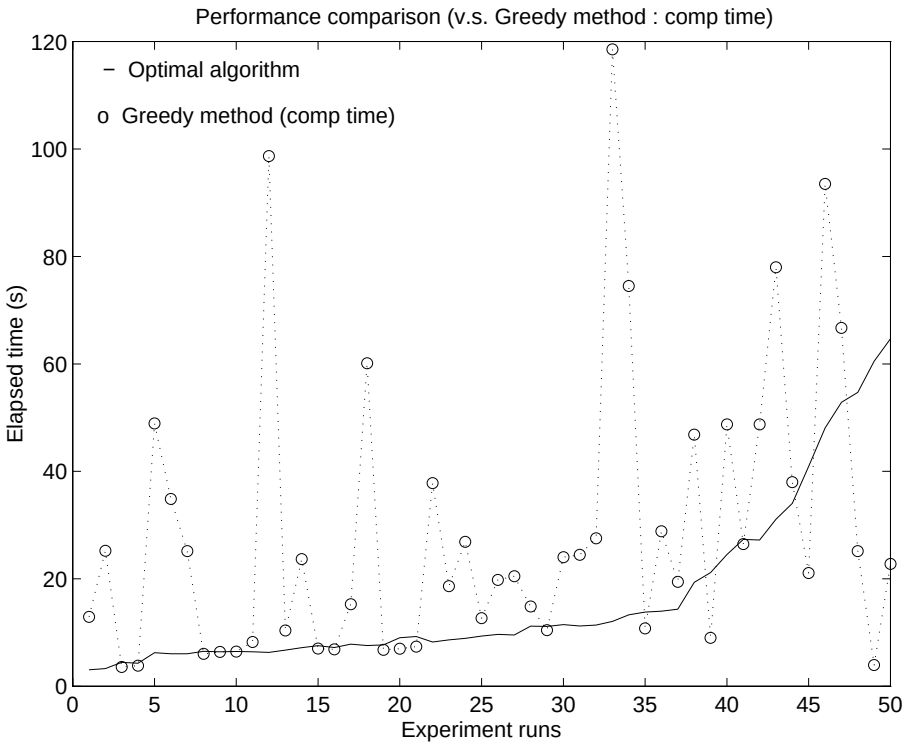

Figure 6.4: Performance comparison (Greedy on Comp): Elapsed time

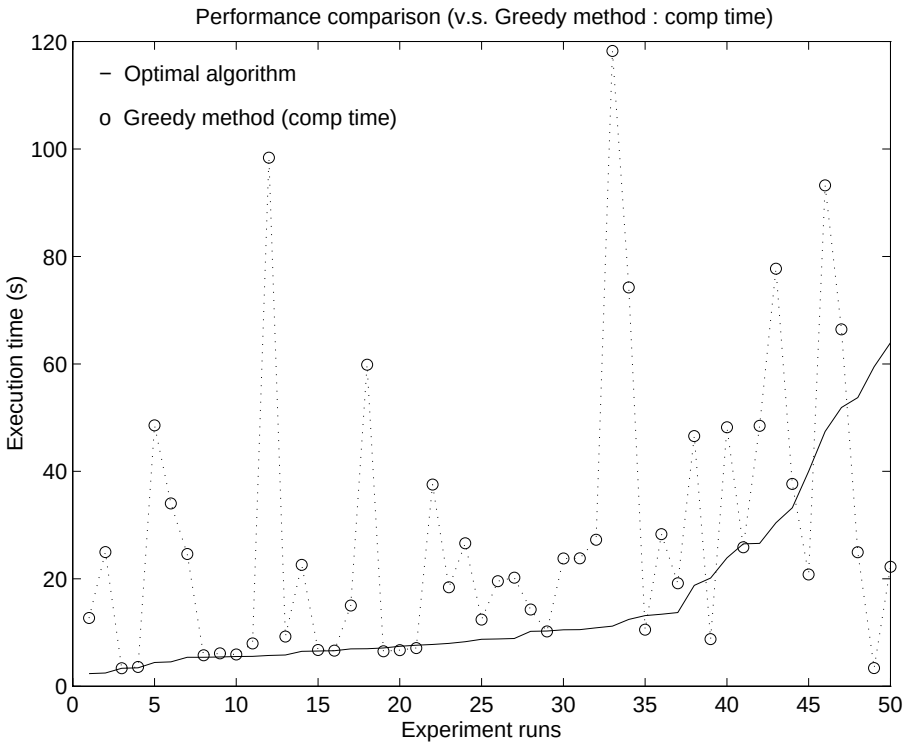

Figure 6.5: Performance comparison (Greedy on Comp): Execution time

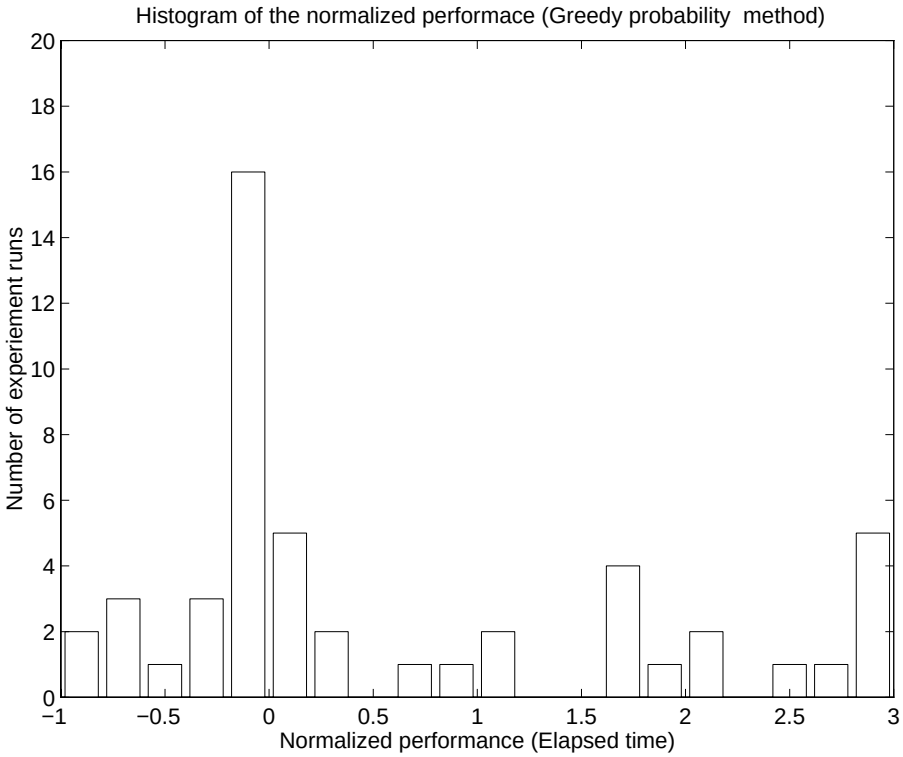

Figure 6.6: Histogram (Greedy on Prob): Elapsed time

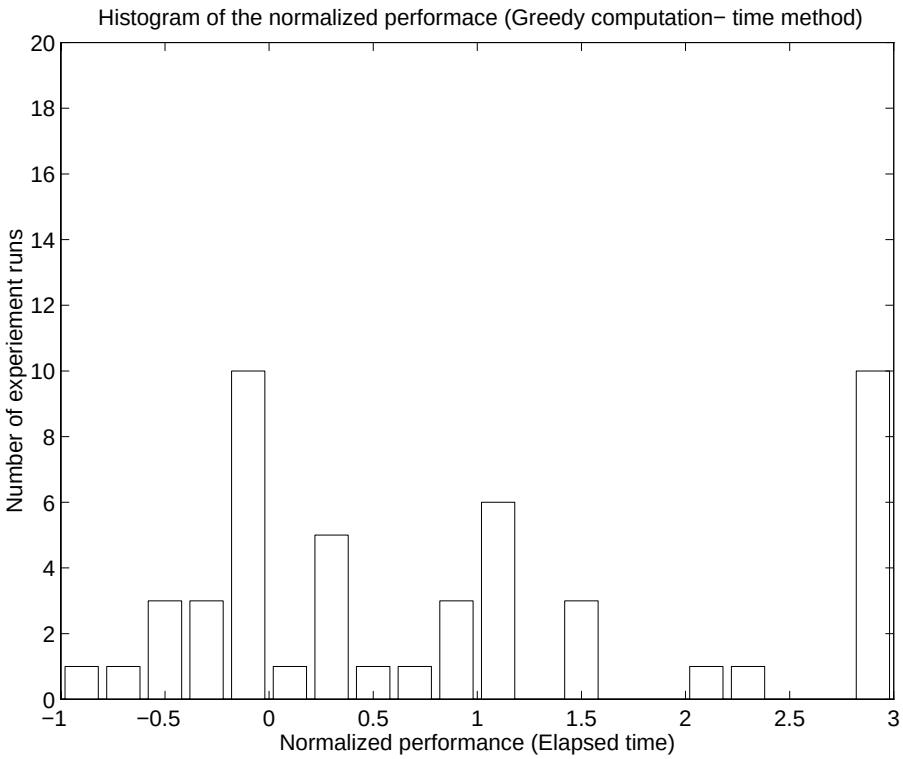

Figure 6.7: Histogram (Greedy on Comp): Elapsed time

tested algorithms.

- Average rank order : the average ranking of each method

  This is obtained as:

  $$\frac{1}{n}\sum_{i=1}^{n} R_M(i)$$

  where $R_M(i)$ is the rank of the method M on the ith run. The range of ranking that each method may take is from 1 to the number of methods.

- Arithmetic mean : the normalized average value of times of each method

  This value is obtained as;

  $$\left(\frac{\sum_{i=1}^{n} Time_M(i)}{\sum_{i=1}^{n} Time_{Op}(i)}\right).$$

  This value won't be valid if the deviation of times is large because a large time dominates the effect on the value.

- Arithmetic mean (normalized) : the average value for normalized times of each method

  This value is calculated as:

  $$\left(\sum_{i=1}^{n} \frac{Time_M(i)}{Time_{Op}(i)}\right)/n.$$

  where $Time_M(i)$ stands for the elapsed time (or execution time) for each method M on its ith run, $Time_{Op}(i)$ stands for that of the optimal algorithm on its ith run, and n is the number of runs. Note that this statistic is not considered a good performance measure [24, 53].

- Geometric mean : the geometric mean of times of each method

  It is calculated as:

  $$\left(\prod_{i=1}^{n} \frac{Time_M(i)}{Time_{Op}(i)}\right)^{\frac{1}{n}}.$$

