SCHEDULING WITH TEMPORAL AND LOGICAL CONSTRAINTS

BY

GREGORY B. JONES

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
APPLIED MATHEMATICAL SCIENCES

UNIVERSITY OF RHODE ISLAND
1998

Abstract

The scheduling of tasks in a real-time system requires that, in addition to existing logical constraints, temporal constraints must also be met. We consider real-time systems where timing constraints are imposed on tasks through a function of multiple deadlines with cumulative penalties for missing deadlines. Logical constraints are enforced through the partial ordering of tasks, and penalties are assessed for violating these precedence constraints. We propose a bi-criteria scheduling approach in which tasks are scheduled to meet both logical and temporal constraints without preference to either. When both logical and temporal constraints cannot all be met simultaneously, a mechanism for trading off performance in one measure for performance gains in the other is developed. We use a frontier of possible solutions to express the relative merits of schedules that perform well in each performance measure. Branch and bound algorithms plus heuristic methods are proposed to solve several different classes of related problems.

Acknowledgments

In my progress as a student, I quickly came to realize that the work presented here was not due to my efforts alone. Without the help, guidance and encouragement of many others, I never would have reached this plateau.

First, I would like to thank my two major professors, Dr. Manbir S. Sodhi and Dr. Victor Fay-Wolfe. I am indebted to them for the many hours they devoted to helping me with this research. The technical issues and concepts presented here were a result of their efforts and collaboration. More than this, they joined me in this journey as faculty advisors, and in the end became my friends.

I also would like to express my appreciation to my employer, the Naval Undersea Warfare Center of Newport RI (NUWC). I never would have attempted such an undertaking without their support. At NUWC, I would especially like to thank my immediate supervisor, Dr. James Meng, for his support and encouragement. In addition, I thank Dr. Richard Nadolink and Mr. Frederic White for their support as well.

A number of faculty members at URI encouraged and assisted me in pursuing this degree, I am especially thankful to, Dr. Agnes Doody, Dr. Richard Scholl, Dr. Russ Koza, Dr. Alan Humphrey, Dr. Joan Peckham and Dr. Lisa DiPippo.

In these last several years I have asked much from my family–and they have always responded with encouragement and understanding. My wife, especially, provided loving encouragement and support. To my children Stephanie and Kevin, I've been a student longer than they can remember. Watching them grow, I learned just how exciting learning can be.

Finally, I want to acknowledge my Mom and Dad. Throughout my life, they've been a constant source of unquestioning love and support, and more importantly, they believed in me at times when few others did.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Scheduling tasks is an activity that is characterized in many applications as being $\mathcal{NP}$-hard. Nevertheless, real-time systems still require tasks be scheduled to achieve optimal or near-optimal performance.

The focus of this research is the relationship between task scheduling and precedence constraints in applications where temporal and logical consistency constraints can be quantified. We have developed techniques and algorithms that maximize the utility of a system through maximizing both temporal and logical consistency. Techniques of task scheduling are relevant to database transactions, the scheduling of jobs in a manufacturing system, and other scheduling applications that operate under similar constraints.

1.1 Motivation

There are a number of applications that can utilize the work described here. We will draw primarily from three applications which will be used for motivation and example.

1.1.1 Real-Time Databases

Database scheduling and concurrency control seeks to produce schedules that meet some correctness or goodness criteria. A correctness criteria is a requirement that must be met, a goodness criteria is a goal. The transaction manager in a real-time database attempts to schedule transaction tasks such that both these constraints are met. However, it is often the case that both concurrency requirements and timing requirements cannot be met. The goals of scheduling algorithms that seek to maintain *temporal consistency* requirements such as timing constraints, or synchronization, can have fundamental conflicts with the goals of concurrency control techniques, which seek to maintain *logical consistency* requirements, like serializability. Typical database management systems have separate scheduling algorithms and concurrency control, and if temporal consistency is met at all, if is only after the requirements of logical consistency are met.

Consider an aircraft tracking and control system. A database is maintained of track and attribute information related to air traffic in the sector being monitored. Data from radar sensors is used to update aircraft information. Tracking algorithms process the raw data to associate the new sensor information with existing aircraft tracks, and thus update that data. Display consoles are updated with new track data to show the relative positions of aircraft in the sector. In this example, a real-time database is used to store the relevant information. The processing of tracking algorithms must be performed with system defined deadlines in order to provide timely information to air traffic controllers. Deadlines are based on several factors, including the speed of aircraft and their proximity to other aircraft. Up to date information is crucial to air traffic controllers in order to move aircraft efficiently and safely through the sector. In periods of overload, it may be necessary to reduce the time required to process tracking data so that a reasonably up to date display can be maintained.

Typically, database-scheduling algorithms enforce some sort of concurrency control that preserves logical consistency, often by ensuring serializable schedules. In addition, the schedule may also attempt to maximize average throughput of transactions as a temporal goodness criterion. However, in a real-time database [Ram93], ensuring temporal consistency by meeting timing constraints is a correctness criteria. As we have seen in the previous example, transactions to determine the positions of aircraft may have to be performed by a deadline to be correct. In addition to transactions, *data* in such a real-time database may also be time constrained. For instance, the database's recorded position of a tracked aircraft may be constrained such that data more than several seconds old is no longer valid, and cannot be used.

Transaction scheduling in most real-time databases seeks to maintain temporal consistency by meeting as many transaction deadlines as possible, while scheduling tasks as required by the concurrency control mechanism. These concurrency control techniques enforce logical consistency, but are not cognizant of temporal consistency. Even proposed real-time database concurrency control techniques have only limited support for temporal consistency[YWLS94].

Although an individual scheduling technique can support temporal consistency and an individual concurrency control technique can enforce logical consistency, when the techniques are combined, often one form of constraint must be sacrificed for the other. In typical databases, logical consistency is enforced absolutely, while temporal consistency is secondary or ignored. This policy occurs in nearly all applications, as logical constraints are nearly always considered inviolate. Consequently, temporal goals are usually the secondary goal after temporal goals. A unified approach to scheduling and concurrency control, where both constraints are considered together, is rarely attempted.

The synthesis of database scheduling and concurrency control is complicated by the fact that, although they both seek to establish schedules of operations, their goals are usually not the same. In fact, there can be a fundamental conflict between a concurrency control technique's goal of maintaining logical consistency, and a scheduling algorithm's goal of maintaining temporal consistency. That is, maximizing performance for logical constraints may result in the deterioration of temporal performance.

1.1.2 Scheduling Jobs for a Flexible Manufacturing System

A second application for this work relates to manufacturing systems, and specifically for an FMS (Flexible Manufacturing System). An FMS has the capability to process a number of different parts with minimal setup time in between parts of different types. An FMS provides the automation advantages of an assembly line with the flexibility of a job shop.

An FMS refers to a set, or cell, of automated machines that are serviced by an automated material handling system. The FMS is typically controlled by a processor. Each machine usually holds a magazine of several tools. In this manner material delivery, as well as machine setup, is automated. Depending on the tools loaded into the machine, several machines can perform the same operation, providing a choice of which machine to route each part to. With the flexibility to route parts based on machine loading as well as machining requirements, the issue of scheduling tasks becomes more complicated.

Parts in manufacturing are usually processed in a predefined order of operations. Partial orderings may be required or may be used for convenience, indeed in a flow shop or assembly line the ordering of operations is not easily changed. However, it may be technically possible to perform operations out of order with or without some degradation of quality. When out-of-order processing is possible, a penalty function can be used to specify the feasibility of performing specific operations out of order. The flexibility of the FMS motivates the notion of out of order processing. Performing operations in an alternate order may allow deadlines to be met, or production to be increased.

Consider a part that requires a set of machining operations. Under periods of overload, it becomes difficult, if not impossible, to meet deadlines. By performing some operations out-of-order, that is, violating precedence constraints, it may be possible to process parts more quickly. Thus, in an FMS system, penalties for out-of-order processing may be acceptable when considered conjunction with deadlines met or increased production. With the availability of an FMS, it now becomes possible to make this trade-off, but it also becomes a challenge to schedule work for the FMS. Without proper production planning, the advantages that FMS can provide are lost.

1.1.3 Mission Planning for Undersea Vehicles

Unmanned Undersea Vehicles (UUV), commonly used for military exercises provides yet another application for this work. The UUV is an self-propelled submersible vehicle that, once launched, must meet pre-defined mission goals without further human intervention. This requires on-board processing sufficient to plan and re-plan to meet mission goals. Such systems are attractive since they greatly increase the range and capability of a ship or submarine at lower cost and reduced risk. UUV's have a number of possible missions, including payload delivery, search, surveillance and others. Often, such vehicle may have multiple mission goals and subgoals. However, UUV's have limited operational duration due to limited propulsion power. In some missions, a deadline may also be imposed on one or more goals. During the course of the mission, both external and internal situations may develop which change the goals or deadlines. Given this scenario, mission planning becomes a scheduling problem, similar to the scheduling problem for a real-time database or FMS. The need for a UUV to meet goals within a certain time frame must be balanced against the need to perform all mission tasks. In the more difficult case, not all goals can be met within the time allotted, and either some goals must be sacrificed, or deadlines missed. The ability to solve this problem such as to maximize the UUV's utility is a measure of the *intelligence* built into the vehicle.

1.1.4 Other Applications

There are numerous other applications from fields such as combat control systems, finance, multimedia, target tracking, and automobile traffic monitoring in intelligent highway systems, where the goals of temporal consistency are in conflict with logical consistency goals. While it is usually the case in these applications that tasks are scheduled to meet temporal requirements only after logical consistency is achieved, it is feasible to consider both logical and temporal consistency simultaneously. When conflicts between these requirements arise, the scheduling system must be capable of addressing both sets of requirements. This project develops a unified approach to scheduling and concurrency control that can trade-off logical and temporal consistency requirements.

The techniques that will be investigated through this research will be relevant to any application where there is a necessity to trade-off logical and temporal constraints. However, this work is not applicable to those applications where the violation of temporal constraints (or, alternatively, logical constraints) constitutes a failure of the system. Also this research is not applicable in situations where it *is* feasible to permit a trade-off between logical and temporal constraints, but the trade-off cannot be explicitly specified.

One common thread that ties all these applications together is the requirement of *strong* timing constraints, that is, failure to meet a deadline causes a measurable reduction in the quality of service provided by the system. It is with this understanding that we contemplate the violation of logical constraints.

In the scheduling of tasks there are several assumptions that we make. First, we assume that there exist several measures of performance against which success is determined. Second, by changing the scheduling order, the system performance will change in these measures. Third, penalties due to the incorrect ordering of tasks are related to a metric space. Fourth, and perhaps most importantly, performance in one measure can be traded off for better performance in another, and the value of the trade-off is understood.

The problem described by this research is the natural extension of previous research in the field. At The University of Rhode Island (URI), the problems of both scheduling and real-time databases have been extensively been studied [PDPW94]. However, up to this time, the concept of a trade-off between logical and temporal consistency has not been fully exploited. Previous research has built a solid foundation for the need to relax logical constraints in light of temporal requirements.

1.2 Goal of Research

The goal of this project is to produce results that lead to the unification of transaction scheduling and concurrency control in real-time systems such as real-time databases and FMS's. These results will lead to a better general understanding of the relationship between these two crucial scheduling issues in a real-time system.

Through the results of this work, we have developed scheduling techniques relevant to a wide range of applications. Using examples, we suggest how these techniques can be applied to several applications. More specifically, we seek to apply the notion of penalty function to describe temporal requirements and generalize logical constraints using precedence requirements. We then seek to establish a model for the characterization of real-time tasks in applications where a trade-off between temporal and logical constraints is possible. We seek to develop a theoretical understanding of this trade-off and show that these two performance measures can be controlled through the scheduling of tasks. Finally, we seek to develop algorithms and heuristics to maximize both logical and temporal requirements and validate these methods through simulation.

1.3 Approach

The problem of scheduling tasks has been widely studied in both the fields of Operations Research and Computer Science. Scheduling can be a difficult problem, and indeed, such problems as the Traveling Salesman Problem (TSP) [LLKS85] are known to be $\mathcal{NP}$-hard. The TSP cannot be optimally solved in polynomial time. If an enumerative technique is used to find a solution, then for a problem of size n, the *search space* includes $n!$ possible sequences which must be considered. On the other hand, scheduling problems that require a partial ordering of tasks are usually considered easier, since the imposition of additional constraints reduces the search space, that is, a number of sequences are not viable. A set of tasks is *partially ordered* if it is required that some tasks be performed prior to some other tasks, for example: $a \prec b$. We will formalize the notion of partial ordering in §2.3.3. In the problems that we are considering, however, we *relax* the constraint imposed by partial ordering. Thus, rather than reduce the size of the problem, it actually is larger. The issue of search space size is important in this work and will be discussed at length in Chapter 6.

We used an incremental approach to this problem. We first start out with a simplified problem, but one that embodies the concepts of trading off temporal for logical requirements. This first problem models the total processing time or *makespan*, with precedence ordering between tasks. The model requires the deletion of tasks, with penalty, if precedence violations occur. The reduction of makespan for violations of precedence ordering forms the basis of a trade-off. The solution to the simple problem leads to results that are useful in several applications. More importantly, the solution to this problem validates the central concept of our work and we will continue with this concept to solve problems that are more difficult.

The model of the second problem extends the results of the first model to include transactions. In this model, transactions have individual deadlines. Transactions are modeled to include several subtasks and a final, or commit, task. Precedence constraints relate all subtasks to a single commit task. As in the first model, tasks that violate precedence constraints are deleted with penalty. While this model is still rather limited, it does allow individual deadlines for transactions. The solution to this problem is provided by a heuristic, which does not necessarily result in an optimal solution. Simulation is used to assess the effectiveness of the heuristic.

The third and final problem is the most general. This problem is modeled such that all tasks have timing constraints, expressed through a series of deadlines each with an increasing penalty if missed. Partial ordering between tasks is permitted, with penalties for precedence violations. A number of solutions for this model, both exact and non-exact are explored.

The sequence of models considered begins with the simple, restricted model and leads, in subsequent models, to the general model. The restrictions assumed in the first model are incrementally removed to provide a more complex problem. In all cases, we first seek formal definition of the problem. Problems are formulated as Integer Program (IP) problems, which leads to possibility of their being solved optimally with commercial software packages. This, however, is not suitable for most applications because these packages do not take advantage of the unique structure of the problem leading to processing times that are much longer than can be obtained by other means.

We develop heuristics to obtain solutions the problems we describe. Such heuristics have the advantage of being able to solve the problem quite quickly, although in general, the solution is not optimal. Exact solutions are possible with enumerative techniques such as branch and bound. Branch and bound can be tailored to the application by using the unique structure of the problem. However problem size is still limited and performance (time to solution) can not be guaranteed. Both exact and heuristic solutions have advantages and and limitations; these considerations are discussed in depth.

1.4 Dissertation Outline

The chapters that follow in this dissertation are organized as follows. We begin in Chapter 2 with a discussion of literature related to this work and how we will expand on this literature. In Chapter 3 we provide a detailed statement of the problem and provide additional motivation. In Chapter 4 we discuss in detail the first simplified problem, a problem which illustrates the concepts and further defines the nature of the more general problem. Chapter 5 defines a more difficult problem and its solution. In Chapter 6 we discuss the problem of scheduling with penalty functions. In Chapter 7, we then include precedence constraints. This is the most general problem and we provide several solution strategies. In Chapter 8 we present a summary and conclusion of our work and in Chapter 9 we suggest additional work that might be performed to extend this work.

Chapter 2

Related Work

In this chapter, relevant work from closely related fields is reviewed. We consider previous work performed at the University of Rhode Island and the application of those results to our work. We also examine work in the fields Computer Science and Operations Research from which we draw from for work in real-time computing, concurrency control and scheduling.

Since our proposed project involves synthesizing database scheduling and concurrency control by handling both logical and temporal consistency, we focus on work in those areas. In particular, we examine work on real-time task scheduling with value or penalty functions, and on concurrency control techniques that allow some logical imprecision by relaxing serializability.

In the next section, we review work from the University of Rhode Island RTSORAC real-time database system. Specifically, we extend the RTSORAC project results in scheduling transactions to meet deadlines while allowing some imprecision. We also review work done in scheduling and real-time computing, including value based scheduling and multi-criteria scheduling. From these areas, we will develop the basic concepts for task scheduling which supports a trade-off with logical constraints. The second major issue related to this work is the enforcement of logical consistency.

We review work in concurrency control which leads to the more specific topics of serializability, non-serializable concurrency control, epsilon serializability, similarity, precedence ordering and work related to real-time concurrency control.

The work reviewed here builds a solid foundation for the work that follows. Using concepts from scheduling and concurrency control, we have been able to develop the theory and practice of scheduling to meet timing and logical constraints.

