Using Temporal Information to Learn in a Dynamic Environment

by

Saleh Matar Mohammed Al-Hatali

Bachelor of Science
Computer Engineering
University of Arizona
1989

A thesis submitted to the Graduate School of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
October, 1995

We the undersigned committee
hereby approve the attached thesis

Using Temporal Information to Learn in a Dynamic Environment

by
Saleh Matar Mohammed Al-Hatali

_____	_____
Robert A. Morris, Ph.D. Associate Professor Computer Science Program Committee Chair	Dan E. Tamir, Ph.D. Associate Professor Computer Science Program
_____	_____
Margot A. Haberhern, Ph.D. Associate Professor Dept. of Humanities	William D. Shoaff, Ph.D. Associate Professor Chair, Computer Science Program

Abstract

Title: Using Temporal Information to Learn in a Dynamic Environment

Author: Saleh Matar Mohammed Al-Hatali

Major Advisor: Dr. Robert A. Morris

Key Words: intelligent agents, temporal events, duration uncertainty, events ordering, shortest path, Hamiltonian tour

This thesis explores some techniques of reducing duration uncertainty of events in planning and scheduling systems. Finding an effective ordering of events requires identifying events with shared stages. Grouping events with shared stages in close temporal proximity tends to eliminate the shared stages of the next recurring event which, in turn, eliminates the duration uncertainty associated with these stages. It is observed that events with shared stages have smaller standard deviations of their mean durations. Hence, finding the standard deviation of relative average duration of paired events allows intelligent agents to identify events with shared stages. A representation is presented based on the idea of a σ-graph. Once the σ-graph is built, a shortest path algorithm such as the Hamiltonian Tour can be used to traverse the graph to produce the final ordering of the events.

Table of Contents

List of Figures

Acknowledgments

I take great pleasure to extend my deepest thanks to Dr. Robert Morris who introduced me and guided me patiently to a very practical world of artificial intelligence. Without his extreme tolerance, supportive supervision and continuous help, this study would not have been possible. In addition, my sincere thanks to Dr. Dan Tamir for his great suggestions and the helpful moments from his valuable time that he spent on this thesis. I am also very grateful to Dr. Margot Haberhern for her remarkable comments and joyful suggestions. My sincere thanks go also to Ms. Margorie Beekett for spending the time to review this work. I would like also to thank all my fellows and friends for their valuable comments and helpful suggestions.

Dedication

To my parents

To my wife

To my children

To my family and friends

for their continuous help and support

بِسْمِ اللهِ الرَّحْمَنِ الرَّحِيْمِ

﴿ قُلْ إِنَّ صَلاتِي وَنُسُكِي وَمَحْيَايَ وَمَمَاتِي للهِ رَبِّ العَالَمِينَ ﴾

صدق الله العظيم

In the name of Allah, the Most Beneficent, the Most Merciful

"Say: Truly, My prayer and My service of sacrifice, My life and My death, Are (All) for Allah, The Cherisher of the Worlds."

Chapter 1 Introduction

1.1 Problem Statement

This thesis is aimed at studying how intelligent agents can use temporal information to learn about changes in their world. When looking at a set of events, one cannot always infer with certainty about their duration. This phenomenon is called *duration uncertainty*. Duration uncertainty is a major obstacle for developing intelligent agents that depend on the order of the events to accomplish a certain goal. Agents need to carry out a series of events with a prior knowledge of how long the whole task will take to accomplish. However, with the existence of duration uncertainty on some or all of the events, the agent will be unable to determine the total duration of the task. The problem is further complicated by the time and space constraints imposed by most environments in which agents act. This complication is due to, as we know, the fast response that is expected from the agent. For this reason, duration uncertainty "leads to the failure in the completion of plans and schedules" and requires a "time-consuming repair and revision" [12].

1.2 Objectives/Proposal

The goal of this thesis is to study duration uncertainties exhibited in events that occur repeatedly in dynamic environments. To minimize duration uncertainties, intelligent agents, such as humans, robots, or computer systems, will collect data about their world. Over time, these data will allow the agent a more accurate picture of the world. For example, a mail-delivery agent will use information about streets, road signs, and rush hours to learn the best route to follow to reach a certain address. On the other hand, a telescope scheduler will use the positions of stars, weather conditions, and user preferences in its attempt to build a robust schedule. Agents will be able to decompose the events occurring in their world into smaller pieces, called stages. Then, agents use similarities among these stages to infer an effective ordering of the events.

1.3 Significance of Work

This work is intended to help automate systems that operate in environments that change dynamically. The thesis presents a new approach to solving the problems faced in planning and scheduling systems. This approach makes use of statistical temporal information that intelligent agents can use to learn about their world. Using this information helps the agent adjust its knowledge about the world. Most of these environments currently have humans carry out the work

manually or have automated systems that lack the ability to account for dynamic changes in their world. With the proposed procedures for minimizing duration-uncertainties by using an effective ordering of events, such environments will be fully automated. Moreover, the intelligent systems doing the work will have the ability to adjust themselves to account for new changes in their domains.

1.4 Summary of Research

This thesis concentrates on researching old ideas, looking for new substitutes, and trying to consider the different environments that are applicable to the new concepts. A literature search to be familiar with the problems faced in planning and scheduling systems was essential. Other related issues such as clustering, temporal reasoning, and searching algorithms were also explored. Graph representations are basic building blocks in developing the ideas presented in this thesis, and algorithms such as Dijkestra's algorithm, the Traveling Salesman problem, the Near Neighbor algorithm, and the Hamiltonian Tour problem were found relevant. The thesis also deals with developing appropriate representations for the environments that are taken as examples in this research. These representations were carefully selected to account for the dynamic changes in the environment domains. Simulating these environments by using the Pascal language was the next goal.

One problem encountered at this point was devising a good simulation of the real data. Permutations and random data were used to simulate the actual data.

1.5 Summary of the Thesis

This thesis is organized in a way that will make sense to the reader. Chapter 1 introduces the reader to the main theme of the thesis and gives an overview of the topics at hand. The following chapters present the ideas discussed but in more detail. Chapter 2 begins with the background of the problem and the related topics. Both theoretical and application concepts are discussed focusing on the algorithms used and how the concepts were brought together. This chapter also gives an overview of the related material in the literature that was used as the basis for this research. Scheduling and clustering are two such examples. From there, Chapter 3 discusses the computational theory behind this research. The focus will be on efficient event ordering. Also discussed are the possible representations of events and how to deal with them in a dynamically changing environment. After that, Chapter 4 gives details about the different algorithms that are used to develop the techniques presented in this thesis. Implementation issues of these algorithms are also discussed. Last, Chapter 5 summarizes the findings of this research. This chapter consolidates the ideas presented throughout this thesis. A discussion of the problems encountered throughout the course of this research is included, and the

areas of possible enhancements are discussed. The discussion will encourage researchers to enhance the concepts presented in this thesis so that some of the problems encountered can be alleviated. To summarize, Chapter 1 serves as the front door to enter a new era of machine learning.

Chapter 2 Background

2.1 Properties of Environment

Throughout this thesis, the focus will be on environments that have properties that make them applicable to the analysis. Assume we have a set E of n events, E_1 to E_n. We assume they have the following properties [12]:

1. The events in E should be causally independent. This means that there should be:
   - No event E_i in E that prohibits the execution of any other event E_j.
   - No event E_i presumes the execution of any other event E_j.
2. Each event in E should have the tendency to exhibit duration uncertainties. This means that if we execute any event in E several times, then the duration of each execution is not fixed.
3. The duration uncertainties of part or all of the events should be eliminated if possible.

These properties are very important for our analysis because otherwise, the ideas presented in this thesis become unnecessary for the environment under study if such an environment lacks these properties.

2.2 Scheduling Difficulties

Scheduling is the process of allocating a finite set of resources to specific operations so that they obey the temporal restrictions of activities and capacity limitations of the shared resources [7]. Good schedules must reflect both the full detail of the operating environment and the influence of a conflicting set of preferences that range from global organizational objectives to specific operational constraints [13]. Some researchers view scheduling as a Constraint Optimization Problem (COP) [5, 6], and others view it as a Constraint Satisfaction Problem (CSP) [13]. Schedules can range from simple operations such as scheduling daily activities to very complex operations such as factory job-shop scheduling. The more constraints introduced in the scheduling problem, the more difficult it becomes to solve. Scheduling has been proven to be NP-Hard (belonging to a class of inherently intractable problems) [1, 8]. The situation in real-world scheduling environments is considerably more complex. Here is a list of constraints that should be considered in scheduling systems.

- **Scheduling Constraints**

These constraints should always be considered when scheduling a set of events. Failure to account for these constraints will result in unrealistic schedules.

* **Causal or Temporal Restrictions**. Typically there are precedence constraints associated with the sequence of operations for each activity as well as with the activities themselves. There are different temporal relations that relate operations and activities together. These restrictions are complicated further by other factors such as the maximum allowable time between operations. In addition, each individual operation possesses a well-defined set of resource requirements that must be satisfied for specific periods of time either before or during the execution of the operation [13].

* **Physical Restrictions**. All resources, especially machines, have specific capabilities that restrict the type of operation and the amount of load to be performed on them. In addition, these machines have specific operating characteristics, such as speed and setup procedures, that limit the amount of work carried out on them over a certain period of time [13].

* **Resource Unavailability**. In some cases, resources have unpredictable periods in which they are not available. Examples include backup times, break-downs, and sick or absent operators.

- **Preferential Constraints**

This set of constraints enhances the efficiency of the organization and the schedule itself and will vary from one organization to another. Some constraints, such as resource unavailability due to inefficient ordering of the scheduled events, are the cause of the scheduler itself.

* **Organizational Goals**. Some organizations have certain philosophies that govern the way their work is accomplished. Most of these goals are aimed toward customer satisfaction, stable creditability, and increasing profits. Some restrictions that organizations might impose include meeting order due dates, minimizing the processing time, maximizing resource utilization, and minimizing the amount of disruption caused by revisions of the schedule [13].

* **Optimization**. Due dates may be extended indefinitely into the future, and there may be many solutions to a scheduling problem, some of which are better than others. Hence, finding the ***best*** schedule is, almost always, the goal. The optimization process involves minimizing the amount of time between the activity's completion date

and its due date, minimizing the time required for the activity to be completed and minimizing the overall cost (money, effort, or resources usage) of activities [7].

* **Feasibility and Relaxation.** In some situations, a feasible solution that is within the time and cost constraints does not always exist. In such cases, the best partial schedule must be generated with some of the constraints relaxed. The problem is in finding the set of constraints to relax and how to relax them. Some of the constraints such as capacity constraints should not be relaxed, while others such as cost or due dates could be [7].

- **Unpredictable Consequences**

In some environments, the scheduler needs to account for some of the unpredictable situations in which resources become unavailable. These include machine break-down, failure of orders to pass intermediate quality-control inspection, introduction of engineering changes, or sick operators [13]. It is worth noting that this set of constraints is the main source of schedule inefficiencies. The reason is that these actions are of uncertain durations, and accumulating such durations might cause other actions to fail scheduling.