- Weighted arithmetic mean : the weighted arithmetic mean of times of each method

This value is obtained as:

$$\left(\sum_{i=1}^{n} \frac{Time_M(i)}{\sum_{j=1}^{m} Time_{M_j}(i)} \right) / n$$

where $Time_{M_j}(i)$ is the time of the jth method M_j that is used on the ith run of the experiment and m is the number of methods.

This value represents the average percentage of a time for each method where 100% is the sum of times of methods that used on a same run of the experiment.

- Average planning time : the average planning overhead time that includes time for accessing network statistics and planning

  The unit of this value is seconds.

- Planning time / Elapsed time : average of the ratio of planning overhead time versus the elapsed time (execution time + planning overhead time)

  This is calculated as:

$$\frac{1}{n} \sum_{i=0}^{n} \frac{T_p(i)}{T_p(i) + T_{ex}(i)}$$

  where $T_p(i)$ and $T_{ex}(i)$ are the planning overhead time and the execution time of a method on the ith experimental run, respectively, and n is the number of runs.

As we can see in the figure of performance comparisons, the optimal planning algorithm does not always outperform the other two methods. This is explained by the stochastic nature of the planning problem. For example, an agent may find the information at the first machine even if it has a very small probability of success. The optimal algorithm guarantees the minimum *expected* time until the desired information is found, not the minimum time in all cases. Thus, due to the stochastic character of the planning problem, it is appropriate to compare algorithms based on the average values shown in Table.6.1.