2.1 RTSORAC

In research conducted at URI, a semantic concurrency control technique has been developed for RTSORAC (Real-Time Semantic Objects and Constraints), a real-time object-oriented database model and system [PDPW94]. This technique allows the database designer to relax logical consistency for each object such that greater temporal consistency might be achieved. The amount of logical imprecision introduced by this technique can be bounded under certain general conditions. This result leads to the conclusion that logical consistency can be relaxed with the consequence of greater imprecision. Implicit in this concept is that a schedule could take advantage of relaxed logical consistency to provide better temporal performance. However, without a scheduler that seeks to exploit this idea, enhanced temporal performance is not assured. What is required is to explicitly define this trade-off through the implementation of a scheduler that maximizes both temporal performance and logical consistency. In our goal to develop a unified scheduling and concurrency control technique, we seek to exploit the trade-off between temporal and logical consistency. The RTSORAC system is strongly object-oriented where objects have both data and methods associated with them. The goal is to schedule methods in such a way as to maintain an object's logical consistency, but to also meet deadlines on method completion. By relaxing the constraints imposed by logical consistency, more schedules are logically feasible and thus, the chance that a schedule exists that will meet both logical and temporal requirements is increased. Logical consistency is enforced in a number of ways, including locks, semantic locking, similarity and other techniques [BHG86]. RTSORAC relaxes logical constraints with a semantic method that uses a *compatibility function*. Method invocations may interleave, and may even access the same data object. The compatibility function is a table that defines the possibility of conflict between methods depending on the type of object access being considered. Using less restrictive forms of concurrency control allows more schedules to be considered. However, RTSORAC also permits an additional relaxation of logical constraints resulting in imprecise results. Normally, a concurrency violation is the illegal access of shared data objects, which may result in erroneous data being stored or output from the system. In the work of DiPippo [DiP95], potential concurrency violations are mapped to an increase in precision. Based on the value of the data being accessed and the type of access planned, the potential error in the data can be determined. This potential error is then a bound on the imprecision of the data. An upper bound on the imprecision is set as a matter of system policy, prior to a transaction that might increase imprecision, the system limit is checked. If the transaction does not exceed the bound, it is permitted. This system then provides a means of trading off imprecision with the ability to meet deadlines. This work will form the basis for much of our work here. The notion of violating concurrency controls to permit better temporal performance is core to our work.

2.2 Real-Time Computing

Real-time tasks are tasks that have an associated timing constraint as part of the task definition. For a system to be considered real-time, the system must provide a mechanism to handle timing constraints in a predictable way. This more precise definition is commonly used in the field [Sta88], and differs from the notion of "fast" processing. Faster processing certainly improves processor throughput and can help to meet required timing constraints. However, in our work here, we are primarily interested in the case where the system, for whatever reason, cannot meet the required timing constraints. Timing constraints imposed on a task may include start times, end times, duration or synchronization with other tasks. Frequently, timing constraints are expressed as deadlines, which in turn, can be categorized as hard, soft or firm. While some authors suggest that failure to meet a hard deadline leads to catastrophic results, or unrecoverable failure, it suffices to say that failure to meet a hard deadline is a failure of the system [Jen96].

Hard real-time requires predictable execution and *a priori* guarantees, which places hard real-time outside the scope of this work, since hard deadlines preclude the possibility of missing deadlines in order to meet logical consistency requirements. With a soft timing constraint, there may still be some benefit in completing the task outside of the timing constraints-although this value likely decreases with time. Meeting a firm timing constraint is not a necessary condition for correctness, but there is no value in executing the task outside of its firm timing constraints. In this dissertation, we are primarily interested in soft and firm real-time.

An important implication of real-time computer processing is that the scheduling process must often execute concurrently with the tasks to be performed, that is, the overhead of the scheduling algorithm must be considered. This is especially true for soft and firm deadlines where, unlike hard deadlines, dynamic scheduling is an option. The overhead required for scheduling is of more or less importance depending on the domain. In a real-time database, for example, the time required to schedule transactions is time that could have been used to actually perform the transactions. In the worst case, transactions performed in some arbitrary order meet more deadlines than transactions in an order determined by the scheduler.

Scheduling systems for processor tasks must be efficient to be useful, that is they must impose low overhead on the system. Some systems provide a separate processor for scheduling, however, this barely sidesteps the question of whether it would be more efficient to apply the additional computing power to processing transactions.

Considering the processing time for other applications, such as in an FMS (flexible manufacturing cell), a computer-based scheduler would normally execute much faster than the actual manufacturing operations. Thus, the use of a more sophisticated scheduler can be justified by the improvement in overall temporal performance without too much concern about the overhead of the scheduling processor. The advances in computer technology suggest that faster computers will be available in the future, however, machining operations are less likely to show a similar improvement in processing time. This allows the use of increasingly sophisticated scheduling algorithms.

We are not particularly concerned with scheduler processing time in this work, although we do address the issue for algorithms developed. We are looking to find feasible ways to schedule tasks, implementing the methods proposed in an efficient scheduler is work left for future investigations. Algorithm complexity has important implications for, since this determines the required processing time as task set size increases. Where the processing time to create a schedule is an issue, it may be possible to construct various schedules in advance for "typical" or expected cases. These schedules, or schedule templates, can then be used when needed while greatly reducing the scheduling overhead.

2.2.1 Scheduling

Scheduling is the process of ordering tasks for execution. In spite of all the complexities involved with scheduling, the result can be summarized in the simple question:

"what task goes next?" The output of the scheduler is simply a list of tasks with start times. If the scheduler has introduced no idle time between tasks, then the scheduler output is simply a list of tasks in the order they are to be executed.

Scheduling algorithms usually create an ordered list of tasks by allocating shared resources to maintain some correctness or goodness criteria. Typically, a scheduling algorithm assigns priorities to tasks. These priority assignments establish a partial ordering among tasks. Whenever a scheduling decision is made, the scheduler selects a task with the highest priority to use the resource. Some algorithms are preemptive so that the highest priority task gets the resource immediately. Some scheduling algorithms, such as earliest deadline first, and least slack time have been proven to produce an optimal schedule for specific performance measure and under certain assumptions [LL73]. Other scheduling heuristics provide good performance under less restrictive assumptions, but will not be optimal [Pin95].

Schedules may be characterized as *delay* or *non-delay*. A delay schedule is schedule that can be improved by adding idle time between tasks to delay the start of a successor task. In the work presented here, we assume that all schedules are non-delay, that is, the addition of idle time will not improve the schedule in the relevant performance measure. Alternatively, we can say that for any delay schedule, a non-delay schedule exists with equal or better performance. With the appropriate restrictions on penalty functions, we will show in Chapter 6 that delay schedules are not necessary.

Scheduling is central to this research. We build on scheduling algorithms and methods to use them for our work. Some specific areas of scheduling which have direct relevance to our work are discussed in the following sections.

2.2.2 Imprecise Scheduling

Two techniques applicable to real-time scheduling are proposed by Lin, et. al. [LLS$^+$91]. In order to meet deadlines, they suggest that an *imprecise* computation be performed. The advantage of using such a computation is that the imprecision will allow a shorter processing time. In situations where there is not sufficient time to calculate a precise result, an imprecise result may be acceptable. There are two methods suggested to accomplish this. First, the *sieve* method requires the operating system to select from one of several algorithms to perform the necessary calculation. Ideally, each algorithm provides a trade-off between precision and processing time. If the time available is known, then the operating system can select the algorithm that executes in the available time and produces an result with acceptable imprecision. A second approach, the *milestone* method is applicable where the computation can be performed in an iterative fashion, such as successive approximations. With each iteration, the result becomes more precise. The operating system can terminate the process when either the result reaches an acceptable level of precision or there is no time remaining for processing.

The sieve and milestone methods each have several advantages and disadvantages. The sieve requires that several algorithms be available that can perform the necessary calculations. Furthermore, algorithms with longer processing times must produce results of greater accuracy. More importantly, the available time for processing must be known in advance of the selection so that the operating system can choose the appropriate algorithm. The milestone method, by contrast, does not require that the available processing time be known in advance. However, it does require that the algorithm be iterative in nature. That is, the algorithm must provide an approximate result quickly, and then, increasingly more accurate results as the algorithm continues. These methods suggest the notion of trading off precision with processing time, a concept that is important to our work. We propose algorithms, some of which are fast but imprecise, others more precise but slow. The work of [LLS$^+$91] provides a framework in choosing among algorithms depending on the requirements of the application.

In addition to the work cited above, Lin et. al. have also proposed several algorithms for scheduling tasks when imprecision can be permitted. In these algorithms, tasks are decomposed into a mandatory part and an optional part. The mandatory part is considered to have a hard deadlines, the optional part is considered to have a soft deadline. These algorithms attempt to schedule all mandatory parts of tasks and to schedule optional parts to minimize some error metric. The error metric indicates the consequences of not executing an optional part. The authors also discuss several error metrics, each with a different scheduling algorithm that minimizes it. For instance, in systems with task importance levels, error might be weighted by each task's importance. In an accompanying algorithm, a method to schedule the optional parts of higher importance tasks is presented. This suggests a bi-criteria approach, which is discussed next.

2.2.3 Multi-Criteria Scheduling

In this paper we focus on bi-criteria scheduling where we seek to maximize two goals: temporal and logical constraints. Lawler observes that if tasks are partially ordered, that is, they have logical constraints, then the solution space is reduced and the problem is often easier to solve [Law78]. However, since we consider that logical goals are not necessarily met, the solution space is *not* reduced—and consequently the problem may not be easier.

In general, there are several ways that two goals can be maximized. One is the *secondary criteria* approach where the secondary criteria is maximized, subject to meeting the primary criteria. For example, a *tardy* task is one that completes after its deadline. If several tasks are tardy, the measure of *maximum tardiness* (or *max tardy*, refers to the most tardy task. Consider a set of tasks to be scheduled with the secondary criteria: "minimize the number of tardy tasks subject to minimizing the maximum tardiness." In other words, from the set of schedules for which maximum tardiness is the minimum possible, choose the schedule that has the least tardy tasks. This is a common approach to bi-criteria scheduling, particularly where the primary criteria are due to logical constraints. Note that the secondary criteria approach only attempts to actually meet one of the two criteria, performance in the secondary criteria may be poor.

As an enhancement to the secondary criteria approach, the primary criteria can be relaxed so that the set of all schedules meeting, or "nearly meeting," the primary constraints is enlarged. With this change, it is expected that there is a likelihood of improved performance in the secondary criteria. The problem with this approach is that it is not clear just how much the primary constraint should be relaxed. It might be the case that a schedule with the same performance in the secondary criteria could have still been found, even if the primary constraint had not been relaxed as much. Alternatively, there is no way to know if, by relaxing the constraint just a *little bit* more, a schedule with *much* better performance in the secondary criteria could have been found.

A second approach to bi-criteria scheduling is the *alternate criteria* approach. In this approach, we seek to find the best schedule in the measures of both criteria without bias toward either. In this way we explicitly make the trade-off between the two measures and select a schedule which meets, or comes close, to meeting both [CLB94].

Hariri and Potts [HP94], propose a solution to a scheduling problem that is somewhat related to our work and that of Locke [Loc86]. Hariri and Potts consider a problem of scheduling tasks on a single machine. Each task has a due date by which time the task is expected to complete. Each task also has a deadline by which time the task must complete, although with some penalty. Hariri and Potts propose both a dynamic programming algorithm and a branch and bound algorithm. This penalty function is similar to the notion of a *value function*, which specifies the consequences of completing at various times. We discuss an extension of this simple step function to a function of several steps later in this work.

A solution to the bi-criteria scheduling problem is discussed by [VCY95], a branch and bound algorithm is proposed to minimize the total tardiness of a job set keeping the number of tardy jobs to a minimum. That is, for the set of schedules that produces an optimal solution with respect to total tardiness, the secondary goal to find a schedule that results in a minimum number of tardy jobs. Branch and bound provides a useful basis for heuristic, and we propose to use branch and bound as a possible solution method.

The trade-off between performance measures in the bi-criteria problem leads naturally to the notion of a *frontier*. French suggests a frontier to conceptualize the orthogonal goals arising from independent performance measures [Fre82]. Given a set of n tasks, there are a number of possible schedules and, for each possible schedule, there are two measures of performance. If each of these measures is plotted, with each axis representing one measure, then the resultant plot is a scatter diagram. The envelope of this scatter diagram defines the frontier. More precisely, of the set of all possible schedules, a schedule is on the frontier if there does not exist a schedule that is at least as good in both measures and strictly better in one. French refers to such schedules on the frontier as *efficient*. Although we may refer to a single *point* on the frontier, this point may actually represent many schedules. The frontier described by French provides a strong motivation for our work, however, French only discusses a trade-off between two temporal measures, he does not discuss the possibility of relaxing precedence constraints. The frontier plot is further described and illustrated in §3.2.5.

The concept of a frontier is the mechanism that we will use to formalize the trade-off between temporal and logical consistency. The characteristics of the frontier depend on the actual problem, however in the general case, a trade-off can be made. The notion of a frontier can easily be extended to the multi-criteria problem where there are m measures of performance and the frontier is m dimensional.

2.2.4 Value-Based Scheduling

Of interest to our work is value-based scheduling developed by Locke [Loc86]. In this model, rather than define a single temporal attribute (i.e. deadline) for a task, a function is defined that indicates the relative value of completing a task at some instant of time. The scheduler then attempts to schedule tasks in such a way that the total system value—that is, a combination of the values attained by each task—is maximized. In general, a value function can be any function of value vs. time. With value functions, both the urgency and importance of the tasks deadline can be defined as well as utility of completing the task after the deadline has passed. Thus, value functions can capture more general temporal consistency requirements than a single deadline can express.

An advantage of value-based scheduling is that a single performance measure need not be chosen for the system. Performance measures such as deadlines missed, average lateness and average tardiness are not, in general, all optimal for the same schedule. Furthermore, a performance measure that is suitable for some transactions may not capture the performance goals of all transactions. Value-based scheduling, by contrast, allows each task to specify its unique performance goals through the value function. By scheduling to maximize the total system value as the sum of the value attained by each task, the performance goals of each transaction can also be maximized.

The total value to the system is achieved by some combination of the value attained by each task. Usually that combination is simply the *sum* of all task values. Another choice might be the *maximum* value attained by all tasks. Through the specification of task value function and how task values are combined to form total system value, many commonly used performance measures can be modeled.

Locke developed a scheduling heuristic called a *best-effort* scheduler that sought to maximize value. In his model, Locke derived deadlines from value functions and used earliest-deadline-first (EDF)[1] to determine if the system could schedule the tasks to meet all deadlines. If EDF failed, his scheduler would determine which tasks not to schedule based on the amount of value that they would impart to the system [Loc86].

Value based scheduling is not without problems however. Scheduling using value functions can be a difficult problem. At the very least, value functions must be restricted, such as to one of several classes—which is the approach of Locke. Most commonly used performance measures, such as max-tardiness, number of deadlines missed, act as a *proxy* for a value function. A proxy is an easily measured performance characteristic that represents the true characteristic. For example, a deadline is a proxy for a value function. The deadline is a single point measure of timing requirements that captures only a small part of the value function, but it is easier to understand and use. By using the proxies, the scheduling problem becomes more readily solvable, and in the aggregate, the desired scheduling performance can often be achieved. Yet, using a proxy prevents the scheduler from considering more aspects

[1]In Operations Research, the term Earliest Deadline Due (EDD) is used instead of EDF, the two are used interchangeably here.

of timing requirements and instead imposes a “one size fits all” methodology on the task timing requirements. Using a proxy, all transactions or tasks are optimized to the same measure.

Locke made several assumptions about the form of value functions that enabled the development of an effective heuristic. For example, all tasks were assigned deadlines so those tasks could initially be scheduled by EDF. For many functions, the deadline is obvious, but in others, the assignment of a deadline is somewhat subjective. But, in any case, it is clear that the use of arbitrary value functions makes developing a reasonable heuristic unlikely.

Locke did not consider concurrency control of other shared resources or bounding the logical imprecision that his scheduler’s preemption might induce. Nevertheless, Locke’s work forms a strong basis for flexible treatment of temporal consistency in scheduling tasks. However, to be used here in a unified scheduling technique, it needs to be augmented to also handle logical consistency of other shared resources and data required in a database as well as augmented to handle the inevitable trade-off between the two forms of consistency.

In addition to the work of Hariri and Potts, the work of Clark is also related to the work of Locke. Clark extends the work of Locke to include precedence ordering, although he does not consider the possibility of violating precedence ordering [Cla90]. Clark proposes that the value of completing a task includes not only the value of completing that task, but also the values contributed by predecessor tasks. In this way, the values of predecessor tasks are combined with the successor tasks. Task importance is calculated to include the precedence chain rather than just the final task. Clark, however, does not consider the possibility of violating precedence constraints.