The list of different categories of constraints is not exhaustive. Since some of these constraints are introduced by the scheduler itself as a result of scheduling a subset of the events, duration uncertainty associated with the introduction of these new constraints is often the hardest to reduce. The reasons for this difficulty are the unavailability of prior knowledge about such constraints and the continuous accumulation and propagation of new duration uncertainties. The next section will build a framework for intelligent agents working in a dynamic environment.

2.4 Scheduling in a Dynamic Environment

The purpose of this thesis is to build a framework that will enable an intelligent agent to react to changes in a dynamic environment. "A dynamic environment is defined by a set of states and a set of physical laws that govern how the state of the environment changes over time" [3]. A system embedded in a dynamic environment is typically modeled with a set of states, a set of actions, and a state-transition function. The state-transition function models the physical laws governing the dynamic environment and determines the next state, given an initial state and an action performed by the system. Since changes in the environment occur with no prior knowledge of their existence, the agent will always be in a state of reacting to these changes. This behavior does not mean that the agent is totally unaware of what is happening in its world becuase the agent can define the boundary of its world. It is only when a change in the environment occurs within this boundary that the agent adjusts itself to this new change. Here, the agent senses any change in the environ-

ment, evaluates its state, and reacts only to changes that will make the agent move to another state. In one form of learning, called *reinforcement learning*, the agent moves from one state to another by following a state-transition diagram and a function that determines its next state [3]. Depending on the next state, the agent receives feedback from its environment in the form of rewards. Figure 1 shows an example of a *deterministic* environment in which the agent knows its next state. In this figure, nodes represent states labeled 0, 1, 2, and 3. Arcs represent actions labeled +, -, and @.

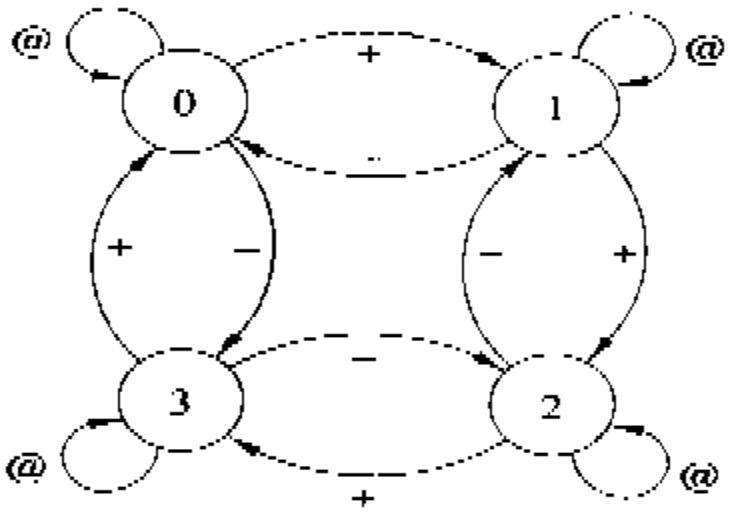

Figure 1 State-transition diagram for a deterministic environment.

In a *stochastic* environment, the next state for a given initial state and action is determined by a probability distribution on the set of states. Figure 2 shows the transition probabilities for a stochastic variant of the environment of Figure 1. The

transition probabilities are shown only for state 0. The transition probabilities for other states are identical. According to Figure 2, if the agent performs action + in state 0, then 70% of the time the agent will end up in state 1. In stochastic as well as deterministic environments, uncertainty makes learning the state-transition diagrams more complicated.

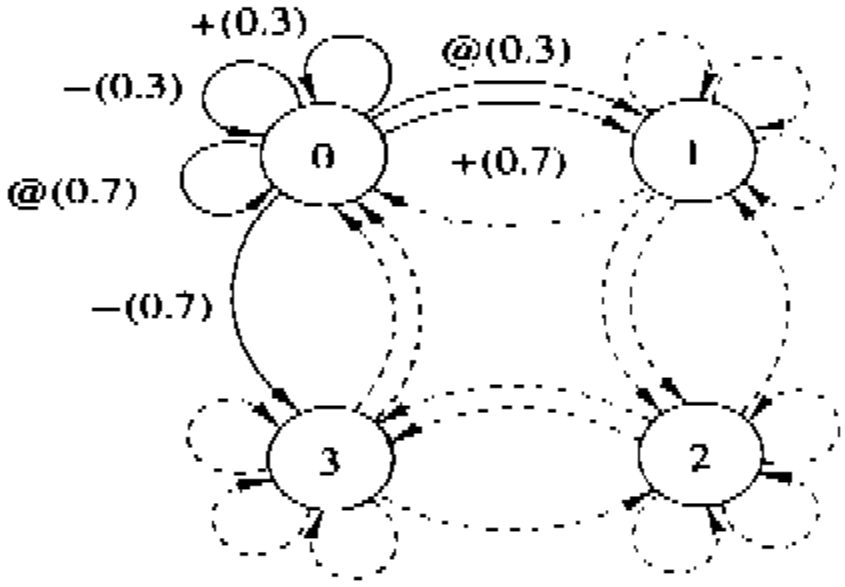

Figure 2 State-transition diagram for a stochastic environment.

Traditionally, state-based diagrams are used to model the behavior of intelligent agents. However, state-based representations are not effective, because states are well-defined entities and moving from one state to another means changing the actual state of the agent to accommodate itself for the following state. These hard

boundaries that define each state make it difficult for agents to learn the finer details of their environments. Temporal events, on the other hand, have durations which can be as large or as small as one desires. These types of events give finer granularities to the behavior of the agent and make the choice of selecting the right duration very easy. In addition, relations that define how states are related to each other are limited, which complicates the learning process. However, with durations, many temporal relations exist that give an agent much more freedom to determine where to move next. Before closing this chapter, the next section will introduce some of the potential applications that are applicable to the new techniques that will be discussed.

2.5 Potential Applications

After giving a brief discussion of the sources of difficulties encountered in scheduling systems, a demonstration of these difficulties will be presented by discussing two applications that will be seen throughout this thesis. These two examples are telescope scheduling and robot navigation. A detailed discussion of the telescope scheduling can be found in [4], whereas a similar setup for the robot navigation in an office environment is discussed in [3].

2.5.1 The Telescope Scheduling Problem

The telescope dealt with in this scheduling problem has an automated system to schedule observations. It electronically receives observation requests from as-

tronomer users. Each astronomer should specify the constraints on the observation such as the *observing window*, which is an interval of time, typically between one and eight hours, that indicates the applicable start and end times of the required observation. Each request should also include an *enablement interval* which is a subinterval of the observing window. This interval actually restricts the time to start the observation. All requests are put in a queue, and can remain in the system for weeks or months. During the day, the data are fed to a scheduler that produces a feasible, reasonable-scoring observing schedule. The resulting schedule is composed of *actions* that will be executed in the scheduled sequence. After an action completes execution, if the current time is outside the next action's enablement interval, then the schedule breaks and execution halts. Each action involves moving the telescope to a point at (or near) a star and then searching for a limited section of the sky in order to center the star within the telescope lenses. The amount of time required to complete the star centering process is impossible to predict. This difficulty makes predicting how long each action will take to execute more difficult and hence a better chance for the schedule to break.

To reduce the effects of schedule breakages, a group of NASA researchers has developed a technique called Just-In-Case scheduling, or JIC for short, that searches for places in the schedule where there is a chance for breaking. For each breakage point, JIC tries to find a new substitute schedule. JIC repeats the process for the newly generated schedules and then stores these schedules for later use.

When the telescope starts running, it starts executing the optimal version of the schedule. Throughout the night, if a breakage occurs, the telescope then selects the substitute schedule at that point.

Breakage can be caused by several factors. Weather conditions are the most important ones which make breakage unavoidable. Since the scheduler has to center the telescope on a star, the time it takes to carry out the process varies depending on where the telescope is before it starts centering. This centering process is the main cause of uncertainty. Additionally, schedule breakage could occur due to propagated duration uncertainties that make some actions fall outside their enablement interval.

2.5.2 Robot Navigation

In this domain, a robot is set to deliver mail to addresses in a crowded city. For this robot, states are delivery points, and actions correspond to the robot's traversing a route from one address to another. The robot is capable of traversing the streets by using an initial map. It collects information about street segments, road signs, traffic lights, rush hours, and accidents on its way from the post office to the delivery points. The robot continually evaluates its position from its destinations by taking into account the new information that it collects. Hence, the robot can potentially change its route to reach the current or a closer destination. The concern of this example is to show how an agent, a robot in this case, could learn about its environment. The assumption is that the robot has the capability to move through street

segments and react to any change in the environment. No assumptions regarding how the robot can perform such actions are of concern to the study of this thesis.

2.6 Summary

This chapter presents a brief background on scheduling and introduces some of the basic concepts that will be used for the rest of this thesis. The chapter shows scheduling as one area of artificial intelligence that deeply suffers from the problem of duration uncertainty. Scheduling difficulties give the reader a sense of the difficulty of reducing the effect of duration uncertainty discussed in this thesis. This chapter also clearly defines the notion of dynamic environment. To introduce the reader to the theme of this thesis, a brief discussion is given on how scheduling in a static environment differs from that of a dynamic one. Then, the two sample applications that are discussed throughout this thesis are introduced. The introduction of these two examples will further motivate the reader to continue with the remainder of the thesis until the proposed solutions are reached.

Chapter 3 Uncertainty and Learning

3.1 Duration Uncertainty and its Effects

In scheduling a set of tasks, one often cannot determine with certainty how long each event will take. Events normally occur in a repeated fashion. For example, Figure 3 shows four executions of an event over a period of time. The duration of the event varies from execution to execution thus causing what is called *duration uncertainty*. Duration uncertainty is depicted as a difference in the lengths of each line representing a single duration. Duration uncertainty makes it difficult to anticipate how long the next occurrence of the event will take.

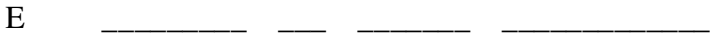

Figure 3 Four occurrences of a repeating event.

Duration uncertainty is defined as the set of possible durations of an event [10]. That is, the total variations in the lengths of each execution of the event give a measure of the duration uncertainty of that event. For example, in a telescope-

scheduling environment, each observation requires the telescope to be centered on a star. The time needed to accomplish such a task depends on how accurately the telescope is pointed when it starts the centering search and how clear the sky is. This centering process makes it impossible to predict exactly how long each observation will take [4]. Duration uncertainty is undesirable for intelligent agents, because it leads to the unsuccessful completion of plans and schedules and thus requires extensive repair and revisions. Because there are many causes of duration uncertainty, scheduling systems that account for duration uncertainties are often not sufficiently robust. Duration uncertainty is often inherited in some periods or *stages* of the event. A stage can be seen as one action that is performed as part of performing the event [12]. A stage is normally the smallest entity that comprises events.

For example, in telescope scheduling, each observation consists of three stages: first, moving the telescope to a point at (or near) a star; second, searching a limited section of the sky in order to center the star within the telescope lenses; third, taking an instrument reading [4]. Although not every stage has an intrinsic duration uncertainty, the propagation of duration uncertainties from earlier stages in the event causes the subsequent stages to contribute to the total duration uncertainty of the event. The propagated duration uncertainty might cause the next scheduled events to fail in execution. Figure 4 shows how the propagation of duration uncertainty causes a schedule breakage in a telescope-scheduling system. Note that due to the propagated duration uncertainty from action 0, action 1 is expected to start anytime between

17 and 21 and to finish anytime between 25 and 37. The reason for the schedule breakage is that the next observation to execute lies outside its enablement interval.