If you look at the count of best cases of each method and the average ranking based on the elapsed time in Table 6.1, you can notice that the optimal algorithm does not necessarily perform best, though it has been proved to be optimal. However, comparing the execution times only,

	Optimal algorithm Algorithm	Greedy algorithm (probability)	Greedy algorithm (computation time)
First place finishes	19 (27)	20 (15)	11 (8)
Average rank order	1.86 (1.60)	1.84 (1.98)	2.30 (2.42)
Arithmetic mean	1 (1)	1.08 (1.11)	1.76 (1.83)
Arithmetic mean (normalized)	1 (1)	1.74 (1.94)	2.59 (2.91)
Geometric mean	1 (1)	1.22 (1.31)	1.66 (1.79)
Weighted arithmetic mean	1 (1)	1.19 (1.26)	1.58 (1.67)
Average Planning time (s)	0.842	0.377	0.36
Planning time/Elapsed time	0.0934	0.0304	0.0257

Table 6.1: Performance Comparison : TAP without deadlines (In the first 5 rows, numbers without parentheses are based on the elapsed time, while numbers in parentheses are based on the execution time)

the optimal planning algorithm outperforms the other methods. This situation explains that the itinerary created by the optimal algorithm is better, but it takes longer to create it. The larger ratio of "Planning time / Elapsed time" supports this argument. Moreover, in Fig.6.2 and Fig.6.4, there are many cases where the elapsed time of the optimal algorithm and that of the other methods are close to each other, e.g., 20 out of 50 cases in Fig.6.2 and 16 out of 50 cases in Fig.6.4. In these cases, the greedy methods slightly outperform the optimal algorithm on elapsed time due to their shorter planning time. As a result, if the planning problem is large, i.e., the expected elapsed time is large, the optimal algorithm would outperform the other greedy method because the ratio of planning time (which tends to be constant) to its elapsed time will decrease.

The histograms in Fig.6.5 and Fig.6.6 explain that both greedy methods perform well enough in many cases (there are large counts near zero) but in some cases their performance is significantly worse compared to the optimal algorithm. The bad performance in those cases contributes largely in the increase of all the kinds of mean values in the table 6.1 .

6.2.2 Multiple Agents without Deadlines

The multiple agent planning problem without deadlines uses several different algorithms for solution depending on the distribution of probabilities of success and CPU load (or the computation time). Therefore, experiments for this planning problem consist of sub-experiments where each of the planning algorithms described in Section 4.3. is tested. Each sub-experiment is executed on 20

Sub-experiment	Algorithm	Probability range	CPU range (%)
Probability $\simeq 0.9$	Theorem 11	[0.85 0.95]	[5 15]
Probability $\simeq 0.7$	Theorem 11	[0.65 0.75]	[5 15]
Probability $\simeq 0$	Theorem 8	[0 0.5]	[5 15]
Constant exec time	Theorem 12	[0 1]	[5 5]

Table 6.2: The setting of sub-experiments

	Optimal planning Algorithm	Greedy algorithm (probability)
First place finishes	16 (17)	4 (3)
Average rank order	1.2 (1.15)	1.8 (1.85)
Arithmetic mean	1 (1)	1.48 (1.51)
Arithmetic mean (normalized)	1 (1)	1.74 (1.82)
Geometric mean	1 (1)	1.44(1.46)
Weighted arithmetic mean	1 (1)	1.38 (1.40)
Average planning time (s)	0.56	0.49
Planning time/Elapsed time	0.057	0.041

Table 6.3: Performance Comparison : Prob $\simeq 0.9$ (In the first 5 rows, numbers without parentheses are based on the elapsed time, while numbers in parentheses are based on the execution time)

differently-configured environments where the probability of success and the CPU load are randomly decided between each respective range as shown in Table 6.2.

Note that the above ranges of probabilities and CPU load are chosen based on the experimental experience.

There are two agents in these experiments. The agents communicate with each other so that as soon as one of them successfully completes the other can terminate.

The two greedy algorithms employed in the previous experiment are used again in this experiment for performance comparison. The greedy algorithms sort sites to be visited according to either the probability of success or the computation time and then assign the next unvisited site to the next available agent.

The results of the experiment are shown in Fig.6.8 through Fig.6.12 and in Table 6.3 through Table 6.6. Each figure compares the performance between the optimal algorithm and the greedy methods. The x-axis indexes the experimental run and the y-axis the execution time of each run. The format of those tables (Table 6.3 through Table 6.6) are same as that of Table 6.1.

The optimal algorithm is same as the greedy probability method in the first two sub-experiments

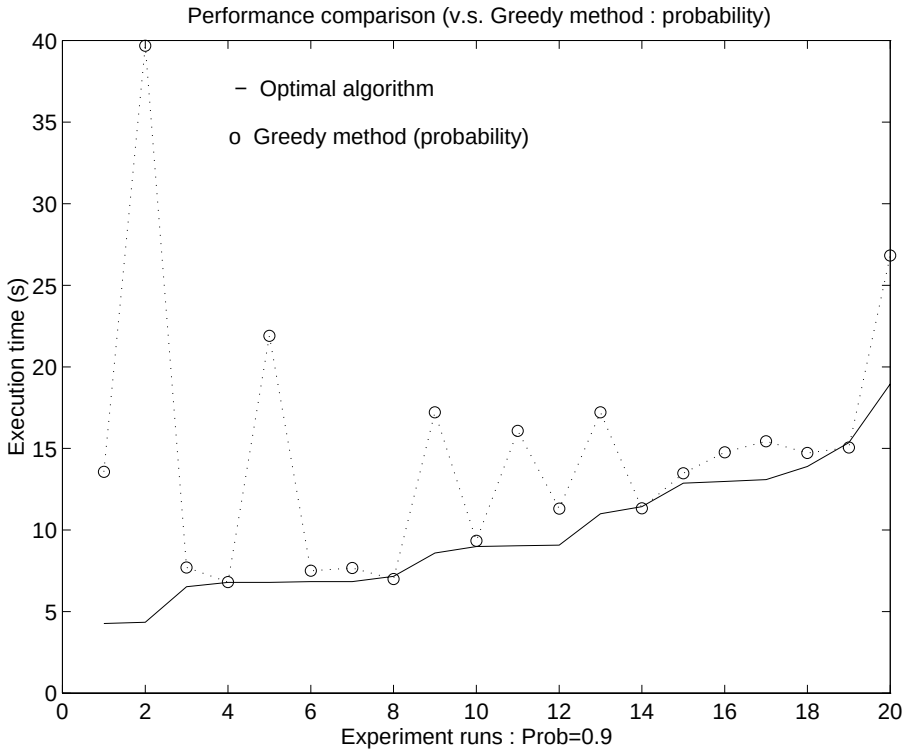

Figure 6.8: Performance comparison (Greedy on Prob): Prob $\simeq 0.9$

	Optimal planning Algorithm	Greedy algorithm (probability)
First place finishes	17 (18)	3 (2)
Average rank order	1.15 (1.1)	1.85 (1.9)
Arithmetic mean	1 (1)	1.10 (1.16)
Arithmetic mean (normalized)	1 (1)	2.10 (2.16)
Geometric mean	1 (1)	1.57 (1.59)
Weighted arithmetic mean	1 (1)	1.53 (1.54)
Average planning time (s)	0.35	0.40
Planning time/Elapsed time	0.043	0.036