Some of the work here will use a notion similar to value functions. We refer to these functions as *penalty* functions. Through this penalty function, we will define multiple deadlines for a task and then associate a penalty to be assessed for missing each deadline. Penalty functions will be discussed in detail in Chapter 6.

2.3 Concurrency Control

Concurrency control seeks to uphold a logical consistency correctness criterion. Concurrency control insures that tasks processed concurrently do not interfere with each other such that an incorrect result is obtained. In the context of this work, we use the term *concurrent* to refer to tasks that are interleaved or *time-shared*.

There have been many concurrency control mechanisms developed for databases [BHG86]. Concurrency control techniques typically work by limiting the degree of concurrency among tasks. However, this can have the undesirable effect of eliminating some benefits of concurrency. Thus, in any concurrency control mechanism, a trade-off must be made between the complexity of the algorithm and the restrictiveness of the controls. For example, a simple concurrency control such as two-phase locking may guarantee correct results but may limit access to shared data. Overly restrictive access to shared data may reduce transaction throughput to unacceptable levels. More complex concurrency algorithms attempt to permit greater access to shared data by examining more closely the nature of the data contention to see if shared access is possible. These controls may provide better database throughput for certain types of data but are more difficult to implement and may require high processing overhead.

Another class of concurrency control protocols is optimistic concurrency control. This control seeks to maximize throughput by placing no restrictions on planned operations under the assumption that conflicts are unlikely to occur. Instead of attempting to decide whether a transaction should proceed in advance, optimistic concurrency control simply allows the transaction to proceed, but then analyzes the result to see if a conflict did occur which led to incorrect results. In the rare case an error occurred, the transaction must be aborted. If the transaction has committed a rollback will be required [BHG86]. Even if a rollback is not required, the transaction may be initiated several times before it is finally committed successfully, wasting valuable time. However, if the assumption of limited conflicts holds, then optimistic concurrency control systems can perform quite well, as they trade the overhead of checking for conflicts prior to each transaction to simply correcting problems when they occur. Rollback, if required, puts the system in the state of a prior instant in time, but in a real-time system it is often not possible to "go back" in time. Sensor inputs and other real-time events cannot be rolled back and it is often not possible to re-create older states. During transaction rollback, time is moving forward—making it even more difficult to meet future deadlines.

We do not consider the use of *optimistic concurrency control* here because of the issues of re-submitting or rollbacks. Instead, we assume that the system or database will continue operation and either accepting or compensating for any errors introduced by violations of concurrency control.

2.3.1 Non-Serializable Concurrency Control

A simple method to avoid interaction between operations ready to be performed at the same time is to simply execute them serially, that is, begin the second task only after the first completes. This, however, leads to unacceptable delays in processing while transactions waiting for external events, stall all pending transactions. By allowing concurrent, or simultaneously executing transactions, this delay is practically eliminated, but this raises the possibility that transactions will conflict with each other. A transaction interleaving is *serializable* if there exists a serial ordering of transactions which produces the same result. Thus, serializable transactions cannot produce errors due to interleaving. Most concurrency schemes are based on serializability.

Serializability is not the only way of maintaining logical consistency. Work in non-serializable concurrency control includes techniques that use the semantics of the application to gain information about allowable interleaving of transactions. Garcia-Molina [GM83], classified transactions into semantic types based on what they do in the database. For each type, a compatibility set is defined to identify which other types are compatible with, i.e., may interleave with, the given type. Instead of serializability, compatibility sets determine correctness. The work of Garcia-Molina is expanded by both [Lyn83] and [FO89]. The Real-time Semantic Objects and Constraints (RTSORAC) model [PDPW94] incorporates features that support the requirements of a real-time database into an extended object-oriented model.

Concurrency is enforced though allowable concurrent actions within a relationship, defined between two or more objects. This concept is based on the semantic data modeling notions [PM88].

RTSORAC uses ad hoc correctness criteria and therefore any imprecision that results may be unbounded. Other concurrency control techniques have been devised, based on more formal correctness criteria, that are designed to bound imprecision explicitly. In Kuo and Mok [KM92] similarity is defined as a correctness criterion that allows data that are "close enough" to be considered the same. The Similarity Stack Protocol (SSP) described by [KM93] defines similarity of data based on the time at which the data is written, i.e., two data items are considered to be similar if their timestamps are within a specified bound. Transactions are placed on a scheduling stack according to their priorities. Read/write events of different transactions may swap positions on the stack as long as they are similar.

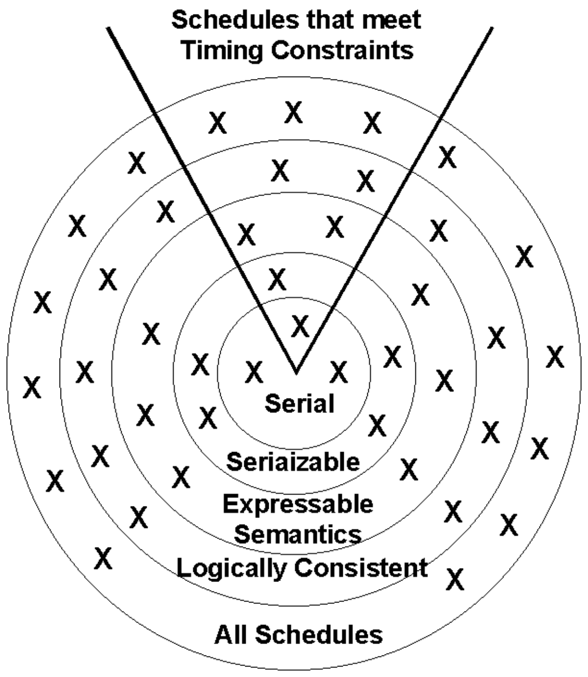

Figure 2.1: Hierarchy of Concurrency Controls

Figure 2.1, proposed in [DiP95], shows the benefits of non-serializable concurrency control as the set of schedules that are acceptable with respect to logical requirements is enlarged. As concurrency controls are relaxed, there are more schedules available, thus increasing the possibility that one of those schedules also meets temporal requirements. Enlarging the set of logically consistent schedules also increases the probability that the set will include a greater number of temporally consistent schedules.

2.3.2 Epsilon Serializability

Several concurrency control techniques have been designed to maintain Epsilon Serializability. Ramamritham and Pu suggest that allowed concurrency errors in a linear data space result in a predictable error in the data [RP]. By placing a limit on the allowed error, transactions can proceed in spite of concurrency controls so long as the error limit has not been reached [WYP92]. In this way, the errors are predictable and bounded. Concurrency control techniques are described by [WYP92] in which read-only transactions need not be serializable with other update transactions, but update transactions must be serializable among themselves. The techniques are variations of two-phase locking, timestamp ordering and optimistic concurrency control. The work in [RP] suggests that imprecision can be bounded and ultimately corrected.

Epsilon serializability was used in [DiP95] as a way to relax logical consistency with the hope that temporal consistency could be met. Figure 2.1 shows the expected increase in temporally consistent schedules through the use of less stringent concurrency control methods. However epsilon serializability differs significantly from the other concurrency control methods. While the other methods all provide greater concurrency over serializability, they all require that the concurrency control rules be followed and in return, the integrity of the database is maintained. Epsilon serializability, by contrast, permits violation of the concurrency mechanism in return for bounded imprecision in the data. Thus, the benefit of epsilon serializability is not fixed, but controlled by the imprecision allowed.

Epsilon serializability requires that the database data space be a *metric space*. The values of all data in a database define the state of the database. For the data to be a metric space, several conditions must be met. First, a *distance* function must be defined such that absolute value of the difference between the precise value of a data item and the recorded imprecise value can be measured. Second, the distance is *symmetric*; the distance from the recorded value to the actual value is the same as the distance from actual to recorded. Hence, the distance $\varepsilon = |V_a - V_r| \geq 0$ (where V_a is the actual or precise value, and V_r is the recorded or imprecise value). Finally, the *triangle inequality* must hold so that the distance from one object to a third is no greater than the sum of the distances from the first object to the second and the second to the third or, $\varepsilon_{total} \leq \varepsilon_1 + \varepsilon_2$. The consequence of these conditions is that the increase in imprecision due to multiple violations of logical constraints is no greater than the sum of such increases for each violation individually. This is important since it guarantees that errors in the database (or system) do not increase in a non-linear fashion. If the conditions of a metric space are not met, then the logical constraint violations cannot be made since there would be no way to measure the accumulated penalty. Imprecision may be *imported* into the system from the introduction of imprecise results being used to update a data item. Imprecision may also be *exported*. This occurs if imprecise results are provided as a result of a query. Exported imprecision is the easier to deal with since it has no lasting effect on the database. Imported imprecision remains in the database and can propagate through the database with successive transactions.

In our work, we will require that the data form a metric space. The possibility that the consequences of logical concurrency violations can be measured, and possibly bounded, allows us to propose such violations. Epsilon serializability, moreover, sets an upper bound on the error thus created, any violations that would cause that limit to be exceeded are not permitted. In the work presented here, we do not set an explicit bound. Such a bound would always be in jeopardy of being violated. Indeed given a large enough penalty for missing a deadline, almost any logical constraint would fail. In our case, a bound, if set would be used to signal the system that corrective action must be taken as proposed by [WYP92]. Nevertheless, epsilon serializability provides some assurances, that, given certain assumptions are met, violating logical constraints does not lead to unbounded error. We also apply the concepts of a metric space and epsilon serializability to non-database operations.

Using the techniques reviewed here, we plan to generalize concurrency control to an enforcement of logical consistency that is expressed as precedence orderings among task or transaction operations. Relaxations of logical constraints are permissible as they lead to bounded levels of imprecision in the database. These violations are the embodiment of the trade-off to gain temporal consistency.

2.3.3 Precedence Ordering

The effect of a concurrency control technique on a system is the enforcement of a set of operation orderings that are a subset of all possible orderings. Thus, all *pessimistic concurrency control* systems can be modeled by constructing a set of partial orderings that restrict operations or tasks from executing during times that the concurrency control technique prohibits. By completing tasks in a permissible order, the concurrency requirements are met. In our work, we permit the selective violation of precedence constraints. We use the notion of penalty to capture the imprecision that accrues as a result of violating a precedence constraint. An example of a typical concurrency control technique's set of permissible orderings is the set of orderings representing serializable schedules. All possible schedules of transaction tasks, and therefore all permissible schedules, can be expressed as a partial ordering of tasks [BHG86].

Partial ordering is commonly used to express ordering among objects or tasks. A partial order $\prec$ on a set is *transitive* and *irreflexive* relation defined on some ordered pairs of elements. Thus if $x \prec y$ and $y \prec z$ then $x \prec z$ but $x \prec y$ and $y \prec x$ cannot both hold [Bol90]. Unfortunately, precedence constraints cannot express all possible orderings of objects. For example, consider a requirement where only two orderings are permissible: $a \rightarrow b \rightarrow c$ or $c \rightarrow a \rightarrow b$.[2]. Any set of precedence constraints to specify this ordering would require that both $a \prec c$ and $c \prec a$, which is not permissible. For a generalized tree structure, which is necessary to capture all orderings, a different notation is required. Such a notation has been described by previous work by [JS95] and may prove to be useful here. However, we will continue

[2]We use $a \rightarrow b$ to denote an instance of a sequence The notation $a \prec b$ denotes a constraint.

to refer to partial orderings by the conventional notation with the assumption that a more complete notation could be used in applications that require it.

We have chosen partial ordering as our primary control over logical consistency since partial ordering can be used to specify any arbitrary ordering. Since all pessimistic concurrency controls ultimately define an acceptable ordering of tasks, such an ordering can also be specified using partial ordering. Thus, if our work is valid for partial ordering, it must be valid for any other pessimistic concurrency control mechanism. In practice, the methods described in this work could be easily modified to use the concurrency control mechanism most appropriate to the application rather than partial ordering.

2.3.4 Real-Time Concurrency Control

In a real-time system, concurrency control cannot ignore temporal consistency requirements. Instead, concurrency control must take into account both logical consistency and temporal consistency of data and operations. Several real-time concurrency control techniques have been proposed that apply real-time scheduling concepts to traditional concurrency control protocols, such as two-phase locking [YWLS94]. Other research in real-time concurrency control has found that the serializability requirement is too restrictive. By relaxing this restriction, and therefore increasing concurrency, there is a possibility of meeting more timing constraints as shown in Figure 2.1. A comprehensive survey of real-time concurrency control techniques is presented in [YWLS94]. The notion of relaxing logical constraints to improve temporal performance is similar to our work. However, as we have seen, this relaxation approach cannot guarantee better temporal performance. Instead it simply enlarges the solution space with the hope that a better solution can be found. Our work seeks to explicitly make the trade off between logical and temporal performance.

Logical consistency can be mapped to a *penalty* table that expresses the penalty to be assessed if a violation of the required task ordering occurs. In a database system, the required orderings are often imposed by serialization orderings. Pessimistic concurrency control schemes impose a subset of allowable task orderings on the set of all orderings, which is equivalent to imposing partial orderings on tasks. In an applied setting, use of a more conventional concurrency control scheme might be desirable. Thus, a trade-off between temporal and logical constraints can be achieved independent of the pessimistic concurrency control method selected.

We have described the work of other researchers that is relevant to our work. We have developed a scheduling technique that uses a combination of penalties for violating temporal constraints with penalties for violating logical constraints. By scheduling tasks such that total penalty is minimized, the system meets, as closely as possible, the logical and temporal requirements imposed.

Chapter 3

Methodology

3.1 Incremental Approach

In the chapters that follow, we develop a unified scheduling technique that synthesizes scheduling and concurrency control to handle both logical and temporal consistency in a database or other task processing system. Our approach is to incrementally work towards this goal by first considering a simplified model, then exploring models that are more complex. The first models will serve to illustrate the basic problem and verify the legitimacy of a trade-off between logical and temporal constraints. The simple models also have the advantage being easily solved and lead to relatively fast algorithms that provide optimal solutions. The simple models are also useful in solving real problems in situations where the limitations placed on the model are valid. Models that are more complex will be developed which relate to general scheduling problems. For these models, algorithms that guarantee optimality are not so easily constructed. We propose to extend our simple models in small steps so that results from one model can be applied to the next.

In formalizing each incremental model, we have followed a similar procedure. First we describe the model, then provide a formal definition using a notation that expresses the problem as an IP (Integer Programming) problem. An IP is an linear optimization problem (LP) in which some or all of the variables must be nonnegative integers [Win93]. An IP can be solved using one of several commercially available software programs, such as LINDO [lin94a] or CPLEX [cpl94]. While LINDO or CPLEX will find an optimal solution to the IP, this method is generally not suitable for real-time scheduling due to the protracted processing time required. However, solving the IP formulation provides valuable insights into the problem and provides optimal solutions—at least for reasonable sized problems—against which any heuristic can be judged.

For practical solutions to the scheduling problems discussed here, we develop heuristics that are capable of finding a schedule quickly, but without any guarantee of optimality. In many cases, it is desirable to have several heuristics available, each tailored to the specific needs of the application for which it will be used. It is generally true that the performance of a heuristic for complex problems such as these cannot be mathematically proven. Instead, a heuristic is evaluated using one of three methods [LLKS85]. Performance can be determined using analysis by *worst case*, *probabilistic* or *empirical* measures.

Worst case analysis provides a lower bound guarantee on heuristic performance, but often, the worst case is both unlikely to occur and may be dismally poor. A worst case analysis provides a mathematically correct proof the worst case, but for complex problems, this proof may be difficult to construct. A probabilistic approach attempts to overcome the problem of the worst case analysis by providing an average or expected level of performance. Probability distributions are used in the analysis to find the expected performance of the heuristic. However, the validity of probabilistic predictions depends on assumptions about probability distributions, which may not be well known. Actual results may vary significant from probabilistic predictions if the distributions assumed are not correct. Finally, empirical testing can provide useful measures of heuristic performance using typical problems for which the heuristic will be used. Empirical analysis can make no guarantee of the general accuracy of the heuristic. The accuracy of the empirical analysis depends completely on the similarity of the simulated data to the actual that will be used by the heuristic. This problem-specific data will vary from problem set to problem set. However, empirical analysis has a strong advantage in that it is relatively simple to implement simulation, even for very complex heuristics. If the data sets are similar to the actual data, then the results are indicative of the results that will be achieved with real data.