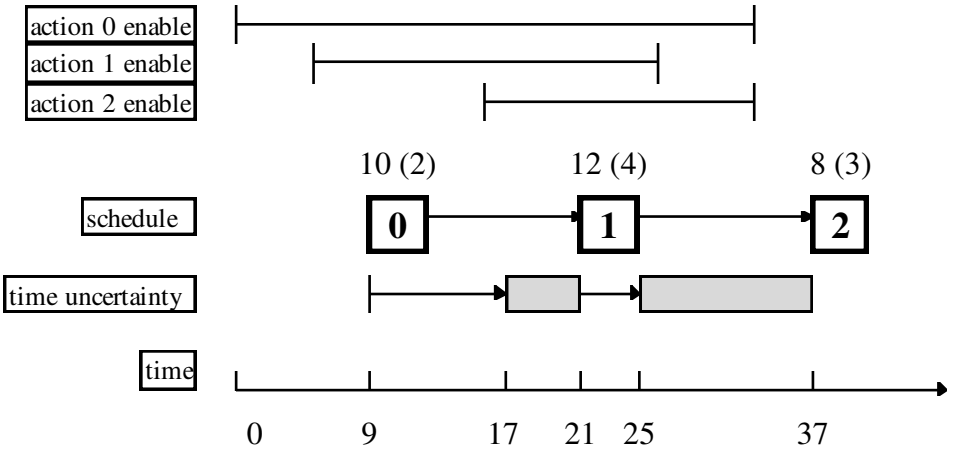

Figure 4 Propagation of temporal uncertainty.

3.2 Event Interaction

Suppose that each morning an intelligent agent performs three tasks: clean the car (C), read the e-mail (M), and type the daily column to the local newspaper (N). When each task is considered in isolation from the other two, the result is that the agent cannot predict how long the next occurrence of each event will take. However, by using intuition, one could realize that each of these tasks is composed of several stages. For example, cleaning the car requires preparing the cleaning utensils, walking to the car, cleaning the car, and then returning the cleaning utensils. On the other

hand, sending the daily column to the newspaper consists of turning on the computer, running Microsoft Windows, executing the word processor, typing the column, printing it, and sending it to the newspaper. Each of these stages could be further decomposed into finer stages. Hence, by using intuition, one could perform events M and N together, either before or after event C. Doing so will eliminate some setup stages for either M or N such as booting the computer and starting Microsoft Windows. This process is depicted in Figure 5. In this figure, the three events behave in the same way of unexpected duration for each execution when each event is executed separately. However, when events M and N are paired, the agent will be able to expect how long each occurrence of event N will take (this is depicted in the figure by the bars of almost equal length). The reason is that some of the stages in event N that exhibit duration uncertainty have been eliminated due to the pairing with event M. From this example, one can notice that one way to minimize or at least reduce duration uncertainties is to group together events that share stages. This reduction is possible because putting events with shared stages in close temporal proximity means eliminating the shared stages of the next recurring event. Elimination of these shared stages will, in turn, eliminate the duration uncertainty associated with these eliminated stages.

C ____ _________ _____ _______

M ___ _______ ___ _____

N ____ ____ ____ ____

Figure 5 Three tasks that an agent performs every morning.

As another example, assume that the mail-delivery robot is delivering mail to three addresses A, B, and C as shown in Figure 6. Because address A occurs in the same route that connects address C with the post office (PO), one could realize that following the routes to address A, then address B, and then address C will not eliminate any of the duration uncertainty associated with each route. However, if the robot delivers mail to address A followed by address C, then the duration as well as the duration uncertainty for the segment from the post office to address A will be eliminated from the route to address C.

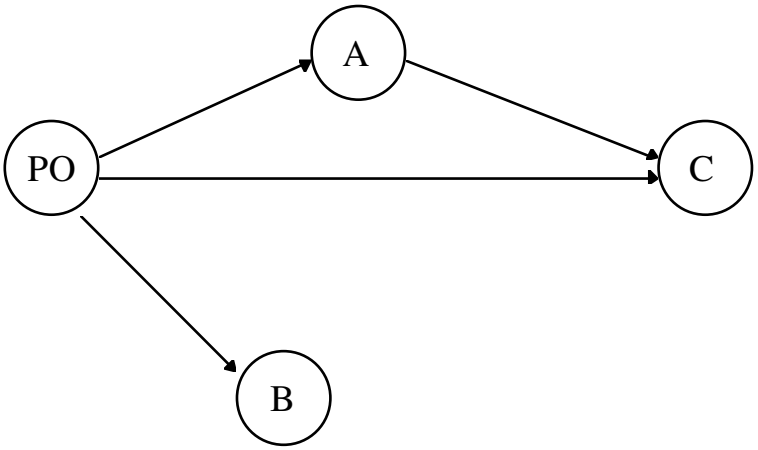

Figure 6 Three delivery addresses for the mail delivery robot. Each segment of the street has an associated duration uncertainty.

Interestingly, eliminating shared stages does not always eliminate duration uncertainty, , , dbecause the eliminated stages might not be the ones contributing to duration uncertainty. For example, assume that there are two events with stages A, B, and F for the first event and C, D, and F for the second. If duration uncertainty is caused by stage F, then eliminating stage F from one of the events will reduce the total duration uncertainty from both events. However, if duration uncertainty is caused totally by stages A and C, then eliminating the shared stage F will have no effect on reducing the duration uncertainty [10]. Although in Figure 6 intuition is used to determine the events with shared stages, intelligent agents need other means to recognize the events that have shared stages. The next section suggests that knowledge from experience could be used to infer events with shared stages through the concept of relative durations.

3.3 Relative Durations

As stated previously, intelligent agents have no prior knowledge about the relation among events occurring in their world. However, agents can induce this information by looking at the relative duration of pairing two events. Let us define the relative duration of event e with respect to event e' (rd (e, e')) as the duration of e when e immediately follows e' [10]. Relative duration can be calculated by executing all the events in a set in every possible linear temporal order. For example, a mail-delivery robot can use the data it collects over time to determine the relative durations of the possible street segments to follow from its current segment. Relative duration measures how an event behaves when that event immediately follows another [10]. When two events that have shared stages are executed in sequence, then, as explained in the previous section, some of the stages in the next recurring event will be eliminated. This reduction of stages will eventually make the total duration of the second event as well as its total duration uncertainty shorter than when it follows another event with which it can not share stages.

However, events' pairing does not always result in uniform durations for the next recurring event, because the latter event might still have other stages with duration uncertainty that cannot be shared with the first event. Hence, one should instead consider the relative average duration. The relative average duration of an event e with respect to an event e' is the average duration of e when immediately followed by e', over a set of occurrences of e and e' [10]. For example, in the robot navigation

problem, the robot gets its initial knowledge about the city that it operates in by using a map. Then, with each delivery, the robot updates its knowledge about how long each route takes and the possible delays encountered. As the number of deliveries increases, the robot becomes more and more confident of what route to follow.

Relative average durations eliminate the fluctuations in relative durations that are caused by stages with duration uncertainty that cannot be shared with other events. Another useful piece of information that an intelligent agent can draw from relative average durations is the standard deviation of the relative average durations. Standard deviations of relative average durations tell the intelligent agent how stable the two events are when paired. In other words, the lower the standard deviation, the higher the probability of having shared stages.

To illustrate these concepts, let us look at four events A, B, C, and D that an agent performs each day. Figure 7 shows the durations of five occurences of each of these events. In this figure, durations are in time units and represent the time it takes the agent to perform each event. The "Total," "Average," and "Std" columns represent the total durations, their average, and their standard duration for each event over the five days. This figure shows that when the four events occur in the order given, they require the durations shown in each day. For example, in Day 1 event A is completed in 3 time units, event B in 7 time units, event C in 3 time units, and event D in 11 time units. Here, we are interested in looking at what happens to the durations of an event when it follows others. For example, we notice from Figure 7 that the stand-

ard deviation of event B when it follows event A is small compared to that of the other events. The same anology applies to event C when it follows event B. These low standard deviations could indicate that their corresponding events share stages with their predecessors. However, we cannot make such a conclusion by looking at only one order of these events.

	Day 1	Day 2	Day 3	Day 4	Day 5	Total	Average	Std
A	3	5	9	4	2	23	4.6	5.42
B	2	6	1	4	3	16	3.2	1.72
C	3	2	4	4	2	15	3	0.89
D	11	8	7	1	10	37	7.4	3.50

Figure 7 One possible order of four events that an agent performs daily for five days.

Now, consider two other possible orders in which these four events could occur. Figure 8 shows two such orders. In the first order we notice again that event B occurred after event A, and event C occurred after event B. This time, however, we notice that the standard deviation of event B is 3.07 which is much higher than that of Figure 7. The different standard deviations mean that events A and B probably do not share stages. On the other hand, event C still has low standard deviation, which means a higher probability of sharing stages with event B. However, when event C followed event B, as in the second order of Figure 8, C did not show a reduction in the total durations which, again, means that B and C might not share stages.

	Day 1	Day 2	Day 3	Day 4	Day 5	Total	Average	Std
D	8	4	12	9	11	44	8.8	2.79
A	7	10	3	5	9	34	6.8	2.56
B	10	5	11	9	3	38	7.6	3.07
C	6	3	5	2	4	20	4	1.41
A	5	9	1	8	7	30	6	2.83
D	12	4	5	9	6	36	7.2	2.93
B	10	1	8	11	9	39	7.8	3.54
C	3	12	5	4	10	34	6.8	3.54

Figure 8 Two more possible orders of the events of Figure 7.

The above discussion demonstrates the use of relative average durations and their standard deviations. To calculate the relative average duration, we average all occurences of one event when it follows another in all possible temporal orders. For example, in Figure 7 and Figure 8, event B occurred ten times after event A, and event C occurred fifteen times after event B. Using this information, the relative average duration of event B when it follows event A is $\frac{16+38}{10} = 5.4$ time units, and the relative average duration of event C when follows event B is $\frac{15+20+34}{15} = 4.6$ time units. The standard deviations for the two cases can be calculated in the same manner to yield $\sqrt{\frac{402}{10} - 5.4^2} = 3.32$ and $\sqrt{\frac{433}{15} - 4.6^2} = 2.78$, respectively.

3.4 Representing the Knowledge

The knowledge that an intelligent agent collects about events will need to be organized in a structural way to ease extracting the needed information on time. One characteristic of the data structure to use is not only to store the knowledge, but also to make it easy to infer new knowledge from it. One structure that has these properties will be called the σ-graph,,,,,,,,,,,,,,,, a weighted undirected graph. Figure 9 shows the σ-graph for the events in Figure 5. Each node in a σ-graph represents an event, and every edge is labeled with the standard deviation of the relative average duration of the two events that the edge connects. Based on intuition, one could show that the lower the value on the edge, the higher the probability that the two events have shared stages.