Table 6.4: Performance Comparison : Prob $\simeq 0.7$ (In the first 5 rows, numbers without parentheses are based on the elapsed time, while numbers in parentheses are based on the execution time)

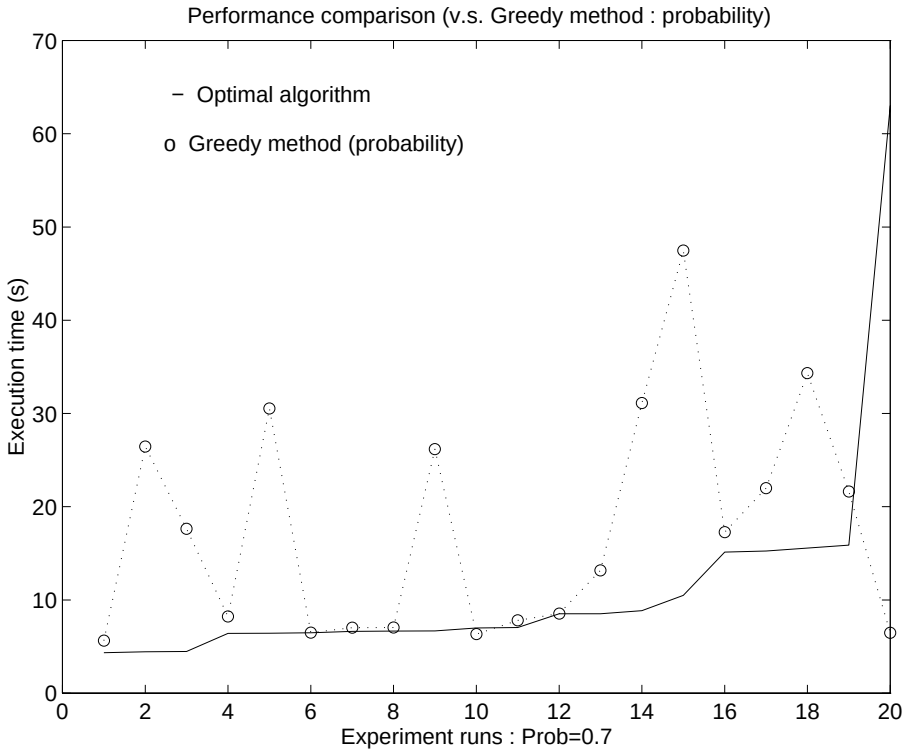

Figure 6.9: Performance comparison (Greedy on Prob): Prob $\simeq 0.7$

	Optimal planning Algorithm	Greedy algorithm (probability)	Greedy algorithm (computation time)
First place finishes	12 (11)	4 (5)	4 (4)
Average rank order	1.50 (1.50)	2.05 (2)	2.45 (2.5)
Arithmetic mean	1 (1)	1.05 (1.06)	1.11 (1.12)
Arithmetic mean (normalized)	1 (1)	1.04 (1.04)	1.13 (1.14)
Geometric mean	1 (1)	0.994 (0.999)	1.073 (1.079))
Weighted arithmetic mean	1 (1)	1.002 (1.006)	1.085 (1.091)
Average planning time (s)	0.75	0.46	0.48
Planning time/Elapsed time	0.015	0.010	0.094

Table 6.5: Performance Comparison : Prob $\simeq 0$ (In the first 5 rows, numbers without parentheses are based on the elapsed time, while numbers in parentheses are based on the execution time)

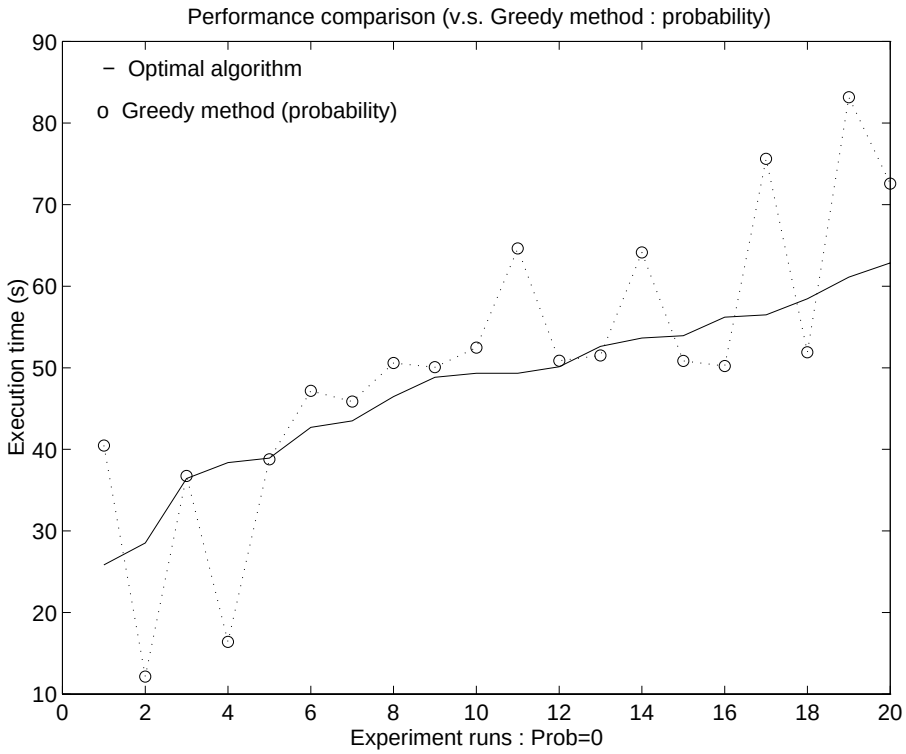

Figure 6.10: Performance comparison (Greedy on Prob): Prob $\simeq 0$

	Optimal planning Algorithm	Greedy algorithm (computation time)
First place finishes	15 (14)	5 (6)
Average rank order	1.25 (1.3)	1.75 (1.7)
Arithmetic mean	1 (1)	1.27 (1.29)
Arithmetic mean (normalized)	1 (1)	0.35 (0.41)
Geometric mean	1 (1)	1.22 (1.24)
Weighted arithmetic mean	1 (1)	1.20 (1.21)
Average planning time (s)	0.38	0.40
Planning time/Elapsed time	0.12	0.11