For this work, we use an empirical approach to measuring heuristic performance. Simulation is used with randomly generated task or transaction sets and the results tabulated.[1] Simulation results will be analyzed against theoretical results and we will determine the usefulness of the heuristic in specific applications. The results from simulation are not, in general, valid for applications that differ in task characteristics. We have attempted to generate tasks set in a most general manner so as not to bias the results. However, simulation of heuristic performance and tuning of heuristics must be performed for each application. This is a major limitation of simulation analysis.

3.2 General Scheduling Model

3.2.1 Tasks

Our scheduling model consists of non-preemptable units of processing that we refer to as *tasks*. Tasks are common in various applications, such as a complete operation performed on machined part, or a maneuver of an underwater vehicle. In a database system, sets of tasks to respond to user request are referred to as *transactions*.

[1]All simulation testing conducted in the course of this work was performed on a Pentium 200 MMX class personal computer

In our work, we consider a transaction to be a set of partially ordered tasks, but otherwise ignore the transaction designation. Associated with each task is a known and fixed processing time. In scheduling, we assume both temporal and logical constraints. The temporal constraints require that tasks conform to some measure of timeliness, for example—they meet a deadline. Logical constraints require that tasks be processed in a certain sequence. We will consider the possibility of relaxing constraints in order to optimize performance.

An optional *temporal penalty function* is provided to capture temporal consistency. This penalty function consists of some number of deadlines and a positive penalty associated with each deadline such that the penalty is assessed if the deadline is not met. The penalty function considered here is a multiple step function and we assume it is non-decreasing. Penalty functions are discussed and illustrated in Chapter 6.

Tasks may also be related to other tasks through a precedence relationship. The total of these precedence relations imposes a partial ordering on all tasks in the system. For each precedence pair, there is a positive *precedence penalty* defined, which is assessed if the precedence ordering is violated in the schedule. We use the notation $a \prec b$ to express the logical constraint that task a precede task b in any schedule. Since we have assumed non-preemtable tasks on a single processor, $a \prec b$ implies that task a completes before task b begins. We consider this precedence ordering to be transitive, that is if $a \prec b$ and $b \prec c$ then $a \prec c$. However, we do not consider penalties to be transitive. The implication is that all penalties must be specified explicitly. The reason for this is that there is not a clear way to assign penalties through transitivity. For example, if the penalty for $a \prec b$ is 10 and the penalty for $b \prec c$ is 15, then consider the sequence of tasks $b \rightarrow c \rightarrow a$. Clearly, the constraints $a \prec b$ is violated, for which there is a penalty of 10. By transitivity, $a \prec c$ is also violated, but it is not clear what this penalty should be. If a is a predecessor task to c, that is, c in some requires a in order to be correct, then some penalty should apply. If c does not explicitly require a then no penalty is necessary. Not only is the necessity of a penalty unclear, but even if we agree that a penalty is justified, what that penalty should be is not obvious. Thus, we require that penalties be specified explicitly for all precedence pairs.

Precedence transitivity is specific to the application, and some applications may lend themselves to a more general rule. If, for a specific application, a rule exists for assigning penalties, then this rule can provide a shortcut to specify all transitive penalties.

3.2.2 Metric Space

Using precedence constraints to model logical consistency, we assume that for all precedence constraints the conditions for a metric space hold, as defined in §2.3.2. That is, for all precedence violations we assume that the error introduced is measurable. We also assume that the error is symmetric, that is the error introduced is the same whether viewed as the error added to give the imprecise result or the error subtracted to give the correct result. Finally, we assume that the triangle inequality holds. This final assumption is the most difficult to meet and it requires that the total error due to multiple precedence violations are not greater than the sum of the individual errors.

Consider an FMS system to illustrate the importance of a metric space. Assume that a part requires several machining operations, some of which are partially ordered. We require that if an operation occurs out of order, then the error introduced is measurable, for example, an increase in the reject rate. We also require that if a second operation is performed out of order, the resulting error can be added to the error from the first precedence violation, and the sum of the two is no less than the total accumulated error introduced (the triangle inequality). This is often the case in practice. For example, if a finish hole is drilled without first drilling the pilot hole, a penalty related to increased tool wear may be imposed. If a second hole is also drilled without a pilot hole, then the total tool wear is no greater than the sum of the wear caused by each hole taken alone.

If the conditions of a metric space do not hold, then violations of precedence order may not be tolerable. To continue with the FMS example, consider an out of order operation resulting in a part that cannot be completed. An initial milling operation, if skipped, may make it impossible to continue with the machining of the part. For practical applications we can handle this condition by using an arbitrarily large penalty such that a schedule that does not have the milling operation scheduled first is infeasible. Similar examples can be constructed for database systems, mission planning or other applications.

For the models that follow, we assume that the conditions of epsilon serializability hold. We do not require that the imprecision resulting from violations of logical concurrency controls be limited to a predetermined bound, but we do require that imprecision be bounded and known.

3.2.3 Schedule

We consider a schedule to be a sequence of all tasks known to the scheduler. An *optimal* schedule for a given regular performance measure is a schedule for which the aggregate performance measure for all tasks is less than or equal to any other schedule. A regular measure of performance (R) is defined as a measure that is non-decreasing in the completion times. Thus if R is a function of $C_1, C_2, \ldots C_n$ such that $C_1 \leq C'_1,\ \ C_2 \leq C'_2, \ldots C_n \leq C'_n$, then $R(C_1, C_2, \ldots C_n) \leq R(C'_1, C'_2, \ldots C'_n)$ [Fre82]. As a consequence of our assumption of non-decreasing time penalty function, there exists an optimal non-delay schedule if an optimal non-delay schedule exists. Thus, we will not further consider delay schedules. For this model, we have defined two performance measures: deadline (temporal) penalty and precedence (logical) penalty. These two measures are orthogonal since the value of one measure is unrelated to the value of the other.

The orthogonality of the measures requires additional consideration for scheduling purposes. There are two approaches to this problem and both require an *equivalence* between the two measures. The equivalence is an expression of the relative importance of meeting deadlines or performing tasks in the correct precedence order. The equivalence can only be specified by the user with knowledge of the application and the specific optimality required. Thus, the equivalence between temporal performance and precedence performance is assumed to be an input to the scheduling process. Note that the penalty values to be specified for deadlines and precedence ordering are entirely arbitrary in absolute magnitude. Since we do not specify the relative magnitude of temporal and logical penalties, it is up to the user to supply this information. As mentioned above, there are two approaches to using this equivalence information. First, the equivalence can be specified as a single value. For example: say a temporal penalty of value "10" is equivalent to a precedence of value "6." Using this information, the scheduler can find an optimal sequence of tasks reflecting this equivalence. A shortcoming of this method is that although the schedule is optimized for this single value, the user has no way of knowing if a "better" schedule exists with a equivalence just slightly different then the equivalence specified—for example, a schedule with much better temporal performance with just slightly less logical performance. This limitation leads to the second approach: the formulation of a frontier. The *frontier*, as introduced in Chapter 2, is the set of points that define schedules for which there are no other schedules better in both measures. From this set of points, a single schedule can be chosen based on the trade-off between the two measures. With the frontier, the user can see the implications of this choice, and the range of optimal schedules available. In the next section, the frontier is discussed in greater detail.

We use the frontier to express the optimality between two performance measures. This concept is easily extended to m performance measures. The optimal schedules lie on the m-dimensional surface defined as the frontier.

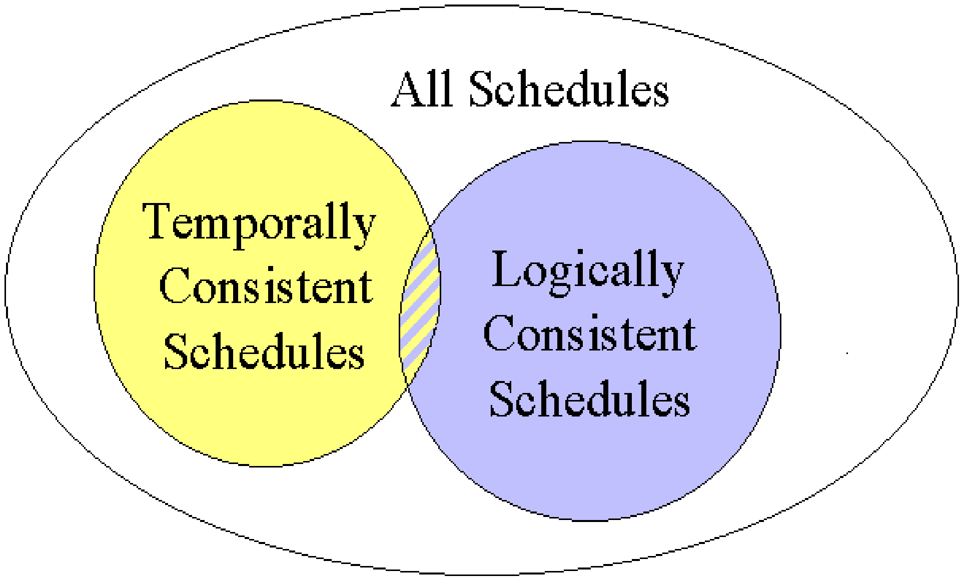
All Schedules
Temporally Consistent Schedules
Logically Consistent Schedules

Figure 3.1: Logical vs. Temporal Constraints in Scheduling

3.2.4 Bi-Criteria Scheduling

In our research, we are concerned with the scheduling of tasks where there are two, or possibly more, performance criteria. This *multi-criteria* measure complicates the scheduling problem, especially where the criteria conflict with each other. The *bi-criteria* scheduling problem (that is, two criteria) where both measures relate to time, has been studied extensively in Operations Research. For example, finding a schedule among tasks with deadlines that minimizes maximum tardiness while also minimizing the number of deadlines missed is both representative and commonly discussed, for example see [CLB94] or [VCY95]. However, the possibility of using bi-criteria scheduling where one measure is logical constraints is not considered in the literature. Note that in this work, logical constraints and temporal constraints can be relaxed; that is, they are not absolute requirements.

We consider the bi-criteria scheduling problem where one measure relates to time and the second relates to logical constraints. These measures can conflict, such that a schedule that performs well in one measure does not perform well in the other. On a single machine, assuming a non-delay schedule, there are $n!$ possible ways to schedule n tasks. Of the set of all possible schedules, there is a set of schedules that meet timing constraints and another set of schedules that meet all logical constraints. This is illustrated in Figure 3.1. If sets of temporally consistent schedules and logically consistent schedules are not empty and they overlap, then the intersection of these two sets contains the set of optimal schedules. These sets may not overlap however, and thus there will be no schedules that meet both criteria.

Figure 3.2 presents this more difficult condition. Here, the sets of logically consistent and temporally consistent do not intersect, and thus there is no schedule that meets both constraints. If, however, we could relax each set of constraints slightly, as shown by the dashed lines, then the two sets would overlap. This provides a method of scheduling to meet both logical and temporal constraints, since from the viewpoint of the scheduler, an optimal schedule does exist. In reality, neither logical nor temporal constraints are strictly met by a schedule in the overlapping region, however each constraint is *almost* met. If it were possible to control the size of each area outlined by the dashed line, then we could control the trade-off between the two constraints.

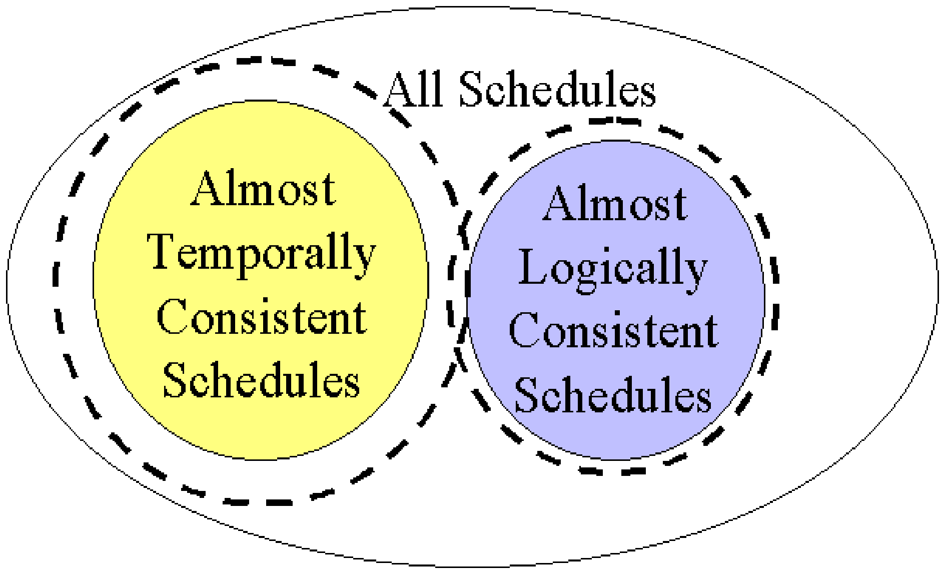
All Schedules
Almost Temporally Consistent Schedules
Almost Logically Consistent Schedules

Figure 3.2: Relaxing Logical and Temporal Constraints

3.2.5 Frontier

As we discussed in Chapter 2, the set of best schedules are defined by a frontier. This frontier is the outer edge of schedules on an x-y scatter plot that score high in both measures of performance. The frontier represents the trade off between the measures of performance and gives the user the option of choosing which schedule embodies the desired trade off.

Assume we can evaluate a schedule of tasks by two measures of performance. In the work presented here, we use one measure that relates to temporal performance and one to logical performance. If we enumerate all schedules and evaluate each for both measures, a set of (x, y) points is obtained. By plotting the points on a two dimensional plane, the resulting scatter diagram illustrates the frontier, and is shown in Figure 3.3. The points marked *a, b, c, d, e,* and *f* define the frontier.

That is, for each point on the frontier, there is no point better or equal in one measure and strictly better in the other [Fre82]. Thus, the optimal schedule can be chosen from among the schedules represented by the frontier. Points on the frontier represent schedules that are better than any other in some measure or combination of measures. The actual point to be chosen depends on the desired trade-off between logical and temporal performance. This choice of schedule from other schedules on the frontier is fundamental to this work. The choice must be made based on the requirements of the user, but in reality, it is more complex. In advance, we can propose a relative importance of temporal versus logical requirements. From this implied equivalence, it is relatively easy to choose the frontier point that most closely embodies this equivalence. As is seen in Figure 3.3, point c might most closely match the pre-determined equivalence between logical and temporal requirements. However, consider point d, this point provides much better performance with respect to temporal performance with only minimal loss of logical performance, and in spite of the pre-determined equivalence, point d might actually better meet the scheduling requirements. Thus, not only must the equivalence between logical and temporal constraints be known, but also the shape of frontier is necessary in order to make the best decision.

We seek to find a method to determine the frontier points, that is, to find schedules that correspond to the frontier points. Strategies include enumerative methods such as *branch and bound* or heuristic methods. Branch and bound performs a tree search for all schedules, by either following a branch to the leaf, or pruning branches that clearly cannot lead to an optimal solution. This pruning is important since for n tasks, the number of possible schedules grows as $n!$. Enumerative methods are generally limited by the problem size, but through pruning, larger problem sizes can be accommodated. Branch and bound algorithms will be discussed in detail in

Chapters 6 and 7. Heuristic methods are also developed that solve this scheduling problem much faster, and are not limited by task set size. However, heuristic solutions presented here do not guarantee a solution within a bounded distance of optimal. We use simulation to measure the performance of heuristics for randomly generated task sets.

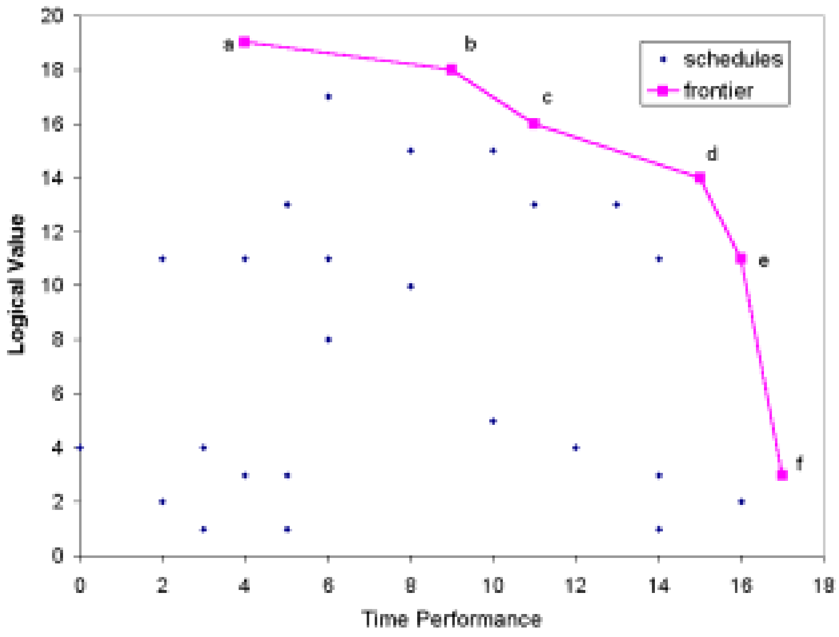

Figure 3.3: A Typical Frontier Plot

3.2.6 Problem Model

In the sections to follow there are several assumptions that apply to all models and will be discussed here.