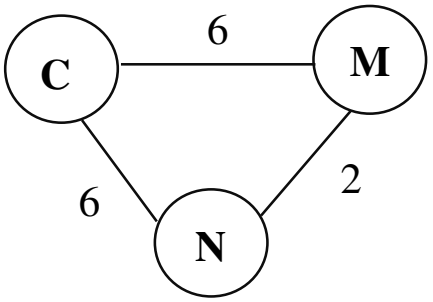

Figure 9 A σ-graph with three events.

A σ-graph enables intelligent agents to infer events with shared stages. To illustrate, assume that an agent, besides the three tasks C, M, and N already described,

is to do two other tasks: *clean the kitchen* (K) and *go to the store* (S). An incomplete σ-graph for these five tasks is shown in Figure 10. This graph is incomplete because each node should be connected with every other node. Again, the lower the values on an edge, the higher the probability of having shared stages between the two events that the edge connects.

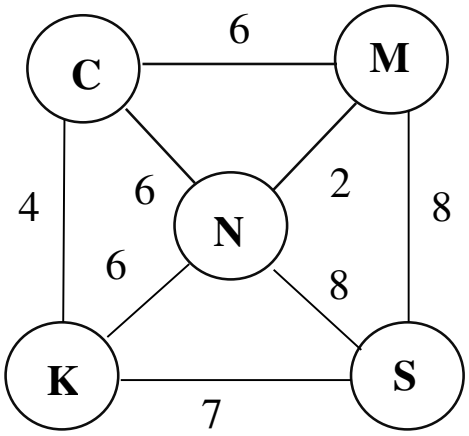

Figure 10 A σ-graph for the five events that an agent performs each morning.

Now, the problem of generating an efficient ordering of these events reduces at this point to the task of finding the shortest path through all the nodes of this graph [10]. One such method is finding a minimal spanning tree by using an algorithm such as Kruskal's algorithm [9]. Applying Kruskal's algorithm to a σ-graph will result in a spanning tree with minimal weight. Then, using a threshold for the standard deviation to identify events with shared stages, one could repeatedly generate a spanning tree,

cut the longest edge, and apply Kruskal's algorithm to this new σ-graph. The process continues until all weights in the edges of the σ-graph are higher than the threshold. Figure 11 depicts one step in the process after merging events M and N. This figure shows events M and N paired because of the low standard deviation of their relative average duration.

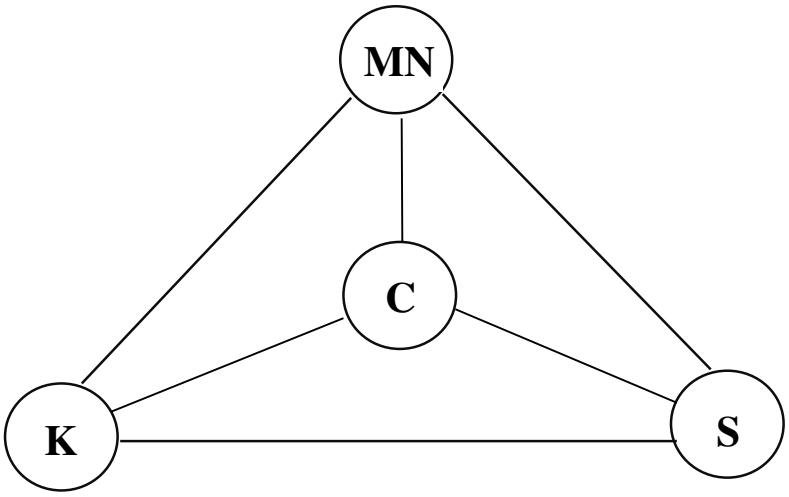

Figure 11 The σ-graph of Figure 10 after applying Kruskal's algorithm and the concept of threshold to identify events with shared stages.

3.5 Summary

This chapter starts the discussion on duration uncertainty and its effects. It shows the severity of introducing duration uncertainty in scheduling and planning systems. Then, the chapter started building a mathematical model for duration uncertain-

ty and topics related to it. First, the concept of relative duration was introduced. Then, two more useful concepts were drawn from relative duration: :: relative average duration and its standard deviation. These two concepts give a formal explanation of how intelligent agents could detect events with shared stages. After that, the chapter introduced the problem of representing the knowledge that intelligent agents gather over time about their world. The concept of the σ-graph was introduced and the technique of finding an effective events ordering using this σ-graph was also given.

Chapter 4 Implementation and Results

4.1 The Hamiltonian Tour Algorithms

The previous chapter discussed duration uncertainty and showed how unsuccessful plans and schedules are when duration uncertainty is introduced in them. The chapter also showed that by using the standard deviations of relative average durations of events, intelligent agents can learn to reduce the effects of duration uncertainty by grouping together events with low standard deviation. Low standard deviations was interpreted to mean a higher probability of shared stages. This chapter will discuss an algorithm that will find an effective ordering of events despite duration uncertainty. This algorithm uses the σ-graph introduced in the previous chapter. The concern here is to find the shortest path in the graph that will visit all nodes only once after starting from a given source node. This path is called a Hamiltonian Tour problem [2], which is a special case of the traveling salesman problem except that there is no return to the source node. Although the Hamiltonian Tour problem is NP-Complete [2], there are approximation algorithms that give good results. However, in some cases, especially when the number of nodes in the graph is small, it is best to find an op-

timal shortest path. Examples of the approximation algorithms that will be discussed next include Kruskal's, Prim's, and the Nearest Neighbor algorithms.

4.1.1 An approximation Algorithm to the Hamiltonian Tour Problem

The approximation algorithm discussed here basically selects a source node in the graph, finds a minimum spanning tree, and then lists the nodes in a preorder traversal [2]. Figure 12 shows a pseudo code of the algorithm. Determining what source node to start with depends on the current state of the intelligent agent. For example, a mail-delivery robot will take the next intersection as the source node and the remaining delivery addresses as the other nodes in the graph. This algorithm uses a graph G such as the σ-graph of Figure 10. The set of vertices V represents events, and the cost vector c represents the weights on the edges.

```
Approx-Tour(G, c)
1. select a vertex r ε V[G] to be a "root" vertex
2. grow a minimum spanning tree T for G from root r using PRIM(G, c,
r)
3. let L be the list of vertices visited in a preorder tree walk of T
4. return the Hamiltonian tour H that visits the vertices in the order L
```

Figure 12 The algorithm that approximates the Hamiltonian Tour.

There are several algorithms that can find a minimum spanning tree from a root node, such as Kruskal's and Prim's. Both algorithms are greedy in the sense that they select the edge with the minimal weight from any given node [2]. For the sake of

this discussion, Prim's algorithm, shown in Figure 13, is selected to generate the minimal spanning tree. In this algorithm, the graph G has a set of vertices V and a set of edges E. The weights in the edges are stored in an array w, and the root node is designated as r. A priority queue Q is used to hold all the edges that are not yet in the spanning tree. An adjacency matrix Adj holds the edges going out of each node. The key array holds the shortest path from any node to the root. The π array holds the parent of each node. This information is used to tell the order in which the spanning tree was traversed. At each step in the **while** loop, a node with the minimum distance is extracted from the priority queue and put in the spanning tree if it is not there yet. The performance of this algorithm, as well as the implementation of EXTRACT-MIN, depends on how the priority queue is implemented.

```
Prim(G, w, r)

1.   Q ← V[G]
2.   for each u ε Q do
3.     key[u] ← ∞
4.   key[r] ← 0
5.   π[r] ← NIL
6.   while Q ≠ 0 do
7.     u ← EXTRACT-MIN(Q)
8.     for each v ε Adj[u] do
9.       if v ε Q and w(u, v) < key[v] then
10.        π[v] ← u
11.        key[v] ← w(u, v)
```

Figure 13 Prim's algorithm to find the minimum spanning tree of a graph G with a set of vertices V and a set of edges E.

After finding the minimum spanning tree by using Prim's algorithm (line 2 of Figure 12), a preorder tree walk on the resulting spanning tree is performed. In a preorder tree walk one recursively visits every node in the tree, listing a node when it is first encountered, before any of its children are visited. Figure 14 shows the steps being discussed. In this figure, the given set of points lie on vertices of an integer grid. The ordinary Euclidean distance is used as the cost function between two points. Figure 14 (b) shows the minimal spanning tree T of these points, as computed by PRIM. Vertex a is the root vertex. The vertices happen to be labeled in such a way that they are added to the main tree by PRIM in alphabetical order. Figure 14 (c) shows a preorder walk of the tree T listing a vertex just when it is first encountered, yielding the ordering a, b, c, h, d, e, f, g. Figure 14 (d) shows a tour of the vertices obtained by visiting the vertices in the order given by the preorder walk. This tour is the Hamiltonian tour H returned by Approx-Tour. Note that with the Hamiltonian tour of Figure 14 (d), the approximation algorithm returns the shortest path with a cost of 19.074 time units. The next section will show that one can do better by finding an optimal tour.

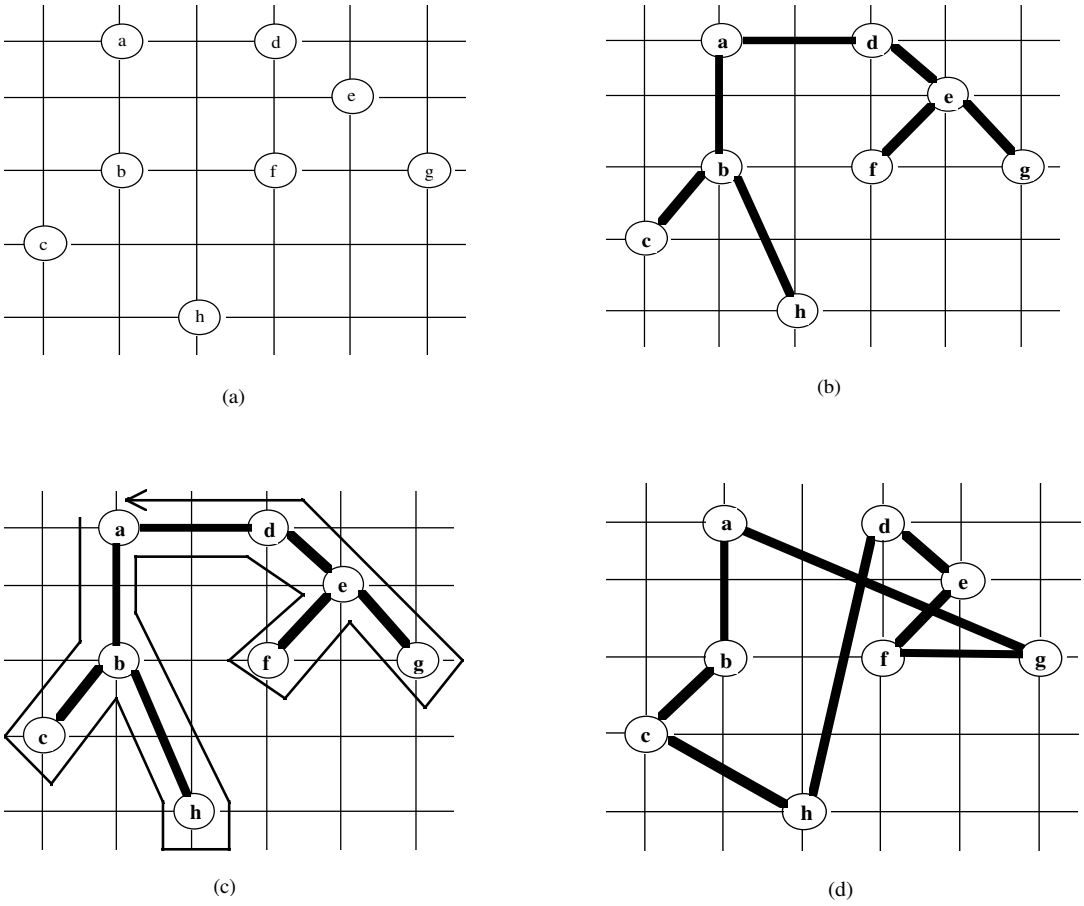

Figure 14 The operation of Approx-Tour algorithm.