Table 6.6: Performance Comparison : Constant computation time (In the first 5 rows, numbers without parentheses are based on the elapsed time, while numbers in parentheses are based on the execution time)

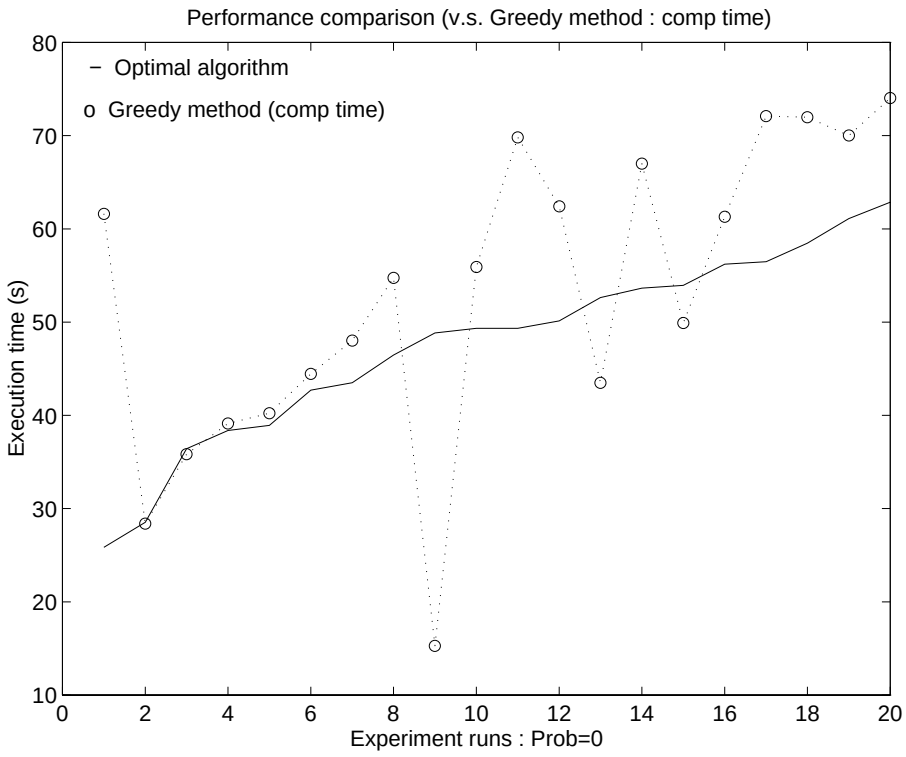

Figure 6.11: Performance comparison (Greedy on Comp): Prob $\simeq 0$

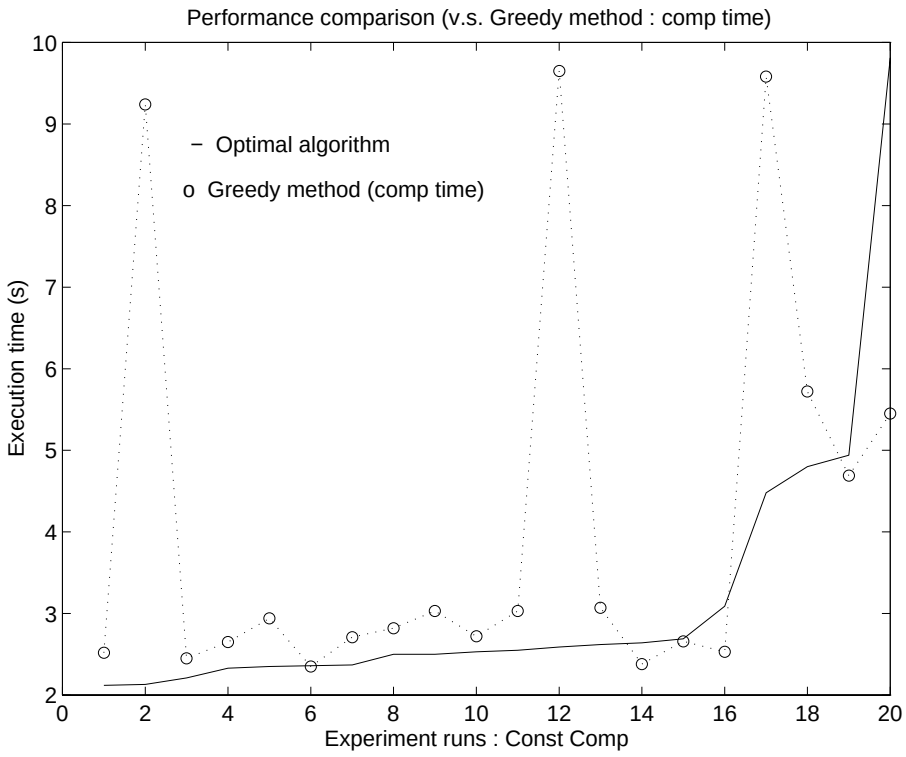

Figure 6.12: Performance comparison (Greedy on Comp): Constant computation time

and is same as the greedy computation time method in the last sub-experiments. As we can see in Table 6.3, 6.4 and 6.6, the optimal algorithms outperform the other greedy method on the all the performance measures in these sub-experiments.

The optimal algorithm used in the third sub-experiment produces an itinerary different from the itinerary produced by the two greedy methods. The algorithm shows the best overall performance according to Table 6.5, especially by looking at the count of the first place finishes. However, the advantage of the optimal algorithm versus the greedy probability algorithm is small as measured by the mean values. In fact, it is worse than the greedy algorithm using the geometric mean. This can be explained by the approximation of the probabilities. The optimal algorithm should be the best in the case where the all the probabilities are zero. The algorithm assumes that mobile agents visit all the machines in failure. However, in this sub-experiment, because of the approximation, probabilities can take values between 0 and 0.05. Mobile agents may be able to terminate a tour early, though the chance is very small. In this situation, the sooner the agent finds information, the shorter the total time will be. Thus, the greedy probability method that may let agents avoid visiting all sites at the earlier stage can outperform the other methods in some runs.