Single machine. We assume all processing is performed on a single processor or machine. Furthermore, tasks being executed are not preemptable.

Static scheduling. We assume a *static* scheduler. That is, all tasks are known to the scheduler prior to the execution of the first task. All task times and penalty functions are also known. The scheduler finds the best schedule or schedules for this set of tasks. If it is necessary to reschedule after some tasks have executed or more tasks are entered into the system, then a *dynamic* scheduling system might be considered. This is discussed further in Chapter 9.

Non-delay schedules. For the models to follow, we consider only non-delay schedules. A *non-delay* schedule is one that does not benefit by the addition of idle time between tasks. This assumption places strong restrictions on the penalty functions that can be considered. Scheduling where *delay* schedules might be optimal requires very different techniques. For example, enumerative search methods are ineffective.

Define a penalty function to be non-decreasing. Thus for deadline $d_2 > d_1$ penalty $\pi_2 \geq \pi_1$. We propose the following lemma:

Lemma 1: *Assume a set of tasks to be processed on a single machine. Assume each task as a known processing time and penalty function. If the penalty function is non-decreasing, then there exists for any delay schedule, a non-delay schedule with equal or better performance for any regular measure of performance.*

Proof: Assume a delay schedule with idle time σ_n after task n. Setting σ_n to zero, will cause all successor tasks to n to complete earlier than they would with $\sigma_n > 0$. Since all tasks have a non-increasing penalty function, the penalty (π_n) for task n completing at time C, must be less than the penalty (π'_n) for that task completing at time C' if $C \leq C'$. Thus, the sum of all task penalties with no delay must be less than or equal to the total penalty with delay. $\square$

Hard vs. Soft Deadlines. We assume that all deadlines specified are *soft* deadlines, and hence can be missed—albeit with penalty. *Hard* deadlines, on the other hand **must**, by definition, be met. A failure to meet a hard deadline can be considered a failure of the system [Jen96]. Since the trade-off between temporal and logical consistency is core to our work, we assume that any deadline is a candidate for being missed. Hard deadlines can not be accommodated by the methods we present here. This is not a significant restriction however. Hard deadlines are a special case outside the continuum of soft to firm deadlines. In nearly all applications, the notion of firm deadline is sufficient. A task with a firm deadline imparts no value to the system if it misses its deadline. A substantial penalty is usually sufficient to insure that the firm deadline task is scheduled to meet its deadline. Typically, in a hard real-time system, worst case analysis is used to insure sufficient resources are available to guarantee that deadlines can be met. In a hard-deadline system, it is wise to dedicate the system to only hard tasks. Tasks with soft deadlines should be processed elsewhere.

3.3 Formal Model

The above description introduces parts of the model in a somewhat informal way. Here we now define a formal description of our model.

The model consists of a sequence of partially ordered tasks, $1, 2, ..., N$. Then, for every task i there is a temporal penalty function $f_i(C_i)$, where C_i represents the completion time of task i. We restrict this penalty function as appropriate in the models to follow. Each task i also has an associated set S_i of successor tasks. For every task $j \in S_i$, there is a ξ_{ij} that represents the logical penalty incurred if j is scheduled before i. Also for every $j \in S_i$, there is an associated decision variable, x_{ij} that takes the value 0 if i is scheduled before j and 1 if j is scheduled before i. Given this model, the scheduling goal is to minimize total temporal penalty and minimize total logical penalty. That is:

$$n/1//min \sum_{i=1}^{N} f_i(C_i), min \sum_{i=1}^{N} \sum_{j=1}^{N} \xi_{ij} x_{ij}$$

where the notation $n/m/A/B$ is used to classify scheduling problem of n tasks on m processors with A task flow pattern between machines, if any, and B performance measure [Fre82]. The bi-criteria goal is to both minimize temporal penalty and minimize penalties for logical constraints violations. This goal is accomplished through the selection of the decision variables, x_{ij}, subject to the appropriate constraints. The IP constraints of this model and the IP models to follow are not relaxed, unlike the temporal and logical constraints of our scheduling problem.

This problem is formulated as an objective function to be minimized. The bi-criteria objective, temporal and logical penalty, is a function of the sequence of scheduled tasks. For each sequence, a measure of temporal performance and logical performance can be calculated. These logical/temporal performance pairs define the frontier from which an optimal schedule can be selected.

3.3.1 Restrictions to the Temporal Function

In this work, we use a temporal penalty function that specifies for each task the penalty for missing a deadline. The penalty function is a multiple step function, where the penalty for missing a deadline is constant until the next deadline is reached. A very large penalty for completing the task after some deadline implies a non-feasible schedule, such that any precedence violation would be acceptable to allow this task to complete prior the final deadline.

Chapter 4

Model: Makespan versus Precedence

We first focus upon the problem of generating a frontier for a scheduling problem where a common deadline is shared by all tasks. That is, for temporal performance we are only interested in the *makespan*, which is the total time to complete all tasks. Logical constraints are met through precedence ordering between tasks, such that tasks scheduled out of order are not performed. The model to be discussed here can be considered the general model presented in Chapter 3 with certain restrictions. In the following paragraphs, we list the restrictions to the general model that apply here.

4.1 Restrictions to the General Model

1. **The total makespan is the measure of temporal performance.** The measure of temporal consistency for this model is the total time it takes to process all tasks. In this model individual tasks do not have deadlines or functions to represent the temporal requirements of a task. Instead, the measure of temporal performance is the time is takes to complete the task set.

2. **Processing times are reduced for tasks executing out-of-order.** Tasks that are scheduled to execute after their successor tasks have their processing time reduced by a "time advantage" factor. For the example presented later in this chapter, we will further assume that only tasks that execute after all successor tasks have their processing time reduced, and furthermore, the reduction is to zero (i.e., the task is deleted).

Consider a partially ordered sequence of tasks, $1, 2, \ldots, N$. For each task i, there is a (possibly empty) set of successor tasks S_i such that task i precedes task j for all $j \in S_i$. Furthermore, for each successor task there is a penalty $\xi_{ij} \geq 0$ if task i is scheduled after task j.

Each task has a nominal processing time, $t_i > 0$. In addition, for each task i, a time advantage $\tau_{ij} \geq 0$ is deducted from the nominal processing time, if task i is scheduled after task j. We assume that $\sum_{j \in S_i} \tau_{ij} \leq t_i$, that is, the total time advantage possible for a task cannot be greater than the processing time for the task. The maximum time required to complete all tasks is $\sum_{i=1}^{i=N} t_i$, and the minimum processing time required is

$$\sum_{i=1}^{i=N} \left(t_i - \sum_{j \in S_i} \tau_{ij} \right).$$

Also, the maximum penalty possible is

$$\sum_{i=1}^{N} \sum_{j \in S_i} \xi_{ij},$$

and the minimum penalty is zero.

This problem can then be stated as follows: given a common deadline T by which all tasks must be completed, it is necessary to determine the set of precedence relations that will have to be violated to meet this timing constraint. That is, if a task is scheduled to execute after a successor tasks then its processing time is reduced possibly to zero. The precedence violation results in a penalty, however the reduced task processing time results in a shorter makespan. This is the basis for the trade-off.

4.2 Formulation for Makespan Problem

This problem can be formulated as an IP (integer program) optimization problem:

$$\min \sum_{i=1}^{N} \sum_{j \in S_i} x_{ij}\xi_{ij} \tag{4.1}$$

subject to:

$$\begin{aligned} \sum_{i=1}^{N} \left(t_i - \sum_{j \in S_i} (1 - x_{ij})\tau_{ij} \right) &\leq T && (4.2) \\ x_{ij} &\in \{0, 1\} && (4.3) \end{aligned}$$

The binary decision variable x_{ij}, if set to zero, indicates task i is executed before task j, and set to one otherwise. The objective function, inequality 4.2 is to be minimized by setting the decision variables to zero or one, subject to the constraint of equation 4.1 being met.

This problem is a knapsack problem, and a variety of techniques can be found in the literature tyat provide a solution. The knapsack problem is $\mathcal{NP}$-hard and thus a polynomial time solution is not available. Pseudo-polynomial time solutions do exist however, and we will utilize them when appropriate [SM90].

For generating the frontier, it is necessary to solve this problem for all knapsacks of size 0 to $\sum_{i=1}^{i=N} t_i$, where size in this instance is the total processing time allocated. A particular approach, using *dynamic programming*, for solving the knapsack problem with the largest possible knapsack will result in the generation of the entire frontier. Define $P(t)$ as the minimum penalty that results from constraining the makespan to $\sum_{i=1}^{i=N} t_i - t$, and τ_{ij} is the temporal benefit earned by violating the precedence $i \prec j$.

Task	**Description**	**Time**
1	$Read_1$	4
2	$Read_2$	2
3	$Update_1$	2
4	$Read_3$	3
5	$Read_4$	4
6	$Update_2$	3
7	$Update_3$	2

Table 4.1: Example Task Definitions

If there are no precedence violations, then makespan is maximized and $t = 0$. Thus, $P(0) = 0$. Then,

$$P(t) = \min_{\forall i,j} \left\{ \xi_{ij} + P(t - \tau_{ij}) \right\}.$$

That is, the solution for any fixed makespan can be found by a previous solution(s) plus some additional task precedence violation. Thus, the process of computing $P(t)$ for $t = 0, 1, \ldots \sum_{i=1}^{i=N} t_i$ can be used to efficiently generate all points on the frontier.

4.3 Example

Problem. As an example, consider a set of real-time database tasks including reads and updates. Each task has an associated execution time, and some of them are constrained by precedence relationships. Table 4.1 lists each task in the example along with its execution time. We will simplify the example with the assumption that the time advantage for any task $\tau_i = t_i$ if $i > j, \ \forall j \in S_i, \ S_i \neq \emptyset$, that is, there is no time advantage for task i, unless it executes after all successor tasks, and if there is a time advantage, it equals the nominal processing time. Note that there can be no time advantage for a task with no successor tasks. In other words, tasks executing after **all** successor tasks are effectively deleted. Table 4.2 displays the precedence

Precedence	**Penalty**
$1 \prec 3$	10
$2 \prec 3$	8
$3 \prec 4$	4
$4 \prec 6$	4
$4 \prec 7$	5
$5 \prec 7$	5

Table 4.2: Example Precedence and Penalties

orderings and associated penalties. To graphically depict the dependencies defined by the precedence constraints, Figure 4.1 shows a network representation of these tasks. Each numbered *node* is a task. The *directed arcs* connecting the nodes specify the order in which tasks must be performed. So, for example, task 7 should not be performed until tasks 4 and 5 are complete. As was discussed in Chapter 3, we do not assume that penalties for precedence violations are transitive. The only penalties that can be assessed are those shown in Table 4.2. Thus, in this example, if task 3 were to complete last, there would be a penalty of 4 due to task 3 completing after task 4. There would be no penalty for task 3 completing after tasks 6 or 7, even though a penalty might be implied by Figure 4.1.

The precedence constraints and associated penalties defined for this example are based on the semantics of the application. For instance, $Read_1$ and $Read_2$ are both specified to occur before $Update_1$. We have assumed for this example that $Update_1$ involves writing data read by $Read_1$ and $Read_2$. Because for both of these read tasks, $Update_1$ is the only successor task, if either read is scheduled after the update, it will not be executed. The penalties associated with the two precedence constraints are different to reflect the fact that $Read_1$ is more important to execute (i.e. less likely to be removed from the schedule) than $Read_2$. The user might specify an overall deadline by which the schedule must be completed, or wait until the frontier is generated to choose the best point.

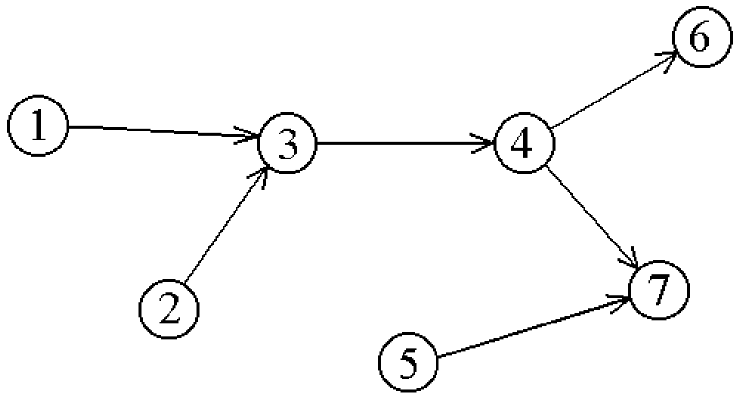

Figure 4.1: Precedence Constraint Network Diagram for Sample Problem

Solution. By the assumptions of the example, any tasks that are scheduled to execute after all successor tasks will be processed in zero time, i.e. those tasks will be deleted. If we consider all possible schedules, we can calculate, for any given sequencing of tasks, the sum of penalties as a result of tasks executing out of order. We can also calculate, for any sequence, the total time of execution, (makespan) of the schedule, which is obviously affected by out of order tasks that may not be executed.

In the knapsack problem [SM90], we seek to fill a container (knapsack) of limited capacity (weight) with a series of objects ("stones"). Each object has a weight and a value. The goal is to fill the knapsack such that total value is maximized without exceeding the capacity. If we begin with a schedule in which all tasks are scheduled after their successor tasks, then all such tasks are effectively deleted (i.e. an empty knapsack). This schedule has the shortest makespan, but also the highest penalty.

If we accept a slightly longer makespan, the question arises as to what task or tasks of the deleted set should be executed (by scheduling it prior to all its successor tasks). Tasks are added such that the resulting makespan is less than or equal to the longer makespan, and the reduction in penalty is maximized. We then again increase the allowable makespan, and again choose tasks to fill the time with the maximum reduction in penalty. Through this process, we are solving successive knapsacks, each of a slightly larger size. While at first, this appears to be unacceptably tedious, dynamic programming provides a solution to a knapsack of given size by using the solutions of smaller knapsacks. Thus, by using dynamic programming to solve the problem for the maximum makespan (all tasks included), we will solve the problem for all shorter makespans (some tasks excluded) in the process, and hence define the frontier.

The dynamic programming algorithm provides an optimal solution to the knapsack problem by starting with a knapsack of size 0 (ignoring for the moment, tasks 6 and 7). Assume a schedule such that all precedence constraints are violated. All tasks (which have successor tasks) complete after their successor tasks, and are therefore deleted (processed in zero time). These tasks form the set of deleted tasks. For this knapsack size of zero, the solution is simply the null set of tasks. All tasks that have successor tasks are scheduled to violate their precedence constraints, so that all these tasks are deleted. For our example, this means that only tasks 6 and 7 are scheduled. These tasks have no successor tasks, and thus cannot violate any precedence constraints. Figure 4.2 stage 1, shows the situation. Makespan is 5, but the penalty is 36.

In the general dynamic programming algorithm, there is assumed to be an unlimited number of "stones" of each size to fill the knapsack. In our case, we have a list of available "stones," and perhaps only one of any given size. This requires that some additional bookkeeping to track which tasks have been scheduled so they will not be considered at the next stage. Obviously, once a task is scheduled it cannot again be considered. When combining "knapsack" solutions from smaller knapsacks, it is possible that each of the smaller knapsacks have scheduled the same task—so again, we must keep track of the tasks included in each solution to verify each task is scheduled only once.

We begin with an empty knapsack, which corresponds to as many tasks as possible being deleted from the sequence. We then fill it to a successively greater weight limit (makespan), that is, we add tasks to the schedule. At each stage, we seek to maximize total value (penalty) so that only precedence violations with low penalty remain deleted. Since there is no penalty for tasks that are **in** the schedule, we seek to get the "high penalty" tasks scheduled first. Thus, for a knapsack of size 1, choose from the set of deleted tasks with processing time of 1 (or less), the single task associated with the greatest penalty. In our example, no task can be found, so the solution for 1 is the same as for 0.

For a knapsack of size 2, we consider a solution using either of the two previous solutions (knapsacks of size 0 and 1) and use the best one. Both are the same so either can be chosen. Next, add a single task such that:

1. The task hasn't been used in the previous solution,

2. The resulting makespan is equal or less than the specified deadline K (knapsack size), and

3. If more than tasks meets the above criteria, then chose the task with the greatest logical penalty.

For a knapsack of size 2, the situation is shown in Figure 4.2 stage 2. In this case there is one and only one task with processing time 2 so it is scheduled prior to task 6.