4.1.2 Finding the Optimal Tour

As already stated, for a graph with a small number of nodes, one can find an optimal shortest path in polynomial time. Figure 15 shows the pseudo code for such an algorithm, that , ,,,,ddddddexplores every possible path from the root node. When this algorithm visits all the nodes, it compares the current path to the shortest path obtained so far. If the current path is shorter, it sets the shortest path to be the current

path. This algorithm calculates (n-1)! paths from the root node for a graph with n nodes. Hence, using this algorithm takes a prohibitively long time with a graph of just few nodes. This restriction makes it difficult for intelligent agents to use such an algorithm because a quick response is expected from them. Figure 16 shows the optimal shortest path for the graph in Figure 14 (a). Here, the cost of this optimal path is only 14.715 time units.

```
OptimalTour(G, V, w, u, min)
1.    visited[u] ← true
2.    for each v ε V do
3.      if visited[v] ≠ true then
4.        sum ← sum + w[u, v]
5.        path ← path + {v}
6.        Optimal-Tour(G, V, w, v, min)
7.        visited[v] ← true
8.    if sum < min then
9.      min ← sum
10.     shortest-path ← path
```

Figure 15 Pseudo code for finding the optimal path.

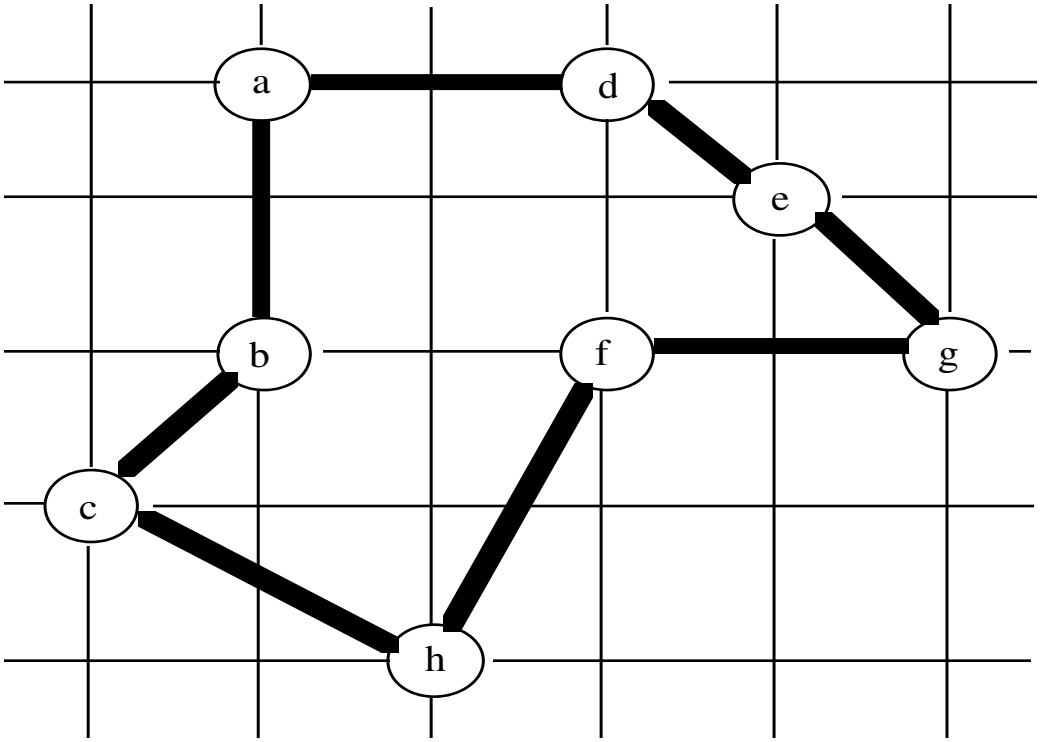

Figure 16 An optimal tour *H* for the set of vertices given in Figure 14 (a).

4.2 Implementing the Hamiltonian Tour Algorithm

To test the validity of the methods being developed in this thesis, the algorithms discussed in the previous section were implemented by using Turbo Pascal 7.0. The intention was to get a feeling of the time and space requirements of these algorithms. While trying to do the implementation, several problems arose such as simulating the actual data. To make the tests realistic enough, a data file was generated to contain all the permutations of a set of events. For each event, ten random numbers were generated to simulate the durations of ten executions of each event. Figure 23 (in the appendix) shows the Pascal program that generated the simulated data, and

Figure 17 shows an example of the different permutations that could be generated by the program of Figure 23.

A	12	8	11	11	6	4	12	9	7	8
B	6	3	2	7	4	5	5	2	4	7
C	2	8	6	6	9	12	1	1	11	1
D	11	6	1	11	9	9	5	1	11	10
D	6	4	8	5	6	1	4	10	8	9
C	5	7	7	8	8	3	8	7	12	2
A	1	4	10	7	11	3	11	4	2	1
B	5	7	2	6	3	7	5	4	7	2
C	10	7	6	9	11	5	8	4	12	7
A	3	9	5	10	3	3	9	6	1	11
B	6	3	1	4	5	4	7	2	1	3
D	12	5	4	9	8	2	11	3	5	6

Figure 17 Three sample permutations that could be seen in the generated file of simulated data.

Once the simulated data were at hand, the problem of converting raw data into a σ-graph was explored. First, the relative duration, relative average duration, and the standard deviation of each pair of events were calculated. Figure 24 (in the appendix) shows the Pascal program used to do these calculations. This program generates an output file, like the one in Figure 18, with the event pairs as edges and the standard deviation of each pair as the weights on the edges. After calculations were performed, the σ-graph was built by using the list of events as the nodes, the pairing of events as the edges, and the standard deviation of each pair as the weights on the edges. Figure

25 (in the appendix) shows the Pascal procedure that loads the calculated data and builds the σ-graph.

A B	2.209
B C	2.401
C D	2.370
B D	2.221
D C	2.291
A C	2.272
B E	2.232
A D	2.276
D B	2.222
A E	2.275
C E	2.310
D E	2.270

Figure 18 A sample output that could be generated by the program in Figure 24.

4.2.1 Implementing an Approximate Hamiltonian Tour Algorithm

To implement an approximate algorithm for the Hamiltonian Tour, the Nearest Neighbor algorithm was used. This algorithm, like Kruskal's and Prim's algorithms, is a greedy one that selects the edge with the shortest path from the currently visited node. Figure 26 (in the appendix) shows an implementation of this algorithm. This implementation works well for a σ-graph since it is a complete graph. However, the Nearest Neighbor algorithm will fail for incomplete graphs unless this algorithm uses backtracking to account for nodes with no outgoing edges. Figure 19 shows the results obtained using this algorithm. In this figure, each line shows the cost for the approximate shortest path when each node in the σ-graph is taken as a source node.

```
Approximate shortest path from node A: A E D G F B C  = 20.580 time units
Approximate shortest path from node B: B F D G C E A  = 20.625 time units
Approximate shortest path from node C: C F D G A E B  = 20.602 time units
Approximate shortest path from node D: D G F B E A C  = 20.613 time units
Approximate shortest path from node E: E D G F B C A  = 20.637 time units
Approximate shortest path from node F: F D G C E A B  = 20.684 time units
Approximate shortest path from node G: G F D A E C B  = 20.615 time units
```

Figure 19 The results obtained using the Nearest Neighbor algorithm.

4.2.2 Implementing the Optimal-Tour Algorithm

For the comparisons of this chapter to make more sense, assume that an implementation of the algorithm that finds the optimal shortest path was written. Figure 27 (in the appendix) shows this implementation. This algorithm was fast enough for the number of sets tested. Figure 20 shows the results that were obtained using this algorithm. Comparing these results with those in Figure 19, one would not see a big difference, although the paths are different in some cases. The reasons for obtaining almost the same results are that the weights in the σ-graph that were used to calculate the shortest paths are relatively narrow in range, and that the number of nodes in the graph are also relatively small.

Optimal path from node A: A E D G C F B = 20.576 time units

Optimal path from node B: B G C F D A E = 20.570 time units

Optimal path from node C: C F B G A E D = 20.584 time units

Optimal path from node D: D A E C F B G = 20.585 time units

Optimal path from node E: E D A C F B G = 20.604 time units

Optimal path from node F: F B G C D A E = 20.585 time units

Optimal path from node G: G F B C D A E = 20.591 time units

Figure 20 Results obtained using the optimal shortest path algorithm implemented in Figure 27.

Figure 21 shows the results obtained by the two algorithms when weights of a wider range and a larger number of events were used. As stated before, intelligent agents that deal with ordering a few events that do not require a real-time response can use the optimal path algorithm. However, if time is a constraint, such as in telescope scheduling when each moment counts, using an approximation algorithm is more feasible. Further, in cases where the number of events is prohibitively large, an approximation algorithm makes more sense.