The optimal planning algorithm requires more time for calculation than the other methods, as seen in Table 6.5. This is due to its larger complexity. The algorithm is of a pseudo-polynomial time form, while the two greedy methods are simple polynomial time algorithms. Recall that the complexity of the optimal planning algorithm is polynomial in the sum of the computation times at each site, while the complexity of the greedy methods is proportional to the number of sites.

Unfortunately, the solution for the multiple agent planning problem without deadline is not complete. The optimal solution to the case where the probability of success is constant but smaller than 0.5, or, in general, the probabilities are variables, is not known yet, though a heuristic algorithm for this case has been already built (but was not described in this thesis.) The development of the optimal algorithm for this case will be explored in the future.

6.2.3 A Single Agent with Deadlines

The third experiment looks at the planning problem that maximizes the probability of success without violating the deadline constraints. The planning methods used for the experiment are the optimal algorithms described in Section 4.3 as well as the greedy algorithms employed in the previous

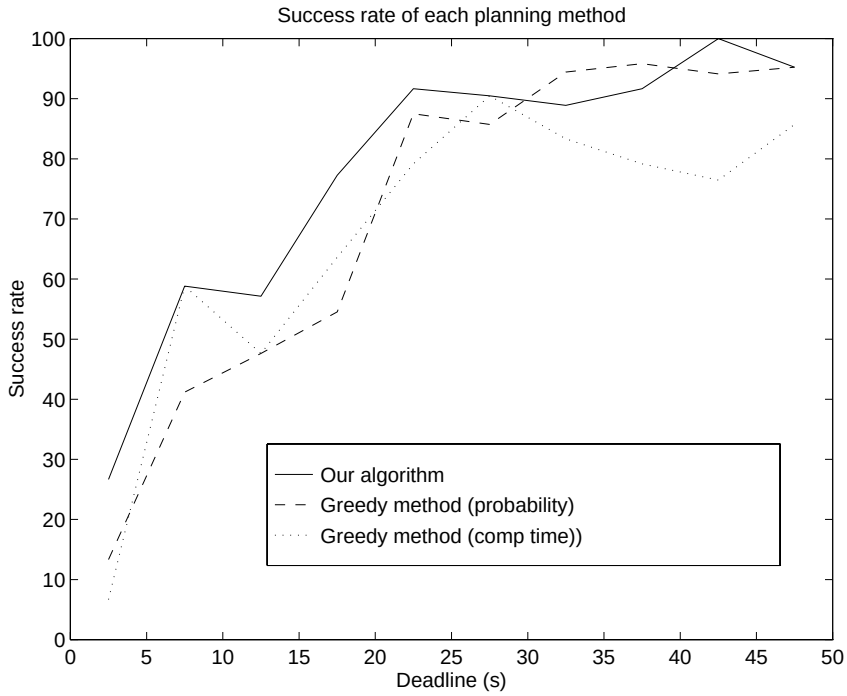

Figure 6.13: Success rate of planning methods (200 runs)

experiment for performance comparisons.

Though different deadlines can be set for each machine in the network, this experiment sets the deadline only on the home machine for simplicity. This is a realistic problem. The experiment counts the number of successful cases. Note that the time is measured only after the itinerary is obtained from the planning agent.

A single agent is executed in 200 different randomly-configured environments. The range of the randomly generated probability of success is [0, 1], and the range of the CPU load is [5 15](%). The number of successful completion are counted for each of the three different planning algorithms in 50 different environments. The deadline at the home machine is also decided randomly between [2500 50000](ms).

The results of this third experiment are shown in Fig.6.13 and in Table 6.7. Fig.6.13 shows the ratio of successful runs for each range of deadlines. Table 6.7 contains counts of successful runs for each planning method.

According to Table 6.7, the optimal planning algorithm has the highest successful count, thus it

	Optimal planning Algorithm	Greedy algorithm (probability)	Greedy algorithm (computation time)
Success count	33	31	28

Table 6.7: Performance comparison : TAP with deadlines

performs the best. The larger a deadline becomes, the better the success rate is for all algorithms. This result is reasonable because larger deadlines make the planning problem easier and allows even weaker algorithms to schedule an agent successfully. This reasoning can explain why the advantage of our algorithm versus other methods is prominent in the small deadline case that is more difficult to solve.

Chapter 7

Further Work

This thesis has developed analytic theories and efficient algorithms for various mobile mobile agent planning problems. An implementation of a planning system for D'Agents using these algorithms has been completed and tested. However, there are several important avenues for the future work.

- Extension of the theories and algorithms

  The developed theories and algorithms do not cover certain cases of the traveling agent problems (TAPs). As mentioned at the end of Section 4.3, in the traveling multiple agents problem(TMAP), the optimal algorithm for the case where the probabilities are variable is not developed yet. Only a heuristic solution has been considered.

  Another possible extension of the theory development is on TAP with Deadlines. The optimal solution maximizes the probability of success, but it does not guarantee the minimum expected total time without violating the deadline constraints. The theory that satisfies the latter criteria would be useful.

  Moreover, TMAP and TMAP with Deadlines employed the strong assumption that latencies are all the same. However, this assumption will not always hold in real applications. Various latency conditions in TMAP and TMAP with Deadline should be studied in the future.

- More general planning problems

  The planning problem we dealt with in this thesis is a very specific case where the agent has only a single task to complete. In many applications, a mobile agent can have multiple tasks, whose order of completion is important. A complete planning system should handle such cases. However, this problem would be very hard to solve because even the single task problem is

NP-complete in general. The compromise of using a heuristic or approximation method may be necessary.

- Advanced directory service

  A primitive directory service has been implemented as a part of the planning system in this thesis. Full functionality will be achieved when information management services are added to the current implementation. Additional services include indexing keeping track of the location of information, updating the table if the location changes, and calculating the reference ranking of the location (probabilities of success used in Chapter 4). The mechanism for calculating a reference ranking also needs to be better understood. One proposed idea is to calculate the ratio of the amount of information at the site over the total amount of related information in the network.

- Network sensing frequency

  The network sensing frequency affects the performance of the network sensing module. A smaller interval makes the collected network statistics to be more accurate, but it increases network traffic. The current network sensing module uses a fixed interval defined by the user. However, it will be better if the interval is adaptively changed by the network sensing module itself based on historical network conditions and some other factors.