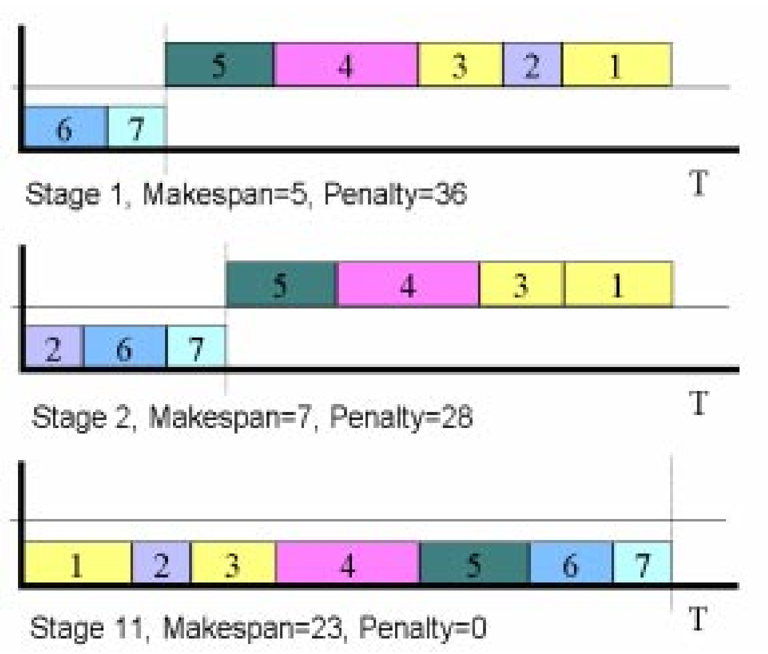
5
4
3
2
1
6
7
Stage 1, Makespan=5, Penalty=36
T
5
4
3
1
2
6
7
Stage 2, Makespan=7, Penalty=28
T
1
2
3
4
5
6
7
Stage 11, Makespan=23, Penalty=0
T

Figure 4.2: Several Stages in Dynamic Programming Procedure

In general, for knapsack of size K_N, we try to combine two smaller knapsacks such that the total weight is equal to K, that is: $K_0 + K_N$, $K_1 + K_{N-1}$, $K_2 + K_{N-2}, \ldots$ Next add a single task which meets conditions above. Of these possibilities, choose the best one (that is, the one that results in the lowest penalty). If no solution can be found for K, then the solution for K_N is the same as the solution for K_{N-1}. Figure 4.2 stage 11, shows the final stage: all tasks are scheduled so the penalty is zero, but makespan is 23.

The graph of Figure 4.3 shows the solution points for all possible schedules of the tasks in our real-time database example found by enumeration. Each point represents a makespan and penalty for one or more schedules. Of specific interest are the points connected by a series of lines; these points define the frontier. From the enumeration of all schedules, the frontier is easily found as was described in

§3.2.5. Dynamic programming however, finds the frontier points directly, without enumerating all schedules.

The frontier describes a set of schedules from which an optimal schedule can be chosen. Any point on the frontier is better than any other point on the frontier in some way. Only with a specified equivalence between the two performance measures can a single schedule be chosen. The specification of the equivalence is determined by the application for which the frontier is specified. Along with setting penalties, it is also necessary to choose the point on the frontier that meets the scheduling goals. Alternatively, it is possible to quantify the equivalence between logical and temporal requirements prior to scheduling, which then leads to selection of an optimal schedule directly. However, if an equivalence is defined at the start of the scheduling process, the choices for trade-off will not be evident.

In our example, if we require that all tasks be completed by a time of 10, then the subset of frontier points that can be chosen are (10,24), (9,26), (7,28), and (5,36). While these points meet the deadline, they suffer in database precision as evidenced by the high penalties. If the database user requires exact precision in the database, then he must be willing to wait for the entire schedule to complete, and miss the specified deadline. Table 4.3 shows the frontier points of our example, and gives a possible schedule of tasks which results in the shown performance measures. Recall that each frontier point can be generated by several different schedules. For purposes of illustration, we show only one schedule for each point, and indeed schedules with the same performance measure are equivalent.

While the restrictions described by this version of the model may seem to be rather limiting, there are real applications for which this model is sufficient. For example, consider the mission planning problem we introduced in §1.1.3 for autonomous undersea vehicles. A typical mission might include several high level goals supported

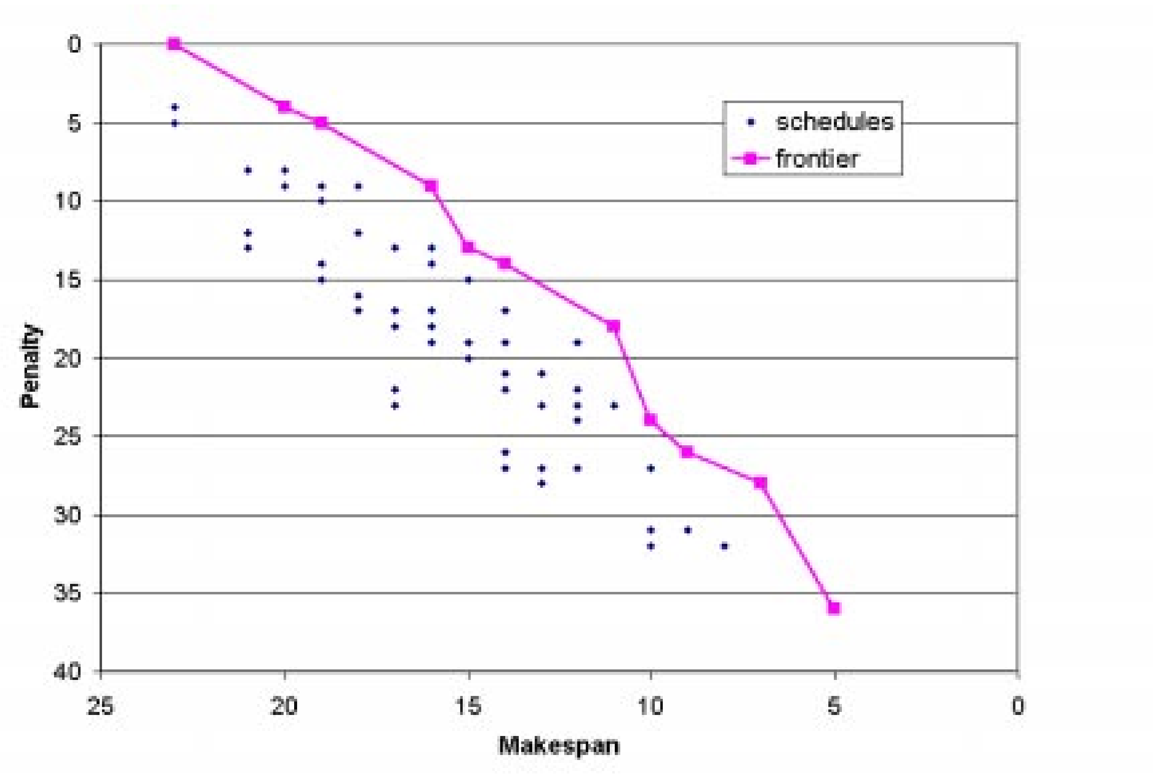

Figure 4.3: Frontier Plot for Example Problem

by lower level goals. For example, a “search” goal might be preceded by a “initialize” goal to properly orient and calibrate the necessary sensors used in the search. Thus, there are precedence constraints between tasks, with a penalties for out-of-order execution. In this example, failure to initialize the sensors prior to the search operation might degrade the quality of the data. Once the search is completed however, there is no need to initialize the sensors. If there is a deadline by which the mission must be completed, it is clear that by scheduling the search prior to calibration, the mission can be completed sooner. Thus, we have a trade-off: the accuracy of the search data versus the timeliness of the mission completion. This example illustrates the

Schedule	Makespan	Penalty
(1 2 3 4 5 6 7)	23	0
(1 2 4 3 5 6 7)	20	4
(1 2 3 4 6 7 5)	19	5
(1 2 4 3 6 7 5)	16	9
(1 2 5 6 7 4 3)	15	13
(1 2 3 6 7 4 5)	14	14
(1 2 6 7 4 3 5)	11	18
(2 3 1 6 7 4 5)	10	24
(1 6 7 4 3 2 5)	9	26
(2 6 7 4 3 1 5)	7	28
(6 7 4 3 1 2 5)	5	36

Table 4.3: Frontier Points For Sample Problem

trade-off that must be made between logical consistency and temporal consistency in the context of this restricted model.

Other applications can also benefit from this model, including the other examples that have been used throughout this work. The basic assumption is that the predecessor tasks have no relevance once successor tasks have been executed. This is often the case, as it is in the above example. Clearly, it is unnecessary to initialize the sensor once the search is complete. It is unnecessary to drill the pilot hole once the finish hole has been drilled. It is unnecessary to read a database item once the update has committed. In each case, the time to perform the complete set of tasks is less since one or more tasks are deleted. However, by deleting a preliminary task, the final goal is met with less accuracy, less precision, or with greater risk.

The dynamic programming algorithm we have formulated to solve this problem provides an exact solution to the problem. Since the knapsack problem is known to be NP-hard, we cannot guarantee that this algorithm will execute in a reasonable time. However, since dynamic programming executes in pseudo-polynomial time thus, it generally executes faster than enumeration. [Win93].

The frontier calculated by dynamic programming must agree with the frontier calculated by enumerative search. This does not imply however, that the same schedules are found by each. Since enumerative search finds all schedules, it therefore finds all schedules on the frontier. Dynamic programming by contrast finds only the points on (or close to) the frontier. Associated with each point found, one or more schedules can easily be constructed, however, finding all schedules associated with a single frontier point is not guaranteed by the dynamic programming algorithm. Of course, in practice, only one schedule is necessary, since all schedules with the same logical and temporal performance are considered equally useful and interchangeable.

Chapter 5

Independent Transactions with Deadlines

In the previous model of Chapter 4 we schedule a set of tasks to meet an overall deadline or makespan. In this chapter, we consider a more general problem, where tasks are modeled as transactions. We assume a model that allows deadlines on transactions and some precedence constraints within the tasks that make up a transaction. However, this model does not allow any precedence constraints between transactions. The restrictions that this model places on the general model presented in Chapter 3 are listed below.

5.1 Restrictions to the General Model

1. **Temporal penalty functions express a single deadline only.** The measure of temporal consistency for this model is the *maximum tardiness* of tasks. Max tardiness is found by computing the tardiness for each task; that is the amount of time each task is late in meeting its deadline. If a task meets, or is early for its deadline, its tardiness is zero. The maximum of the tardiness for all tasks is the *max tardiness* (T_{max}) for the schedule. This measure was chosen because an earliest deadline due (EDD) priority assignment scheme is known to

be optimal for minimizing maximum tardiness. Accordingly, penalty functions are restricted to a unit step function: penalty 0 prior to (or at) deadline and value 1 after. The penalty function models the timing constraint as a single task deadline.

2. **Only the final task in a transaction has a deadline.** While transactions are made up of several tasks, only the final task of the transaction has a deadline. This equates to a total deadline on the entire transaction, which is often the case in many real-time database applications.

3. **Precedence orderings can only be expressed between the deadline task and the non-deadline tasks within a transaction.** Furthermore, a non-deadline task may only be expressed as a predecessor to a single deadline task. This restriction enforces independence among transactions. Thus, there are no precedence constraints between tasks from different transactions.

Transaction tasks consists of a final task—the commit task—and a set of predecessor tasks. Unlike the model of Chapter 4, it is not necessary to delete tasks that violate precedence constraints however. If a predecessor task is placed at the end of the schedule, it has no effect on maximum tardiness.

By permitting deadlines on individual transactions, this model brings us closer to the more general model than the previous restricted model of Chapter 4. Consider several transactions, each with an independent deadline. Each transaction is composed of one or more tasks, and these tasks can interleave in such a way that maximum tardiness is minimized. When there is not enough time to execute all tasks, low value predecessor tasks can be moved to the end of the schedule so that transactions can meet deadlines—but at the cost of executing the predecessor task out of order. Alternatively, transaction deadlines can be missed to allow some or all predecessor tasks to execute.

5.2 Formal Model

Formally, the model consists of a set of partially ordered tasks, $1, 2, \ldots N$ where each task $i = 1, 2, \ldots N$ has an associated processing time t_i. There is a subset of these tasks, D, representing the *commit tasks* where the number of tasks in D is $M \leq N$. Each task $i \in D$ has an associated deadline, d_i. Another subset of tasks, P_i represents the set of predecessor tasks to task $i \in D$. For each task $j \in P_i$, there is a penalty ξ_{ji} for scheduling task j before task i. Also associated with each task $j \in P_i$ is an indicator variable x_{ij} which takes the value 0 if i is scheduled before j and 1 otherwise.

The set of deadline tasks D and the predecessor task sets P_i are disjoint. That is we have:

$$D \cap P_i = \emptyset, \quad \forall i \in D$$

Also, no task is a predecessor of more than one deadline task:

$$P_i \cap P_j = \emptyset, \quad \forall i \neq j$$

Finally, we assume that the tasks of D are ordered by their deadlines, that is $d_i < d_j$ if $i < j$.

Let c_i be the completion time of task i. Let d_j be the deadline of task $j \in D$. For any schedule we measure the maximum tardiness: $T_{max} = \max\{0, (c_i - d_i)\} \;\; \forall i \in D$.

Given a maximum tardiness, (T_{max}), the problem now is the determination of a sequence which minimizes the total penalty. This problem can be formulated as:

$$\min \sum_{i \in P} \sum_{j \in D} x_{ij} \xi_{ij} \tag{5.1}$$

subject to:

$$c_{i-1} + t_i < d_i + T_{max}, \quad \forall i \in D \tag{5.3}$$

$$c_0 = 0 \tag{5.4}$$

$$G_n = c_n - c_{n-1} - t_n, \quad \forall n \in D \tag{5.5}$$

$$\sum_{k=1}^{N} G_k \geq \sum_{k=1}^{N} \sum_{i \in P_k} t_i(1 - x_{ik}) \tag{5.6}$$

$$x_{ij} \in \{0, 1\} \tag{5.7}$$

The expression (5.3), constrains the completion time of each task based on EDD ordering and some T_{max}. Constraint (5.5) computes the size of the gap that immediately precedes each deadline task. Constraint (5.6) accounts for the total time required by tasks to fill each gap (G_N). That is, each gap k prior to deadline task k can be filled with only as many tasks that fit, and furthermore those tasks must be predecessors to task k or greater. This model is an integer programming formulation. A minimum value of T_{max} can be achieved by scheduling no tasks prior to their successor tasks—that is, the EDD schedule of deadline tasks alone. Optimally scheduling the predecessor tasks in the gaps prior to their successor tasks reduces the penalty for precedence violation, and if T_{max} is not allowed to increase, the first frontier point can be found. Increasing T_{max} allows additional tasks to be scheduled. When all tasks are scheduled to meet precedence constraints the maximum value of T_{max} is achieved and the last point on the frontier is obtained.

5.3 Heuristic Solution

We first consider the commit tasks ($i \in D$) alone. These tasks must be scheduled and cannot be deleted. Since the commit tasks each have a deadline and we are using max tardiness as the performance measure, an EDD sequence will yield the best schedule. This first scheduling attempt gives us the value of max tardiness, T_{max}, which we can then use a basis for the next schedules. We schedule each deadline task to complete late as possible, but no later than the task deadline plus T_{max}. This results in a schedule with the minimum value of Max Tardiness, but with "gaps" between tasks such that predecessor tasks can execute. By placing tasks in these "gaps" such as to minimize precedence violation penalties, we can find a set of schedules.

This problem can be viewed as an embedded knapsack problem where the tasks of each distinct transaction must fit into a time frame delimited by the deadline of the transaction. Each of the gaps described above must be filled with tasks that are predecessor to the commit task that ends the current or later gap. A heuristic to solve this problem is an extension of the heuristic to solve the Makespan Problem of Chapter 4.

The heuristic is as follows:

- Step 1: First schedule all deadline tasks (that is, the final task of each transaction) such that they complete at their deadline. If the schedule contains a conflict between any two tasks, that is two tasks overlap in time, then schedule the task with the earlier deadline just prior to the conflicting task. This procedure must be repeated for each conflict, starting with the last task in the schedule and working to the first.

- Step 2: It is possible that, after Step 1, the first task will be scheduled before the starting time (time = 0). If this is the case, then add an equal amount of time to each task's start time such that the first task begin at time zero. This step will fix the lower bound of T_{max} for the frontier.