	Case 1 = 5 events		Case 2 = 8 events		Case 3 = 10 events	
	Path	Cost	Path	Cost	Path	Cost
Near Neighbor from node 1	1 4 5 3 2	86.000	1 7 5 6 3 2 4 8	133.000	1 2 10 3 5 4 8 7 9 6	103.000
Optimal path from node 1	1 4 5 2 3	75.000	1 7 5 6 3 2 4 8	133.000	1 9 6 3 10 5 4 8 2 7	71.000
Near Neighbor from node 2	2 3 5 4 1	186.000	2 4 1 7 5 6 3 8	142.000	2 1 10 3 5 4 8 7 9 6	103.000
Optimal path from node 2	2 1 4 5 3	109.000	2 4 1 7 5 6 3 8	142.000	2 1 9 6 3 10 5 4 8 7	50.000
Near Neighbor from node 3	3 5 2 4 1	183.000	3 2 4 1 7 5 6 8	138.000	3 1 2 10 4 8 7 9 6 5	80.000
Optimal path from node 3	3 1 4 5 2	104.000	3 2 8 5 6 1 7 4	106.000	3 1 9 6 7 10 5 4 8 2	53.000

	Case 1 = 5 events		Case 2 = 8 events		Case 3 = 10 events	
	Path	Cost	Path	Cost	Path	Cost
Near Neighbor from node 4	4 5 3 2 1	168.000	4 1 7 5 6 3 2 8	117.000	4 1 2 10 3 5 7 9 6 8	112.000
Optimal path from node 4	4 5 3 1 2	131.000	4 1 7 5 6 3 2 8	117.000	4 3 1 9 6 8 2 7 10 5	51.000
Near Neighbor from node 5	5 3 2 4 1	186.000	5 6 1 7 4 8 2 3	257.000	5 1 2 10 3 6 7 9 4 8	108.000
Optimal path from node 5	5 3 1 4 2	115.000	5 6 1 7 4 3 2 8	135.000	5 4 3 1 2 10 9 6 8 7	49.000
Near Neighbor from node 6			6 1 7 5 2 4 8 3	213.000	6 1 2 10 3 5 4 8 7 9	95.000
Optimal path from node 6			6 1 7 4 3 2 8 5	132.000	6 3 1 9 4 8 2 7 10 5	63.000
Near Neighbor from node 7			7 5 6 1 4 8 2 3	267.000	7 1 2 10 3 5 4 8 9 6	114.000
Optimal path from node 7			7 4 3 2 8 5 6 1	136.000	7 1 9 6 3 10 5 4 8 2	62.000
Near Neighbor from node 8			8 5 6 1 7 4 3 2	132.000	8 1 2 10 3 5 4 6 7 9	97.000
Optimal path from node 8			8 5 6 3 2 4 1 7	119.000	8 2 1 9 6 7 10 5 4 3	55.000
Near Neighbor from node 9					9 1 2 10 3 5 4 8 7 6	136.000
Optimal path from node 9					9 6 3 1 2 10 5 4 8 7	49.000
Near Neighbor from node 10					10 1 2 7 9 6 8 4 3 5	122.000
Optimal path from node 10					10 5 4 3 1 9 6 8 2 7	71.000

Figure 21 The results obtained using the Nearest Neighbor and the Optimal Path algorithms for three cases with varying number of events per set.

4.3 An Approximation Algorithm for the Robot-Navigation and the Telescope-Scheduling Problems

Chapter 2 introduced the telescope scheduling and the robot navigation problems, and Chapter 3 gave examples of how to minimize the effect of duration uncertainty in these two problems. This section will collect the ideas presented so far to show an algorithm to solve each of these problems. Each of these algorithms puts into account the dynamic changes of the environment in which the agent runs.

4.3.1 The Robot Navigation Problem

The following are the initial steps that a robot follows before starting the mail-delivery process. These steps will be carried out only on the first day of the robot's

operation. These steps are necessay to let the robot gain an initial knowledge of the street segments in the city.

1. Read and store information found in a streets' map of the city.
2. Calculate the relative average duration and the standard deviation of each street segment. The length of the segment and the posted speed on that segment will be used to calculate the expected time to cross that segment.
3. Build the σ-graph from the relative durations and the standard deviations of each segment obtained in step 2.

After performing these initial steps, the robot starts the mail delivery process. The following is the algorithm that the robot follows on its way to each address. The robot will carry out these steps each time it is assigned to deliver mail.

1. Use an approximate or optimal algorithm to find the shortest path from the post office to each delivery address.
2. The robot starts following the shortest path obtained from the previous step.
3. While the robot crosses each segment of the path, it updates the information about the length of the current street segment, the time spent so far on this segment, the average speed, and any street problems encountered so far. Then, during certain intervals, the robot calculates a new relative average duration and standard deviation for the current segment, using the

time spent so far and the old relative average duration and standard deviation. If the new standard deviation starts to outweigh the old one, then the robot starts finding alternative routes. The robot repeats this process by taking the next intersection as the starting point.

4. When at an intersection that is one of the delivery addresses , the robot deletes it from the list of delivery addresses. Otherwise, the robot adjusts the duration uncertainty of the current segment which will be the mean of the previous duration uncertainty and the current one. The current duration uncertainty could be found as the difference between the old standard deviation and the calculated one. The relative average duration and the standard deviation for the current segment will be updated to reflect the calculated data.

4.3.2 The Telescope-Scheduling Problem

Because the number of possible break points in a telescope schedule can be large, the scheduler must arrange the observations so as to minimize the number of breakages. As mentioned earlier, one place that duration uncertainty plays a role in is when centering the telescope on a star. Here, the star-centering process of the telescope is considered a setup stage for the observation process. The claim is that one could minimize the duration uncertainty of these setup stages by arranging the observations so that the time it takes to rotate the telescope from one location to another is minimal. Assume that each observation action, A_i, has two coordinates, x_i and y_i.

The following algorithm will try to minimize duration uncertainties by ordering the actions so that the setup time for each action is minimal.

1. Sort actions in ascending order of enablement intervals, E_i. As already mentioned, an enablement interval is the interval of time in which an observation can actually start. This step is important for the scheduler to determine the earliest action with which to start. Add this action to the generated schedule.

2. Start with the first action in the list. Call this action A_s. This action will have coordinates x_s and y_s.

3. If A_s is the last action, then quit.

4. Otherwise, calculate the angle necessary to rotate the telescope from the coordinates of A_s to the coordinates of every other action A_i. The assumption here is that the telescope can be rotated both clockwise (positive angle) and counter-clockwise (negative angle). The distance the telescope travels between these coordinates will be the absolute value of the rotation angle. Next, find the minimum of all the distances that the telescope must travel from the coordinates of A_s to the coordinates of all other actions. Mark the action A_i with the minimum distance as selected. Call this action A_n for next action.

5. If E_n (the enablement interval for A_n) falls within F_s (the set of finish intervals of action A_s), then

- Add A_s to the generated schedule.
- Unmark action A_n.
- Set A_n to A_s (the next starting action).

6. Repeat steps 3-5.

The above algorithm will select the best schedule that is robust enough among all other possible schedules . The telescope system will start executing this schedule. Because no consideration of weather conditions was put into account when this schedule was generated, the schedule will still have chances of breaking but at no expected points. However, if a schedule break should occur, then this algorithm could be applied to the remaining observations by taking the break point as the start action.

4.4 Summary

In this chapter, the techniques discussed since the beginning of this thesis were laid out on a solid ground. This chapter focused on the algorithms that find an effective ordering that minimizes the effect of duration uncertainty. The Nearest Neighbor and the Hamiltonian Tour algorithms were discussed and a mathematical model for implementing them was first given. Then, the chapter showed how these two algorithms were implemented by using Turbo Pascal. The problems faced during the im-

plementation stage as well as the results obtained were also discussed. Finally, the chapter showed a complete model in the form of proposed procedures to minimize the effect of duration uncertainty from the two sample applications: robot-navigation and telescope-scheduling. In these procedures, the concepts introduced throughout this thesis and the techniques proposed were put together to show the applicability of these concepts and techniques.

Chapter 5 Conclusion

5.1 Discussion

The developing techniques that this thesis discussed will aid intelligent agents to reduce the effect of duration uncertainty of plans and schedules. This work started by studying the nature of events, event interactions, how they could exhibit duration uncertainty, and how the property of shared stages could be used to reduce duration uncertainty. Also explored were the techniques of identifying events with shared stages by looking at the relative duration of events when paired together. This information about relative durations was used to compute other measures such as relative average durations and their standard deviations. Standard deviations became a vital part in the σ-graph that could be used as a representation of the knowledge that intelligent agents collect about events. The σ-graph is an adequate structure for both retrieving existing knowledge about events and inferring new knowledge. An algorithm that makes use of the σ-graph

is the Hamiltonian Tour. Intelligent agents could apply the Hamiltonian Tour to identify effective orderings with minimal duration uncertainty. Two versions of the Hamiltonian Tour were discussed: one that gives an approximate measure of the shortest path, and another that generates an optimal shortest path. Providing these two versions makes it feasible to select the method appropriate to time and space constraints. These techniques were implemented by using Turbo Pascal. The thesis gave details on the issues involved and the problems encountered during the implementation stages. Samples of the test data and the results obtained from these implementations were also discussed. Finally, all the concepts and techniques discussed throughout this thesis were put together to implement procedures that can be used in robot-navigation and telescope-scheduling systems. These procedures demonstrated how intelligent agents could learn the dynamic changes in their environments.

5.2 Limitations and Extensions

By no means are the techniques proposed in this thesis exhaustive. Researchers in AI have being involved for years with temporal aspects of scheduling systems. Moreover, while the techniques presented throughout this thesis acted as bricks in building a complete understanding of the behavior of intelligent agents, the concern was always on enhancing these techniques. This section presents some of the limitations of the techniques being developed in this thesis and the possible enhancements to them.

- Sharing stages does not always yield a reduction in duration uncertainty. Consider, for example, two events E_1 and E_2 where E_1 consists of the stages A, B, and C; and E_2 consists of A, E, and F. If the duration uncertainty of E_2 is completely caused by stage F, then the technique proposed in this thesis would fail to recognize that the two events have shared stages, because E_2 will not demonstrate any reduction of duration uncertainty when paired with E_1. In this case, incorporating mean durations in the σ-graph along with the standard deviation would give better indication of events with shared stages. However, these attributes would imply a two-dimensional description space for the events, and shared stages would be based on a vector of attributes [10].

- In some situations, a certain ordering of the events might cause the total overall duration to be extended beyond its time frame. For example, if two events with shared stages are paired and this pairing causes a stage in one of the events to be extended by at least one hour, then, although the duration uncertainty might decrease, the overall duration would be outside the required total duration. The concept of *duration threshold* could be introduced to reject any ordering that is above the threshold [10].

- This study concentrated on studying the effect of pairing two events. Another perspective to explore would be to gather statistical information about triples, quadruples, etc. For example, more reduction in duration uncertainty might result from pairing C following A→B than by either A→C or A→B [10].

- The techniques discussed in this thesis concentrate on the temporal relation, *immediately follows*, as indication of an effective ordering of events. However, one might consider other relations such as, for example, *eventually follows, follows within a half hour of*. [11]. These types of relations might be closer to the real-world requirements of scheduling.

- There are other questions that come to mind regarding duration uncertainty. These concerns motivate further research and investigations. For example, selecting an ordering with minimal overall duration uncertainty might cause the overall duration of the events to violate the time-frame requirement. This problem means that any proposed technique should detect when such a problem occurs and to reject such ordering. Another concern is whether duration uncertainty always leads to failures in the completion of plans and schedules or whether there are merits to its existence. For example, each action that humans perform always carries some form of duration uncertainty, which means that each event that humans are scheduled to perform should account for some amount of duration uncertainty. Another point is that when the thesis discussed pairing events, it did not consider which of the two events follows the other. Here, the thinking was that the order of the two events is immaterial as long as they share stages. However, if one ordering is preferred over the other or if the environment returns to its original state after executing each event, then shared stages have no

effect on reducing duration uncertainty. These concerns and many more, we hope, are to motivate further researchers' investigations.

5.3 A Research with New Directions

In dynamically changing environments, temporal knowledge becomes outdated as well as being partial and incomplete. The approach of AI researchers was the integration of statistical information about past occurrences of events in order to predict how well a schedule will stand up against possible problems. The interest here is in systems embedded in a dynamic environment with feedback in the form of rewards [3]. An extension proposed in [12] deals with representing the knowledge in the dynamically changing world of intelligent agents. In this model a structure called the *duration network* gives intelligent agents feedback in the form of rewards. Each node in this network identifies a variable whose value represents the relative average duration of two events when they occur in consecutive temporal order. The edges between nodes are labeled by the reward of pairing the two events that the edge connects. These rewards represent the advantage of pairing the two events by virtue of sharing stages. Rewards are always ≤ 0 and constantly changing to reflect the change in the world. Figure 22 shows an example of a duration network. Here, the order $V_1 < V_2 < V_3$ (V_1 immediately before V_2 immediately before V_3) would yield a reward of 2+0=2 time units. This reward will result in an overall duration of 4-2+6+0+5=13 time units to perform this sequence. On the other hand, the

sequence $V_1 < V_3 < V_2$ would yield a reward of 1+0=1 and a total duration of 4-1+5+0+6=14 time units.