- Number of multiple agents

  In the current planning system, the number of multiple agents to be used is defined by the user. It is preferable that the planning system decides the optimal number of agents so that the total expected time can be minimized. The total expected time must include the start-up time of a new Tcl interpreter for creating another agent during execution. Thus, a large number of agents does not necessarily lead to a smaller expected time.

  An easy method for obtaining a good number of agents is to keep calculating the total expected time for several multiple agent cases and choosing the number that minimizes the total expected time. However, the overhead for such calculation should be smaller than the execution time saved by increasing the number of mobile agents.

Chapter 8

Conclusion

This thesis has successfully shown (1) some planning problems for a mobile agent can be solved efficiently if realistic assumptions are made and (2) it is possible to incorporate efficient planning into an actual mobile agent system.

A sequence of planning problems that arises in common mobile agent's applications such as information retrieval and data-mining has been proposed in this thesis. In these application a mobile agent keeps migrating to different machines to execute its task until the task is successfully completed. We have proposed a model where a mobile agent spends time in traveling between machines (the travel time) and in executing its task at a machine (the computation time). Each site is associated with a "probability of success", i.e., the probability that the mobile agent can successfully complete its task at this site. If it fails, it migrates to some other machine to continue its task. The mobile agent planning problem is to find the best sequence of machines for a mobile agent to visit so that it can successfully complete its task in minimum expected time. The problem are called *Traveling Agent Problems* due to the analogy with Traveling Salesman Problems. Unfortunately, the general formulation of the Traveling Agent Problems is NP-complete.

This thesis successfully has developed analytical theories and polynomial algorithms for the Traveling Agent Problems under the following assumptions: (1) the travel time (latency) between all the machines in the network is constant; or (2) the travel time between machines in a same subnetwork is constant, but it is various between machines located in different subnetworks.

Under the first assumption, the solution to the planning problem is to sort the machines in the

decreasing order of:

$$\frac{probability\ of\ success}{computation\ time}.$$

This result leads us to the solution to the more complex problem under the second assumption. The solution is obtained using sorting and dynamic programming. The complexity of the problem becomes

$$\prod_{i=1}^{m}(1+n_m)$$

where m is the number of subnetwork and n_m is the number of machines in subnetwork m.

Different versions of the Traveling Agent Problem also have been considered: (1) multiple agent problems (multiple agents cooperate to complete the same task) and (2) deadline problems (a single or multiple agents need to complete a task without violating a deadline constraint at each location in the network).

The solution to the first problem (the multiple agent planning problem) has been proved to be a simple polynomial-time sorting problem, if the travel time is constant and the probabilities of success are constant and larger than 0.5, or if all the computation time are the same. If the travel time is constant and the probabilities are all 0, the problem becomes 2 partition problem that can be solved in pseudo-polynomial time. In the rest of the cases, the problem needs a heuristic solution.

A concept of deadlines has been introduced in the second problem. A deadline is the time at which a machine becomes unreachable. Thus, a mobile agent needs to finish its task at a machine before its deadline. The planning problem is to decide the sequence of machines that maximizes the probability of success. The optimal solution has been proved to be obtained in pseudo-polynomial time using dynamic programming in a single agent case with subnetworks and in a multiple agent case where the traveling time is constant.

The developed planning algorithms have been implemented as the core of the planning system for a mobile agent package *D'Agents*[26, 27]. The subcomponents of the planning system such as a directory service, a network sensing module, a clustering module have been implemented. The directory service provides the location information where a mobile agent can execute its task along with the probability of success. The network sensing module periodically collects the network statistics (latencies and CPU loads) and reports them to a mobile agent when requested. The

clustering module forms virtual subnetworks (where the latencies are considered constant) based on the collected network statistics.

The experiments of the planning system on the all the developed optimal planning algorithms has showed that in average the planning algorithms outperform available heuristic algorithms (i.e., the greedy methods based on the probability of success and on the computation time) despite their slightly larger calculation overhead.

Thus, this thesis has shown a sequence of planning problems for mobile agents have efficient solutions under realistic assumptions and efficient planning can be incorporated into a mobile agent system such as D'Agents.

Bibliography

[1] J. Ambros-Ingerson and S. Steel. "Integrating planning, execution and monitoring" *In Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88),* Morgan Kaufmann, St. Paul. Minnesota, 1972

[2] A. Acharya, M. Ranganathan, J.Saltz. "Sumatra: A Language for Resource-aware Mobile Programs," *Mobile Object Systems,* J.Vitek and C.Tschudin Eds., 111-130, Springer Verlag, 1997

[3] Richard E. Bellman. *Dynamic programming,* Princeton University Press, Princeton, New Jersey 1957

[4] D. M. Bertsekas. *Dynamic Programming,* Prentice Hall, Englewood Cliffs , NJ, 1987.

[5] D. M. Bertsekas. *Dynamic Programming and Optimal Control,* Athena Scientific, Belmont, Massachusetts, 1995.

[6] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming,* Athena Scientific, Belmont, Massachusetts, 1996

[7] W. Caripe, G. Cybenko, K. Moizumi and R. Gray. "Network Awareness and Mobile Agent Systems", *IEEE Communications,* July 1998

[8] E. K. P. Chong and S. H. Zak. *An introduction to optimization,* John Wiley & Sons, Inc. New York, New York, 1996

[9] G. Cybenko, R. Gray, and K. Moizumi. "Q-learning: A Tutorial and Extensions" *Mathematics of Neural Networks,* Kluwer Academic Publishers, Boston/London/Dordrecht, 1997

[10] M. Cohn, B. Morgan, M. Morrison and etc. *JAVA Developer's Reference,* Sams.net, Indianapolis, 1996

[11] M. J. Conway "Python: A GUI Development Tool", *ACM Interactions,*, April 1995

[12] S. A. Cook, "The complexity of theorem-proving procedures" *Proceedings of 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery,* New York, 1971

[13] K.W. Currie and A. Tate. "O-Plan: The Open Planning Architecture" *Artificial Intelligence,* 52(1). 1991

[14] S. E. Deering. "Host Extensions for IP Multicasting," RFC 1112, 17 pages, August, 1989

[15] T. Dean, L P. Kaelbling, J.Kirman, and Ann Nicholson. "Planning with deadlines in stochastic domains" *In Proceedings of AAAI-93,* Washington, D.C., 1993

[16] T. Dean and K. Kanazawa. "A model for reasoning about persistence and causation" *Computational Intelligence,* 1989