- Step 3: As a result of Steps 1 and 2, we have all the final tasks of each transaction scheduled with possible time gaps between tasks. We also have a set of tasks, without deadlines, but with precedence constraints, that must be scheduled. Starting with the final task N, consider the gap just prior to this task (G_N). It can be filled (without penalty) only with tasks for which task N is the final transaction task, that is tasks (j) with the precedence relationship $j < N$. Using the dynamic programming algorithm of the Makespan problem described in §4.2, fill the gap prior to task N with predecessor tasks of N.

- Step 4: For any gap (G_i) just prior to task i, fill G_i with tasks (j) with precedence relationships of the form $j < i, j < (i+1), \ldots j < N$, that is all tasks (j) which can be scheduled without violating their precedence constraints. This gap is filled using the same procedure as described in the previous step.

- Step 5: Repeat step 4 for all G_i for decrementing values of i until the first gap is filled $i = 1$. At this point, t_{max} is still at the minimum, which is the best schedule in the measure of temporal consistency. However, it is the worst schedule in logical consistency.

- Step 6: To create the frontier of solutions, consider the gap prior to task 1, G_1. Increase the size of this gap by one unit and attempt to again fill it with any unscheduled task. This increases T_{max} by one and reduces the penalty if any additional tasks can be scheduled. Repeat this step, by repeatedly increasing the gap by one until all tasks are scheduled. In a manner similar to the makespan problem of Chapter 4, a frontier is defined. This frontier describes the trade-off between temporal and logical consistency in the measures of T_{max} and penalty (see §3.2.5).

While our solution to the first problem in Chapter 4 generates a set of good solutions, this procedure does not make such a guarantee. Instead, this solution provides a reasonable solution in a reasonable amount of time. Not only does this procedure not produce an optimal solution, but it is not possible to place a bound on the error. To validate this procedure, we instead use simulation.

5.4 Simulation

The heuristic has been simulated for a large number of randomly generated problem sets of approximately 100 tasks each. Each task set was generated randomly, but such that an optimal schedule exists, that is, a schedule with both $T_{max} = 0$ and $penalty = 0$. For each case then, it is possible that the heuristic could find the optimal solution. Knowing that a single optimal solution does exist gives us a way to assess the performance of the heuristic. Simulation results show that if the task sizes are small, then optimal solutions are likely to be found. If task sizes are large, then it is less likely that an optimal schedule can be found.

Each simulation run consisted of 200 problem sets generated. In creating a schedule the attributes of each task were assigned a value chosen randomly. The task attributes are listed in Table 5.1. In the table, the notation U($a - b$), means that the random values for this attribute are distributed uniformly from a to b. F(c) means that this attribute is a fixed value of c. Two sets of tasks were created, *deadline* tasks and *predecessor* tasks. To create a task set, deadline tasks where scheduled by EDD. Task deadlines were increased if necessary so that the schedule was feasible. Predecessor tasks with random processing times were then created to fit in the gaps between deadline tasks, such that the gaps were completely filled. This requires that the time of the final task of a predecessor set might be truncated so as not to exceed the time available. Finally, each predecessor task was randomly associated with a deadline task such that no predecessor task completed after its deadline task.

The number of predecessor tasks depends both on the processing time assigned to these tasks and the amount of processing time available for them. As mentioned, the schedule is generated such that all tasks meet deadlines and precedence constraints. For each task set generated, the heuristic computes a schedule. The frontier for the test set, as generated, is a single point, $(0, 0)$. The heuristic generates a frontier

Task Attribute	**Distribution**
Maximum total processing time (sec)	F(100)
Number of Deadlines	U(2 – 11)
Task Deadline	U(1 – 100)
Deadline Task Processing Time	U(1 – 6)
Predecessor Task Processing Time	U(1 – 7)
Penalty for Precedence Violation (each predecessor task)	U(1 – 11)

Table 5.1: Task Set Attributes

of one or more points. The error is calculated as the minimum distance from any frontier point to the optimal $(0, 0)$. For simplicity, the distance between two points $(x_1, y_1), (x_2, y_2)$ was taken as the rectilinear distance, $D = |x_1 - x_2| + |y_1 - y_2|$. As is usually the case, the results of the simulation are only valid for applications that have a similar probability distribution of task attributes. In addition, we assume that performance of the heuristic is not biased by the fact that an optimal solution does exist for each problem set. Indeed the heuristic does not use the existence of an optimal solution in any way, and furthermore we require the knowledge of an optimal solution to measure the heuristic performance.

Twenty-five simulation runs of 200 problem sets each were performed. For each run, task size, T, was varied from 1 to 25. Figure 5.1 shows the results for five of these runs. Each line represents the distribution of errors, plotted as the number of schedules with a given error for each value of error. The plots show that the heuristic generally performs such that in most cases the error is zero or very small. Figure 5.2 shows the effect of varying the task processing time. The average error was calculated for each run, and this is plotted for maximum task size (T). From this plot is clear that the heuristic performs best with small task sizes. This is because it is easier to "fit" many small tasks into a limited space, than one or two large tasks.

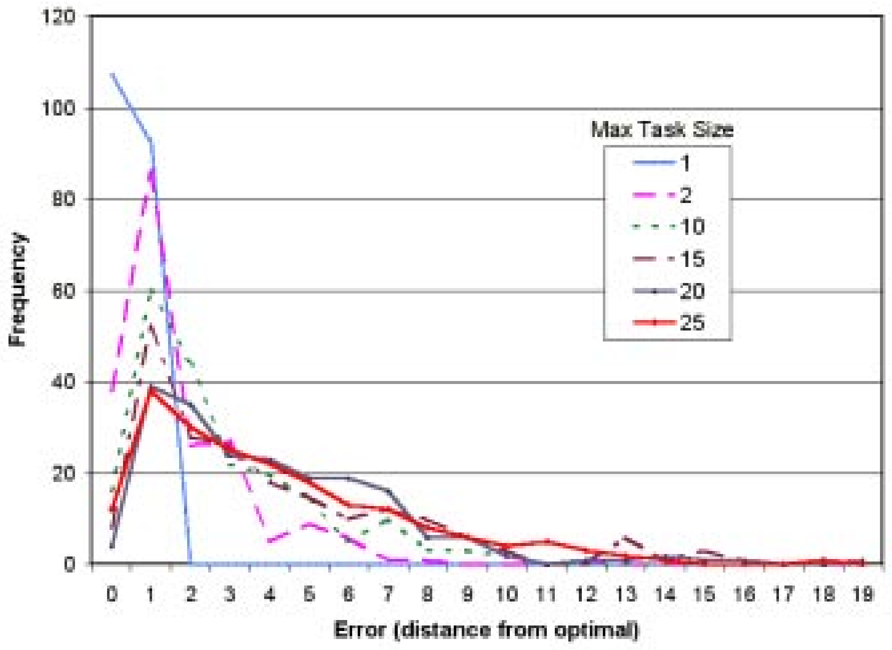

Figure 5.1: Histogram to Show Heuristic Performance

Again, note that while this is not the full general model of real-time database scheduling and concurrency control, applications exist for which this restricted model is sufficient. Recall the air traffic control example of §1.1.1. Consider an air traffic control display, where a single transaction results in the display of a single aircraft. Each transaction has a deadline to insure the timely display of information, and each transaction has a number of tasks which precede the final display task. If there is not sufficient time to complete the display, a predecessor task, such as tracker processing, could be scheduled after the display task. This results in the tracker process being moot and thus the calculation of aircraft position must rely on old heading and speed data. However, the display transaction may now complete by its deadline. Alternatively, executing the tracker task would result in a more accurate,

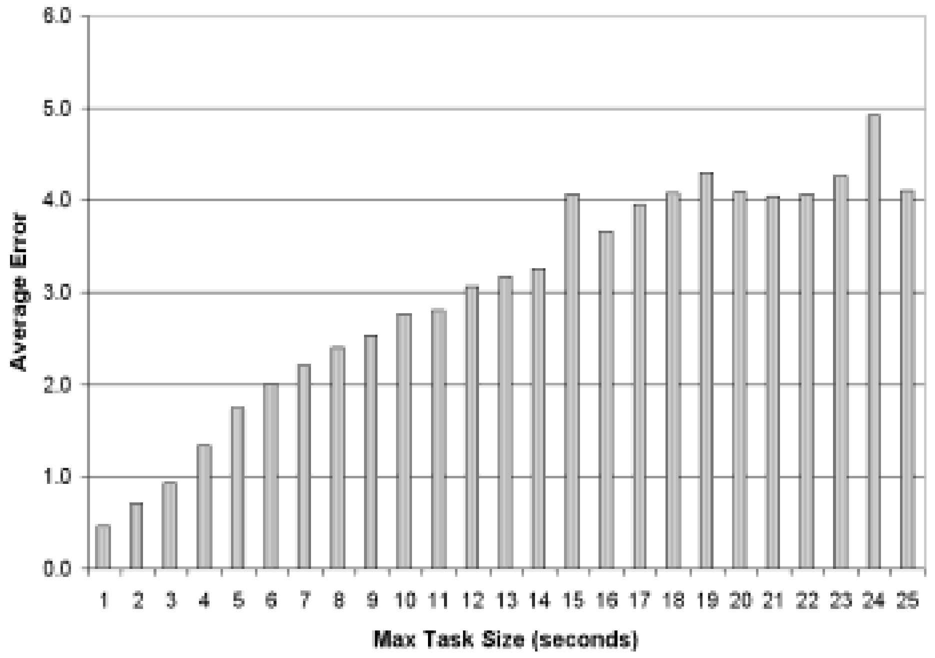

Figure 5.2: Simulation Results: Task Size vs. Heuristic Error

but late display. In the case of overload, where more aircraft are present in the sector or aircraft are clustered closer together than normally expected, some tracker tasks can be sacrificed in order to maintain an up-to-date, but less accurate, display.

Chapter 6

Tasks with Multiple Deadlines

As an extension of the previous problems, we now consider the problem where timing constraints are given by several deadlines rather than a single deadline. This is a discrete version of the problem considered by Locke. As discussed in Chapter 2, Locke uses a *value function* to express the value to the system of completing a task at some point in time. Locke permits a general function of time, but then reduces this to two basic parameters: a *critical time*, which can be considered a deadline, and a *value density* which characterizes the slope of the function after the critical time. Using these two parameters to specify each task, a schedule is created using an EDD (Earliest Deadline Due) algorithm and then deleting tasks with low value density until all remaining tasks are completed by their critical time. The deleted tasks are then placed at the end of the schedule.

The special case with two deadlines has been considered in Hariri and Potts [HP94]. Hariri and Potts assess a penalty for completing the task after the first deadline but prior to the second. All tasks must complete by the second deadline for the schedule to be feasible. The algorithm proposed by Hariri and Potts assumes that a feasible schedule exists. We generalize the problem addressed by Hariri and Potts to include additional deadlines for each task. In addition, we allow, but do not require a final deadline for feasibility. However, imposing a final deadline makes the search for a solution faster by eliminating infeasible solutions and thus greatly reducing the search space. Thus, a final deadline is recommended when appropriate to the application. The penalty function proposed here represents a somewhat more restricted implementation of the value function proposed by Locke. While this would seem to be a disadvantage, the multiple deadline function can be considered a piece-wise linear function representing the continuous value function, albeit in inverted form.

In previous chapters, we have introduced scheduling problems that include both temporal and logical constraints. These problems are characterized by rather substantial restrictions that limit their utility in some applications, but allow the problems to be more easily solved. This previous work has provided the necessary foundation on which to build this final model. In past problems we have proposed both exact and approximate solution methods. Here too, we will suggest both a heuristic which provides a solution but without any guarantee of optimality and an exact solution but without any guarantee of execution time. In general, an optimal solution would always be preferred over the heuristic; however because of the nature of the problem, an optimal solution may take a very long time to find. Thus, given a limited amount of time, as in the case in real-time scheduling situations, or if problem size is large, a heuristic may be necessary to provide a useful solution in a reasonable amount of time. In §9.5 we suggest some strategies for using these two methods for practical applications.

The deadline function expresses the penalty that will result if a task misses one or more deadlines. Figure 6.1 shows typical penalty functions. Each task has zero or more deadlines, and for each deadline there is an associated penalty. We further assume that deadlines are non-negative and non-decreasing in time. We discuss the implications of these assumptions in §6.1.1. There is no bound on the magnitude of the penalty function. However, for practical considerations, a maximum value at which the task cannot be feasibly scheduled is defined. That is, if the final deadline is missed, the schedule is considered infeasible, and a sufficiently high penalty is assessed to denote this condition.

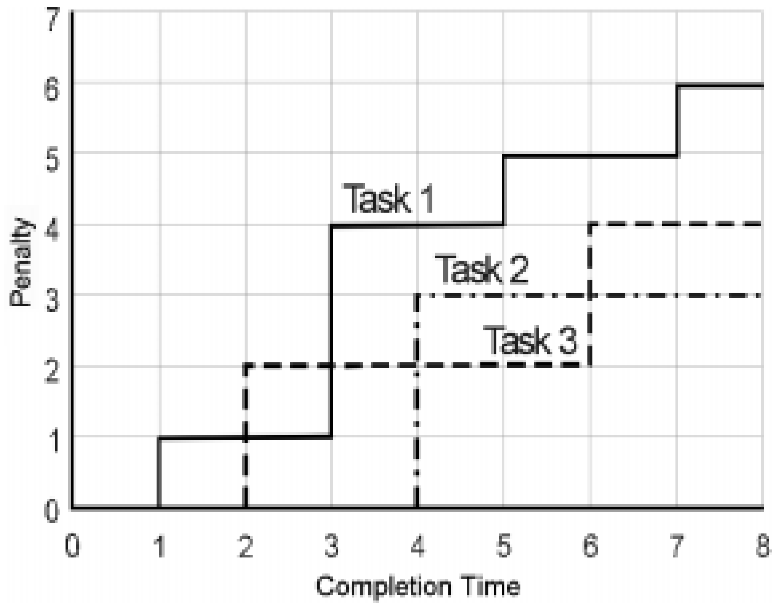

Figure 6.1: Typical Set of Penalty Functions

In this chapter we examine the problem of tasks with temporal constraints only, then expand it to include required precedence orderings in Chapter 7.

6.1 Temporal Constraints

To illustrate this scenario, we will use an example from scheduling tasks in FMS (Flexible Manufacturing System). An FMS can process several different jobs, which can be performed in batches with minimal setup time in between. Assume that associated with each job there is a deadline for completion of the batch. As suggested by Hariri and Potts [HP94], there may in reality be more than one deadline. To see this, consider the shipping time to the customer. For example, parts can be shipped by truck at lost cost, but will require perhaps a week to arrive. This imposes a deadline on the production floor to complete the order. If however, the parts are shipped by air freight, a later deadline can be tolerated at the factory, but will require the additional shipping costs—the penalty. Other factors can also lead to additional deadlines such as other modes of shipment, a customer willingness (or unwillingness) to accept a late order, or perhaps contractual penalties for late delivery. While Hariri and Potts propose two deadlines, it is clear that there can be several deadlines, each with different penalties. In this example, we may have a deadline for shipping by truck and arriving at the customer on time, shipping by truck arriving one day late, shipping by air arriving on time, etc. In each case, we can anticipate a penalty, either in additional shipping costs or in penalty fees for late delivery.

6.1.1 Model

Consider a set of tasks $1, 2, \ldots, N$. Each task i has processing time t_i and a set of deadlines d_{ik}, for $k = 1, 2, \ldots B_i$. Associated with each d_{ik} is a penalty π_{ik}. We order tasks by deadline such that $d_{ik} < d_{i(k+1)}$ and impose a penalty π_{ik} if, for task i, deadline d_{ik} is missed, but deadline $d_{i(k+1)}$ is either met or does not exist. Furthermore, we assume that $\pi_{ik} \leq \pi_{i(k+1)}$. Forcing the penalties to be non-decreasing with deadline will insure that for any delay schedule there is a non-delay schedule that is at least as good. To simplicity the model, we have assumed that all tasks have the same number of deadlines. However, the number of deadlines for any task can be effectively reduced by specifying successive deadlines with the same penalty. The problem of determining the best schedule that meets all constraints can then be represented by the following formulation.