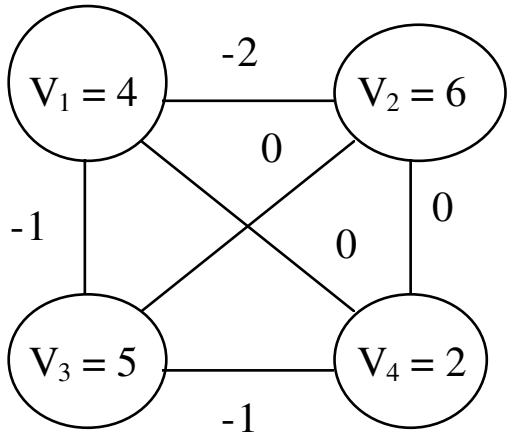

Figure 22 A duration network with four variables.

Intelligent agents could use the information in the duration network to select the tour with the shortest duration. After selecting such a tour, the agent updates the duration network to reflect the change in the mean relative durations. With this model, the agent uses experience to build its knowledge about the world. With the sequence of instantiations of values to the variables in a duration network, the agent becomes more and more confident of what tour of the network to follow as the number of instantiations increases. This growing knowledge will increase the agent's odds of selecting the tour with

the least amount of delay. With this model, the agent has the ability to change its goal by incorporating the appropriate method to respond to and update its world. Some of the goals that an intelligent agent might select include minimizing the overall extent of events, minimizing the overall duration uncertainty, minimizing the standard deviation of relative average durations which means selecting the tour which is least likely to encounter delays, and completing as many events as possible given rigorous time constraints. To summarize, this model says that although the world changes, it does so in ways that are often predictable, and intelligent agents can make use of this fact.

Bibliography

[1] Chapman, D. "Planning for Conjunctive Goals." *Artificial Intelligence* 32:333-377, 1987.

[2] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L, 1990. *Introduction to Algorithms*. The MIT Press.

[3] Dean, Thomas, Allen, James, and Aloimonos, Yiannis, 1995. *Artificial Intelligence: Theory and Practice.* The Benjamin/Cummings Publishing Company, Inc.

[4] Drummond, M., Bresina, J., Swanson, K. "Just-In-Case Scheduling." *Proceedings of AAAI-94*.

[5] Fox, M. S. "Constraint-Directed Search: A Case Study of Job-Shop Scheduling." Technical Report, Carnegie-Mellon University, 1983.

[6] Fox, M. S. "Observations on the Role of Constraints in Problem Solving." In *Annual Conference of the Canadian Society for Computational Studies*

of Intelligence. University of Quebec Press, 1986.

[7] Fox, Mark S. and Sadeh, Norman. "Why Is Scheduling Difficult? A CSP Perspective," *Proceedings of ECAI*, 1984.

[8] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

[9] Kruskal, J. B. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proc. Am. Math. Soc.* 1, 1956: 48-50.

[10] Morris, Robert A. "Automatic Generation of Efficient Orderings of Events for Scheduling Applications." *Proceedings of I-SAIRAS*, 1994: 443-447.

[11] Morris, Robert A., and Tamir, Dan E. "Teach Me What to Do Next: A Statistical Basis for Learning About Events." *Proceedings of TIME-95*.

[12] Morris, Robert A., Ford, Kenneth M., and Hayes, Patrick. "Time and Change in Reasoning about Order." *Proceedings of the 5th Toulouse International Workshop on Time, Space, and Movement*, 1995; forthcoming.

[13] Smith, Stephen F., Fox, Mark S., and Ow, Peng Si. "Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems," *AI Magazine*. Fall 1986: 45-61.

Appendix

PermutationsGenerator

```
{

This program creates all possible permutations of N values.  The program first prompts the user for
the value of N and the name of the output file.  It then calls procedure VISIT to generates the permuta-
tions.  For each permutation, it generate ten random numbers.  These numbers are needed for use with
other programs to simulate ten durations of events.

}

PROGRAM PermutationsGenerator;

CONST
  SET_ELEMENTS = 10;                    { The number of elements to generate
                                          for each event }
  MAX_SETS = 100;                       { The size of the VAL array used for
                                          generating the permutations }


VAR
  val : ARRAY [0..MAX_SETS] OF INTEGER; { The array used while generating the
                                          permutations }
  N,                                    { The number of numbers to permute }
  id : INTEGER;
  outfile : TEXT;                       { The handle of the output file }
  fname : STRING;                       { The name of the output file }
```

PermutationsGenerator

```
{
Visit: This procedure recursively calls itself to generate permutations for K values.  For each permuta-
tion generated, the procedure generates ten random numbers for use with other programs.

}

PROCEDURE visit(k : INTEGER);

VAR
  t, i, j : INTEGER;

BEGIN
  INC(id);
  val[k] := id;
  IF id = N THEN
  BEGIN
    FOR i := 1 TO N DO
    BEGIN
      WRITE(outfile, chr(val[i]+64));
      FOR j := 1 TO SET_ELEMENTS DO                    { Generate the ten random numbers }
        WRITE(outfile, (RANDOM(12)+1):3);
      WRITELN(outfile);
    END;
    WRITELN(outfile);
  END;
  FOR t := 1 TO N DO
    IF val[t] = 0 THEN
      visit(t);
  DEC(id);
  val[k] := 0;
END;

VAR
  i : INTEGER;

BEGIN
  RANDOMIZE;                                       { Initialize the randomization seed }
  id := -1;
  WRITELN;                                         { Prompt for the number of values to
                                                     permute }
  WRITE('Value for N : ');
  READLN(N);
  WRITE('Output file : ');                         { Prompt for the output file }
  READLN(fname);
```

```
  ASSIGN(outfile, fname);
  REWRITE(outfile);

  FOR i := 0 TO N DO                          { Initialize the permutations array }
    val[i] := 0;
  visit(0);                                   { Generate the permutations }
  CLOSE(outfile);                             { Close the output file }
END.
```

Figure 23 Pascal program to generate all permutations for a set of events.

Calculate

```
{
This program reads the permutation file PERM.OUT generated using PermutationsGenerator program
in Figure 23.  Then, it calculates the relative average durations and the standard deviations of each pair
of events in the event file.  The output will be saved to the output file CALC.OUT.

}

PROGRAM Calculate;

CONST
  NUM_SETS = 7;                                  { The number of events that were permuted }
  MAX_PER_SET = 10;                              { The number of durations for each event }

VAR
  list : ARRAY [1..NUM_SETS] OF RECORD           { Array to hold information about the
                                                   NUM_SETS events per set.  For each
                                                   event, it saves the name of the event,
                                                   the number of durations generated
                                                   for this event, and the actual values
                                                   of these durations }
           name : CHAR;                          { The name of the event }
           count : BYTE;                         { The number of durations }
           val  : ARRAY [1..MAX_PER_SET] OF BYTE; { Array to hold the values for the actual
                                                   durations }
         END;

  avg : ARRAY [1..1000] OF RECORD                { This array holds the calculated values
                                                   for all the permutations in the permutations file.  For each entry, it saves
                                                   the name of the two pairs of events,
                                                   the total extent of all durations, the
                                                   relative average durations, and the
                                                   standard deviations of the relative
                                                   average durations }
           first, last : CHAR;                   { The names of the pair of events }
           total,                                { The total of all durations }
           total2 : LONGINT;                     { The square of the total of all durations }
           count : INTEGER;                      { The number of occurrences of this
                                                   pair in the permutations file }
           mean,                                 { The relative average durations of all
                                                   durations of this pair }
```

Calculate

```
      std : REAL;                                   { The standard deviation of the relative
                                                      average durations }
    END;

 i, j, k, count, num : INTEGER;
 f : TEXT;
 found : BOOLEAN;

BEGIN
 ASSIGN(f, 'perm.out');                             { Open the permutations file }
 RESET(f);

 count := 1;
 WHILE NOT EOF(f) DO
 BEGIN
  FOR i := 1 TO NUM_SETS DO                         { Read one set of events from the per-
                                                      mutations file }
  BEGIN
   READ(f, list[i].name);                           { Read the event name }
   list[i].count := 0;
   WHILE (list[i].count < MAX_PER_SET) AND          { Read the durations of this event }
         NOT EOLN(f) DO
   BEGIN
    INC(list[i].count);
    READ(f, list[i].val[j]);
   END;
   READLN(f);
  END;
  READLN(f);

  j := 1;                                           { Do the required calculations on the
                                                      set of events just read }
  REPEAT
   IF j+1 <= NUM_SETS THEN
   BEGIN
    found := FALSE;
    FOR k := 1 TO count-1 DO
     IF ((avg[k].first = list[j].name) AND          { Check if this pair has been found in
                                                      the permutations file before }
         (avg[k].last = list[j+1].name)) OR
        ((avg[k].first = list[j+1].name) AND
         (avg[k].last = list[j].name)) THEN
     BEGIN                                          { If yes, then don't add this pair to the
                                                      list of pairs }
```

Calculate

```
        found := TRUE;
        num := k;
        BREAK;
      END;
    IF NOT found THEN                          { Otherwise, create a new entry for this
                                                 pair }
    BEGIN
      avg[count].first  := list[j].name;
      avg[count].last   := list[j+1].name;
      avg[count].total  := 0;
      avg[count].total2 := 0;
      avg[count].count  := 0;
      num := count;
      INC(count);
    END;
    FOR k := 1 TO list[j+1].count DO           { Update the total, the square of the
                                                 total, and the number of durations }
    BEGIN
      INC(avg[num].total, list[j+1].val[k]);
      INC(avg[num].total2, sqr(list[j+1].val[k]));
      INC(avg[num].count);
    END;
  END;
  INC(j);
 UNTIL j = NUM_SETS;
END;
CLOSE(f);
                                               { Now, calculate the relative average
                                                 durations and the standard deviation
                                                 of all pairs of events using the calcu-
                                                 lated data stored in the AVG array }
FOR i := 1 TO count-1 DO
BEGIN
 avg[i].mean := avg[i].total / avg[i].count;   { Calculate the relative average dura-
                                                 tion }
 avg[i].std  := sqrt(avg[i].total2/avg[i].count -
                sqr(avg[i].mean));             { Calculate the standard deviation }
END;

ASSIGN(f, 'calc.out');                         { Save the calculated results in the
                                                 CALC.OUT output file }
REWRITE(f);
FOR i := 1 TO count-1 DO
 WRITELN(f, avg[i].first, ' ', avg[i].last, avg[i].std:9:3);
```

```
  CLOSE(f);
END.
```

Figure 24 Pascal program to read the raw data of the permutations file generated by the program of Figure 23 and calculate the relative duration, the relative average duration, and the standard deviation of each pair of events.