[17] M. Drummond and J. Bresian. "Anytime synthetic projection: Maximizing the probability of goal satisfaction" *In Proceedings of AAAI-90,* Boston, 1990

[18] O. Etioni, S. Hanks, D. Draper, N. Lesh, and M. Williamson. "An approach to planing with incomplete information" *In Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning,* 1992

[19] R. E. Fikes, P.E. Hart, N. J. Nilsson. "Learning and executing generalized robot plans" *Artificial Intelligence,* 3(4), 1972

[20] U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth. *From Data-Mining to Knowledge Discovery: An Overview,* Fayyad, Piatetsky-Shapiro, Smyth & Uthurusamy, 1-37, 1995

[21] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy. *Knowledge Discovery in Data Bases 2,* AAAI Press / The MIT Press, Menlo Park, CA, 1995

[22] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining,* The AAAI press/ The MIT press, Cambridge, MA, 1996

[23] R. E. Fikes and N. J. Nilsson. "STRIPS: a new approach to the application of theorem proving to problem solving" *Artificial Intelligence,* 2(3-4), 1971

[24] P. J. Fleming and J. J. Wallace. "How not to lie with statistics: The correct way to summarize benchmark results" *Communications of the ACM,* March 1986

[25] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of *NP*-Completeness, W.H. Freeman and Company, San Francisco, 1979

[26] Robert S. Gray. "Agent Tcl: Alpha Release 1.1." Available by WWW at http://www.cs.dartmouth.edu/řgray/documentation/doc.1.1.ps.gz. December 1995

[27] Robert S. Gray. "Agent Tcl : A flexible and secure mobile-agent system" *Proceeding of the 1996 Tcl/Tk Workshop,* July 1996.

[28] R. A. Howard. *Dynamic programming and Markov Processes,* MIT Press, Cambridge, Massachusetts, 1960

[29] J. R. Jackson. "Scheduling a production line to minimize maximum tardiness" *Research Report 43, Management Science Research Project* University of California, Los Angeles, 1955

[30] Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. "Operating system support for mobile agents" *In Proceeding of the 5th IEEE Workshop on Hot Topics in Operating Systems,* 1995

[31] R. M. Karp. "Reducibility among combinatorial problems" in R.E. Miller and J.W. Thatcher (eds.), *Complexity of Computer Computations,* Plenum Press, New York, 1972

[32] S. Kirkpatrick, C. D. Gellantt and M. P. Vecchi, "Optimization by Simulated Annealing", *Science,* Vol 220, 671-680, 1983

[33] S Kirkpatrick. "Optimization by Simulated Annealing: Quantitative Studies", *Journal of Statistical Physics*, Vol. 34, 975-986, 1984

[34] T. Kohonen. *Self-Organization and Associative Memory,* Springer-Verlag, Berlin Heidelberg, 1989

[35] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection,* The MIT Press, MA, 1992

[36] E. Rich and R. Knight. *Artificial Intelligence,* McGraw Hill, New York 1991

[37] H. Levesque and R.J. Brachman. "Expressiveness and tractability in knowledge representation and reasoning" *Computational Intelligence,* 3(2), 1987

[38] E. Nicholas and H. Simon "The data association problem when monitoring robot vehicles using dynamic belief networks" *ECAI 92: 10th European Conference on Artificial Intelligence Proceedings,* Vienna, Austria. Wiley. 1992

[39] B. Noble, M. Sayanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, K. Walker. "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th ACM Symposium on Operating System Principles,* St. Malo, France, 1997

[40] OMG "The Command Object Request Broker: Architecture and Specification" *OMG TC Document Number 91.12.1,* Revision 1.1, December, 1991

[41] John K. Ousterhout. *Tcl and the Tk Toolkit,* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1994.

[42] S. Patek. Ingress planning in FASM, ALPHATECH Technical Report, Burlington, MA, 1997.

[43] J. Pearl. "Fusion, propagation, and structuring in belief networks" *Artificial Intelligence,* 29, 1986

[44] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference,* Morgan Kaufmann, San Mateo, California, 1988

[45] M. Ranganathan, A. Acharya, S.D. Sharma, J. Saltz. "Network-Aware Mobile Programs," *Proceeding of the USENIX Annual Technical Conference,* Anaheim, California, 1997

[46] D. Redfern and C. Campbell *The Matlab 5 Handbook,* Springer-Verlag, 1998

[47] Daniela Rus and Robert Gray and David Kotz. "Autonomous and Adaptive Agents that Gather Information," In *AAAI '96 International Workshop on Intelligent Adaptive Agents,* 107-116, August 1996

[48] D. Rus, R. Gray, and D. Kotz. "Autonomous and adaptive agents that gather information" *In AAAI '96 International Workshop on Intelligent Adaptive Agents,* August 1996

[49] E. D. Sacerdoti. "The nonlinear nature of plans" *In Proceedings of the Fourth International Joint Conference on Artificial Intelligence(IJCAL75),* Tbilisi, Georgia, 1975

[50] E. D. Sacerdoti. *A Structure for Plans and Behavior,* Elsevier/North-Holland, Amsterdam, London, New York, 1977

[51] S. P. Singh and D. P. Bertsekas. "Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems" *submitted to the 1996 Neural Information Processing Systems Conference,* 1996

[52] S. Seshan, M. Stemm, R. H. Katz. "SPAND: Shared Passive Network Performance Discovery," *USENIX Symposium on Internet Technologies and Systems,* 135-146, 1997

[53] James E. Smith. "Characterizing computer performance with a single number," *Communications of the ACM,* October 1988

[54] W. Stallings. *SNMP SNMPv2 and RMON,* Addison-Wesley, Reading, Massachusetts, 1996

[55] R. S. Sutton "Learning to Predict by the Methods of Temporal Differences" *Machine Learning,* Vol. 3, 1988

[56] G. J. Tesauro "Temporal Difference Learning and TD-Gammon" *Communications of the ACM,* Vol. 38, 1995

[57] J. Tash and S. Russell Control strategies for a stochastic planner *AAAI-94,* 1079-1085, Seattle, 1994

[58] C. J. C. H. Watkins "Learning from Delayed Rewards" *Ph.D. Thesis,* Cambridge University, Cambridge, England, 1989

[59] J. E. White. "Telescript technology: The foundation for the electronic marketplace" *General Magic White Paper,* General Magic, 1994