LoadGraph

```
{
 This procedure loads the data file found in CALC.OUT that was generated by the
CALC.PAS program in Figure 24.  Each line of this file contains an edge referenced by
the pair of nodes that it connects, and the standard deviation of the relative average dura-
tions of the pair of events.  The standard deviation will be used as the weight on the edges
for the graph that this procedure creates.  The procedure first builds the graph and then
extracts an adjacency matrix.  This matrix is used in following the shortest path from each
node to every other node.
}

PROCEDURE LoadGraph;

VAR
  f : TEXT;
  num_nodes,                                    { Number of nodes in the graph }
  num_edges,                                    { Number of edges in the graph }
  i, j : INTEGER;
  c : CHAR;
  node : ARRAY [1..MAX_EDGES] OF                { Array to hold information about all
                                                  the edges in the graph }
       RECORD
         first, last : CHAR;
       END;
  list : ARRAY [1..MAX_EDGES] OF REAL;          { This array holds the weights on all
                                                  the edges of the graph }
  found : BOOLEAN;

BEGIN
  ASSIGN(f, 'calc.out');                        { Open the input file }
  RESET(f);

  num_edges := 0;                               { Initialize the number of nodes and
                                                  the number of edges in the graph to 0
                                                  }
  num_nodes := 0;

  WHILE NOT EOF(f) DO
  BEGIN
    INC(num_edges);                             { Increment the number of edges in
                                                  the graph }
    READLN(f, node[num_edges].first, c,         { Read a line from the input file }
           node[num_edges].last, list[num_edges]);

                                                Check if the node that this edge
```

LoadGraph

```
                                                    emerges from already exists }
  found := FALSE;
  FOR j := 1 TO num_nodes DO
    IF nodes[j] = node[num_edges].first THEN        { Node exists }
    BEGIN
      found := TRUE;
      BREAK;
    END;
  IF NOT found THEN                                 { Source node does not exist, so cre-
                                                    ate it }
  BEGIN
    INC(num_nodes);
    nodes[num_nodes] := node[num_edges].first;
  END;

                                                    { Check if the node that this edge
                                                    goes to already exists }
  found := FALSE;
  FOR j := 1 TO num_nodes DO
    IF nodes[j] = node[num_edges].last THEN         { Node exists }
    BEGIN
      found := TRUE;
      BREAK;
    END;
  IF NOT found THEN                                 { Destination node does not exist, so
                                                    create it }
  BEGIN
    INC(num_nodes);
    nodes[num_nodes] := node[num_edges].last;
  END;
END;
CLOSE(f);

g.num_vertices := num_nodes;                        { Set the number of vertices in the
                                                    graph }
FOR i := 1 TO num_nodes DO                          { Number the nodes of the graph se-
                                                    quentially }
  g.v[i] := i;

g.num_edges := num_edges;                           { Set the number of edges in the
                                                    graph }
FOR i := 1 TO num_edges DO                          { Update the source nodes of all edg-
                                                    es in the graph }
BEGIN
```

LoadGraph

```
  FOR j := 1 TO num_nodes DO
    IF nodes[j] = node[i].first THEN
    BEGIN
      g.e[i].source := j;
      BREAK;
    END;

  FOR j := 1 TO num_nodes DO                    { Update the destination nodes of all
                                                  edges in the graph }
    IF nodes[j] = node[i].last THEN
    BEGIN
      g.e[i].dest := j;
      BREAK;
    END;
END;

FOR i := 1 TO num_nodes DO                      { Create the Adjacency matrix }
BEGIN
  Adj[i].size := 0;
  FOR j := 1 TO num_edges DO
    IF g.e[j].source = i THEN
    BEGIN
      INC(Adj[i].size);
      Adj[i].elements[Adj[i].size] := g.e[j].dest;
    END;
END;
```

```
FOR i := 1 TO num_edges DO                      { Update the weights of the edges of
                                                  the graph }
  w[g.e[i].source, g.e[i].dest] := list[i];
END;
```

Figure 25 Pascal procedure that loads the file of calculated data generated by the program of Figure 24.

NearestNeighbor

```
{
This procedure finds an approximate shortest path of a complete graph from a source
node.  The procedure uses the σ-graph built by procedure LoadGraph in Figure 25.
The procedure starts from the source node and then follows all other nodes taking the
edge with the minimum weight from each node as the next node to visit.
}
CONST
  MAX_ELEMENTS = 100;
  MAX_VERTICES = 10;                          { Max number of nodes in the graph }
  MAX_EDGES = 100;                            { Max number of edges in the graph }
  MAX_MATRIX_ELEMENTS = 10;                   { Size of the adjacency matrix }

TYPE
  integers_list = ARRAY [1..MAX_ELEMENTS] OF
INTEGER;
  elements_list = ARRAY [1..MAX_ELEMENTS] OF REAL;

  queue_type = RECORD
    size : INTEGER;
    elements : integers_list;
  END;

  weights_matrix = ARRAY [1..MAX_MATRIX_ELEMENTS,
                          1..MAX_MATRIX_ELEMENTS] OF
REAL;

  edge_type = RECORD                          { Each edge is designated by the source
                                                and destination nodes }
    source,
    dest   : INTEGER;
  END;
  edges_list = ARRAY [1..MAX_EDGES] OF edge_type;   { A list to hold the edge of the graph }
  graph_type = RECORD                         { A description of a graph structure }
    num_vertices, num_edges : INTEGER;
    v : integers_list;
    e : edges_list;
  END;

  adjacency_list = ARRAY [1..MAX_VERTICES] OF     { Structure for the adjacency matrix }
                   queue_type;

  visited_array = ARRAY [1..MAX_VERTICES] OF
                  BOOLEAN;
```

NearestNeighbor

```
VAR
  nodes : ARRAY [1..MAX_VERTICES] OF CHAR;          { A list of nodes in the array }
  visited : visited_array;
  num_vertices : INTEGER;
  d : elements_list;
  next : integers_list;
  g : graph_type;
  w : weights_matrix;
  Adj : adjacency_list;

PROCEDURE NearestNeighbor(G : graph_type;          { The complete graph }
                          w : weights_matrix;      { The weights on all the edges }
                          s : INTEGER);            { The source node }

VAR
  i, this, closest_vertex : INTEGER;
  minimal_weight : REAL;
  no_more : BOOLEAN;

BEGIN
  FOR i := 1 TO g.num_vertices DO                  { All nodes have not yet been visited }
    visited[i] := FALSE;
  visited[s] := TRUE;                              { Start with the source node }

  this := s;

  REPEAT
    minimal_weight := MAXINT;                      { Find an outgoing edge with the mini-
                                                     mum weight }
    no_more := TRUE;
    FOR i := 1 TO g.num_vertices DO
      IF NOT visited[i] AND (w[this,i] < minimal_weight)   { This edge is the shortest so far }
      THEN
      BEGIN
        closest_vertex := i;
        minimal_weight := w[this,i];
        no_more := FALSE;
      END;
    IF NOT no_more THEN                            { If there are still other nodes not visit-
                                                     ed }
    BEGIN
      visited[closest_vertex] := TRUE;             { Set the node at the end of the closest
                                                     edge to be visited }
      next[this] := closest_vertex;                { Set the next node in the path from the
```

```
                                                    current node }
    d[this] := w[this, closest_vertex];             { Update the weights array }
    this := closest_vertex;                         { Visit the next node }
   END;
  UNTIL no_more;
  next[this] := 0;                                  { This is the last node in the path }
END;
```

Figure 26 Pascal procedure that implements the Nearest Neighbor algorithm.

OptimalPath

```
{
- This procedure recursively calls itself to find the optimal shortest path from a source
node that visits all other nodes in a complete graph only once. The procedure visits eve-
ry possible path from the source node. For each path, it compares the cost (the total
weights on all edges) of the path. When it finds a path that is shorter than the shortest
path found so far, it thent replaces the shortest path with the path being found.
}

CONST
  MAX_ELEMENTS = 100;
  MAX_VERTICES = 10;                          { Max number of nodes in the graph }
  MAX_EDGES = 100;                            { Max number of edges in the graph }
  MAX_MATRIX_ELEMENTS = 10;                   { Size of the adjacency matrix }

TYPE
  integers_list = ARRAY [1..MAX_ELEMENTS] OF
INTEGER;
  elements_list = ARRAY [1..MAX_ELEMENTS] OF REAL;

  queue_type = RECORD
    size : INTEGER;
    elements : integers_list;
  END;

  weights_matrix = ARRAY [1..MAX_MATRIX_ELEMENTS,
                          1..MAX_MATRIX_ELEMENTS] OF
REAL;

  edge_type = RECORD                          { Each edge is designated by the
                                                source and destination nodes }
    source,
    dest   : INTEGER;
  END;
  edges_list = ARRAY [1..MAX_EDGES] OF edge_type;   { A list to hold the edge of the graph }
  graph_type = RECORD                         { A description of a graph structure }
    num_vertices, num_edges : INTEGER;
    v : integers_list;
    e : edges_list;
  END;

  adjacency_list = ARRAY [1..MAX_VERTICES] OF       { Structure for the adjacency matrix }
queue_type;

  visited_array = ARRAY [1..MAX_VERTICES] OF
```

OptimalPath

```
BOOLEAN;

VAR
  nodes : ARRAY [1..MAX_VERTICES] OF CHAR;        { A list of nodes in the array }
  visited : visited_array;
  num_vertices : INTEGER;
  g : graph_type;
  w : weights_matrix;
  Adj : adjacency_list;

PROCEDURE OptimalPath(path : integers_list;       { A list of nodes holding the nodes that
                                                    have been seen so far }
                num : INTEGER;                    { Index for the array PATH }
                sum : REAL;                       { The accumulated weight }
                visited : visited_array;          { An array to tell the nodes that have
                                                    being visited so far }
                i : INTEGER;                      { The current node }
                VAR minimum : REAL;               { The minimum weight obtained so far
                                                    }
                VAR shortest_path : integers_list); { An array that holds the nodes for the
                                                    shortest path obtained so far}

VAR
  j : INTEGER;
  temp : REAL;

BEGIN
  visited[i] := TRUE;                             { Set the current node to being visited
                                                    }
  FOR j := 1 TO g.num_vertices DO                 { Visit all other nodes from the current
                                                    node }
    IF NOT visited[j] THEN
    BEGIN
      sum := sum + w[path[num], j];               { Accumulate the weight }
      temp := w[path[num], j];
      INC(num);
      path[num] := j;                             { Add this node to the current path }
      OptimalPath(path, num, sum, visited, j, minimum,
                  shortest_path);                 { Call OptimalPath taking the current
                                                    visited node as the source node }
      visited[j] := FALSE;                        { Reset the status of the current node
                                                    so to be included in other paths }

      DEC(num);
      sum := sum - temp;
    END;
```

OptimalPath

```
  IF num = g.num_vertices THEN                    { Have we visited all nodes? }
  BEGIN
    IF sum < minimum THEN                         { If the current path is shorter, then
                                                    update the shortest path }
    BEGIN
      minimum := sum;
      shortest_path := path;
    END;
  END;
END;
```

Figure 27 An implementation of the optimal shortest path algorithm.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W