Define $L_{ik} = 1$ if task i completes prior to the k^{th} deadline d_{ik}, and $I_{ij} = 1$ if task i is the immediate predecessor to task j and zero otherwise. Assume M to be an arbitrarily large number. Then the objective is:

$$Min \sum_{i=1}^{N} \sum_{k=1}^{B_i} \pi_{ik} L_{ik}, \tag{6.1}$$

subject to the following constraints:

$$\sum_{\substack{i=1 \\ i \neq j}}^{N} I_{ij} = 1, \quad \forall\, j \tag{6.2}$$

$$\sum_{\substack{j=1 \\ i \neq j}}^{N} I_{ij} = 1, \quad \forall\, i \tag{6.3}$$

$$c_i - c_j + p_j < M(1 - I_{ij}), \quad \forall\, i \neq j, j \neq 1 \tag{6.4}$$

$$c_1 = 0 \tag{6.5}$$

$$c_i - \sum_{k=1}^{B_i} d_{ik} L_{ik} \leq 0, \quad \forall\, i \tag{6.6}$$

$$\sum_{k=1}^{B_i} L_{ik} = 1, \quad \forall\, i \tag{6.7}$$

$$I_{ij} \in \{0,1\}, L_{ik} \in \{0,1\} \tag{6.8}$$

The objective (6.1) is to minimize the total penalty incurred by the schedule by optimal selection of the decision variables I_{ij}. This problem is an instance of the TSP (Traveling Salesman Problem), and can be formulated as an assignment problem with additional constraints to eliminate subtours.

The TSP involves the construction of a *tour*, where each city is visited once and only once. The tour begins and ends at the same city, that is, the tour is a complete cycle. Since for our scheduling problem, we need a sequence and not a cycle, a *dummy task* ($task_1$) is added with no processing time and no deadlines. Without increasing the cost of the schedule, this task is used to complete the tour.

Equation (6.2) forces each task to be assigned a unique position in the sequence, and equation (6.3) ensures each position is assigned to a unique task. This is the basic assignment problem. Inequality (6.4) requires that each task's completion time be greater than the predecessor task's completion time by at least the processing time of the predecessor task. The dummy variables, c_i, c_j, also eliminate sub-tours by requiring that each task in the sequence, except the dummy task ($task_1$), has a completion time greater than the task scheduled prior to it. Since there is one task in a subtour that must be both before and after all other tasks in the subtour, subtours which do not include $task_1$ are eliminated. Therefore $task_1$ is the only task to complete a tour; there can be no other tours. The usual sub-tour elimination constraint for the TSP [Win93][LLKS85] is: $u_i - u_j + NI_{ij} \leq N - 1$ which is mathematically equivalent to inequality (6.4) (and precisely equivalent if all task processing times: $p_i = 1$).

Finally, equation (6.5) initializes the cumulative processing time to zero. Inequality (6.6) and equation (6.7) are used to compute L_{ij} which indicates the deadline first missed by task i. The final constraints, equations (6.8) are used to restrict L_{ij} and I_{ij} to binary variables. All variables are assumed to be non-negative in this IP formulation.

6.1.2 Example

Consider the following example. A job shop has five jobs waiting to be completed in an FMS cell. Each job has a deadline required by the customer. However, as is often the case, the deadlines are not "hard" and may be missed, but with some penalty. As we have discussed earlier, the penalties may arise for several reasons, but whatever the source, the company wishes to complete all jobs such that penalties are minimized. Table 6.1 shows the processing times, required deadlines and corresponding penalties

Job	**Deadline (Penalty)**			**Processing Time**
1	27 (8)	32 (16)	34 (19)	8
2	12 (4)	15 (12)	20 (19)	5
3	13 (3)	18 (9)	18 (17)	4
4	12 (4)	13 (11)	16 (16)	3

Table 6.1: Example Job Deadlines

for each job.

Recall this problem is an instance of the Traveling Salesman Problem, which is known to be $\mathcal{NP}$-hard [LLKS85]. There are a number of strategies for solving this problem; several will be discussed later. One possibility is to use one of several commercial software packages. Using LINGO [lin94b] small instances of the TSP problem can be formulated and solved in a reasonable amount of time. For the TSP problem, LINGO can be reasonably used to provide an exact solution for sets of up to ten tasks. Using LINGO, an optimal solution to the problem defined in Table 6.1 is the task sequence: $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$, which has an objective value of 19. Actually there are 6 optimal sequences out of $4! = 24$ possible sequences. However, we require only one solution and thus LINGO provides a baseline method to solve this problem. We will use LINGO to judge the performance of the heuristics discussed below. Of course, this example problem, with only 4 tasks, is very small. The simulation results discussed in §6.4 refer to problems sets of size 10 to 20 tasks.

The processing time required to solve this problem is an issue as the complexity of the TSP and this problem is $O(n!)$, and therefore solution time, is proportional to $n!$. For this problem of four tasks the processing time is minimal. Figure 6.2

shows the processing times for larger problems. As expected, the processing times increase quickly with problem size, making LINGO unacceptable as a means for solving anything but the smallest problems.

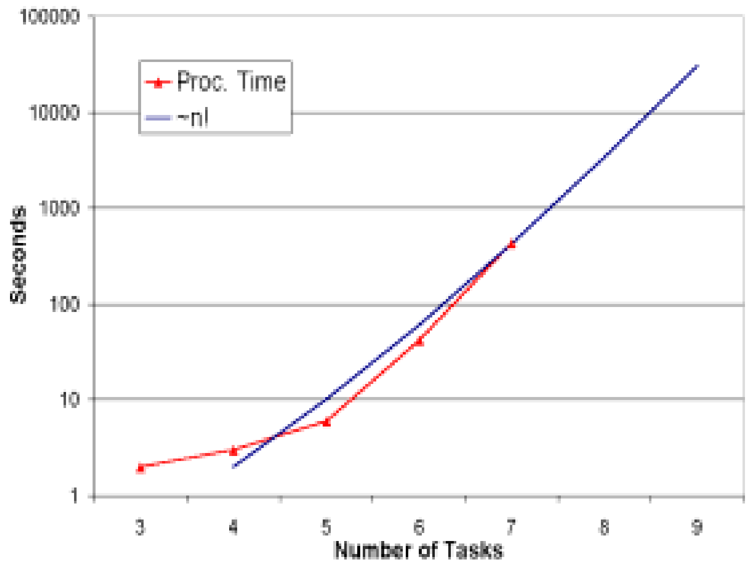

Figure 6.2: Processing Times for LINGO Solutions

6.2 Branch and Bound

LINGO and similar programs find a solution to the formulated problem by relaxing integer constraints, that is, the requirement that some variables assume integer values is relaxed. The relaxed problem is then solved by LP (Linear Programming), which finds a solution relatively quickly. A branch and bound technique is applied to find integer solutions in the vicinity the LP solution. We investigate the use a branch and bound solution for this problem, but where the search is conducted more efficiently by using characteristics specific to this problem. This results in a faster algorithm for this special case. Branch and bound is a tree search and this was implemented as a depth first search (DFS) of the problem space. The DFS provides an exact solution, however without appropriate methods to reduce or *prune* the search space, it is computationally expensive, being of complexity O(n!). Bounds are used to prune the search tree so that all nodes need not be explored.

In any implementation of branch and bound, several issues must be considered in order to yield an effective implementation [LLKS85]. These include:

1. The order in which nodes are considered (branching rule).

2. Initial candidate solution (upper bound).

3. The estimate for the objective at any node (lower bound).

4. Pruning Rule.

5. Termination criteria.

The specific form of these rules is application dependent and is discussed below for this problem.

6.2.1 Branching Rule

The branching rule specifies the next node to be searched from the node currently being explored. There are two basic branching rules, Depth First Search (DFS) and Breadth First Search (BFS). DFS is easily implemented using a simple recursive algorithm. That is, from any parent node, each child node is chosen in a set sequence, and the DFS is performed on that node. When all child nodes have been explored, control is returned to the calling node. DFS also has a major advantage in that it requires that very little information be stored in the course of the search. Only the incumbent solution and some information related to the state of calling procedure (i.e.

the number of the child node currently being explored at each level of the recursion) must be saved. The branching rule based on BFS may offer some advantages in pruning since it can compare all partial solutions at the current level. However, BFS requires a more complicated algorithm for branching, and also requires a very large amount of storage. At each stage, information for all nodes must be stored. At the lower levels of large problems, the storage requirements can become huge. For example, ignoring the benefits of pruning, only twelve tasks will require storage of several hundred megabytes of data.

We use DFS, but enhance it with several other strategies in choosing the next node to explore. Recall that we have assumed the deadline penalty function used is non-decreasing. For some tasks, the penalty for being processed last in the sequence can be quite high. This leads to the observation that some tasks placed last in the schedule will have rather high penalty values, and likely create a poor (i.e. high penalty) schedule. Since tasks tend to have a low value of penalty if scheduled early in the sequence, any task can be successfully scheduled at the beginning of the schedule. Indeed, if we choose *any* task for the first position in the sequence, it must have a lower penalty than the candidate (upper bound) solution since that solution contains all tasks. Clearly, any task placed first in the sequence must have a penalty lower than the incumbent schedule. Alternatively, consider choosing a task for the last position in the sequence. By the assumption of non-decreasing penalty functions, that task *alone* may have a penalty greater than the incumbent solution. Thus, that node and all its child nodes can be pruned. By scheduling tasks from last position to first, large portions of the search space can be pruned early in the search process. Pruning nodes early in the search has the greatest impact on processing time, since a node pruned near the top of the tree eliminates all nodes below it from consideration.

Using a DFS during testing, a task set of less than ten can be searched in a short amount of time on a typical personal computer. However, adding just a few tasks requires a computer with a speed several orders of magnitude faster to process the larger task set in the same amount of time. The spectacular growth in the search space due to *combinatorial explosion* makes problems with just a few more tasks unsolvable in a reasonable amount of time.

Prior to initiating the DFS search, we arrange the tasks according to a heuristic that ranks tasks with respect to an estimate of importance in the contribution to the objective function. This heuristic is described in detail in §6.3. The DFS implementation performs the search on *objects*. In our case the objects are task numbers, however, any task can be assigned to any object. Barring any reason *not* to do so, the first solution to be explored might be $1 \rightarrow 2 \rightarrow \ldots \rightarrow n$, that is, tasks are assigned to objects in numerical order. But tasks need not be assigned in this order, and in fact, we assign tasks in the order of the candidate solution provided by the heuristic. We "prime" the branch and bound algorithm with this solution so that the branching rule at any node tracks the sequence suggested by the heuristic. The implications of this pre-ordering are further explored in §6.2.2. This concept can be extended to recalculate a candidate solution at any node, and again "prime" the algorithm with this value. Testing indicates that the additional time required for the re-calculation is less than the gain it provides, although this is implementation specific. In our testing, it became clear that the heuristic solution sequence at some lower level node was not very different than the initial sequence provided by the heuristic. The small benefit of recalculating the sequence at each node was outweighed by the additional time to do so.

6.2.2 Upper bound

For this problem, the calculation of the upper bound is not nearly as critical as the lower bound. In fact, a separate calculation is not required since the upper bound is calculated in the process of examining nodes. Once a branch is completely explored down to the leaf node, an upper bound is known. The solution becomes the upper bound if it is better than the current upper bound. The upper bound is used in fathoming nodes, which will be discussed in the subsequent sections. To get an initial bound, we can simply explore a branch (any branch) fully. This yields an upper bound, although a not necessarily a very good one. In fact, we use a heuristic to obtain the initial upper bound. The heuristic is discussed in §6.3.

As more and more nodes are explored, we will continue to find better upper bounds. The better upper bound we can obtain, the more efficiently we can prune non-productive branches. This is most evident at the root of the search tree.

6.2.3 Lower Bound

An essential element of branch and bound is the determination of which nodes should be pruned. Nodes are pruned when the penalty accumulated at the node; in addition, an estimate of the penalty for the remainder of the schedule (nodes) is greater than the upper bound. At any node, the penalty accumulated thus far is a simple calculation since the completion time of each scheduled task is known. Thus, it is the calculation of lower bound that determines the effectiveness of pruning. If the lower bound is a weak lower bound, that is, it is too low, then branches that will not lead to a solution will continue to be explored. The closer the lower bound is to the actual solution, the more efficient the search is likely to be.

The calculation of a lower bound is dependent on the structure of the problem. For the problem discussed here, we propose three methods of calculating the lower bound. All three take advantage of the characteristics of this problem.

- **Method 1.** A lower bound can be calculated by aggregating the minimum penalty that may be incurred by each unscheduled task. The minimum penalty for each task is the penalty it would incur if it were to be scheduled first. Since no ordering of tasks could result in a lower penalty, this is indeed a lower bound. However, since only one task will be scheduled first, the actual penalty could be much higher. An advantage of this calculation is that it is easily executed.

- **Method 2.** A second calculation for the lower bound is based on a refinement of the scheme used above. Instead of assuming that all tasks begin execution at the beginning of the schedule, we use a sequence of tasks which allows us to calculate a set of completion times. Assuming for a moment that all task penalty functions are identical, the sequence of tasks that will result in the lowest penalty is the sequence where shortest tasks are scheduled first. This allows tasks to complete earlier in their penalty functions. Hence, this sequence produces a list of most optimistic completion times. Since we do not yet know the order in which tasks will complete, we simply use the minimum penalty over all tasks for each completion time. The total penalty is the sum of the minimum penalties for all completion times. This is a lower bound since no ordering of tasks could result in earlier completion times and no association of penalty function to task completion time could result in an lower penalty. Depending on the actual penalty functions, the same task penalty function may be used several times, resulting a weaker lower bound. This method provides a lower bound that may, or may not, be better than the first method. However, it takes more time to compute.

- **Method 3.** A stronger lower bound can also be calculated by approximating the TSP by an assignment problem. Both the previous methods make poor

Task	Processing Time
1	3
2	2
3	1
4	2

Table 6.2: Unscheduled Tasks for Branch and Bound

assumptions, which we know cannot be true, resulting in a weak lower bound in either case. For this method, we assume the same *SPT* (shortest processing time first) calculation of completion times of the second method. However, instead of using the minimum possible penalty, we associate a single penalty function with a single task such that total penalty is minimized. This association of penalties to tasks is solved as the assignment problem (AP). The assignment problem is an integer problem that is solvable as a linear problem, and thus it is solvable in a relatively short amount of time. In practice, this third method performs the best when calculating the lower bound. However, the gains in pruning more nodes may be offset by the additional time required for calculating the AP solution.

As an illustration of the third method, assume a set of four unscheduled tasks, with associated processing times, as shown in Table 6.2. Figure 6.3 diagrams the penalty functions.

We seek a lower bound on the penalty for scheduling the tasks. The exact solution of the lower bound is simply a subset of the larger problem, the TSP, so clearly we do not expect to find an exact solution (if we could, we would have no need for a lower bound). We seek to solve instead, a simpler problem that will give us a lower bound. We can calculate best case completion times by arranging the tasks in order of smallest processing time first. This ordering is tasks: $3 \rightarrow 4 \rightarrow 2 \rightarrow 1$,

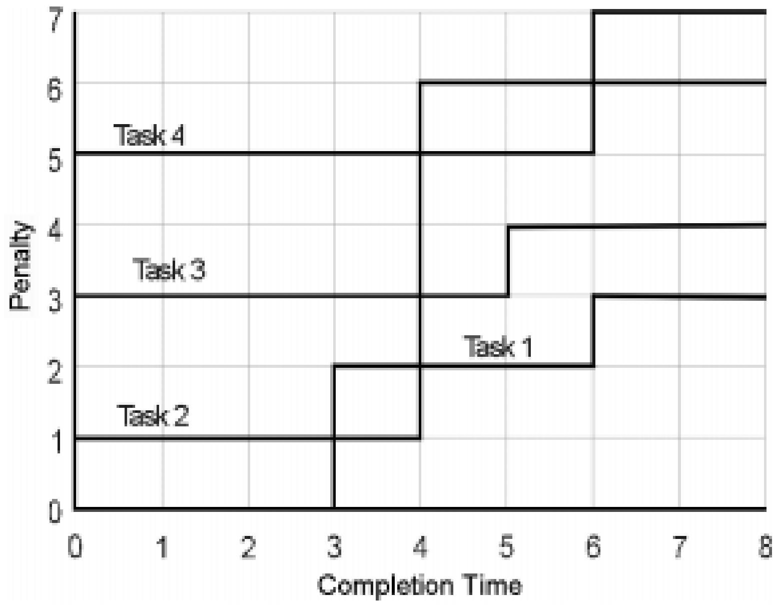

Figure 6.3: Example Penalty Functions for Lower Bound Calculation

and the corresponding completion times are 1, 3, 5, 8 seconds. We can associate, with each completion time and task, a penalty should the task complete at that time. This yields an 4×4 matrix where the penalties for all tasks are based on fixed completion times, independent of task ordering. Table 6.3 shows the resulting matrix for this example. Each entry is the penalty for task i completing at time t. It is an instance of the assignment problem to choose one unique completion time for each task such that total penalty is minimized. The assignment problem can be solved using LINGO, and for this example the assignments that result in the minimum penalty are enclosed in parenthesis in Table 6.3. The sum of the penalties for this assignment, 10, is a lower bound for the TSP. The actual minimum penalty for the TSP of this example is 12, was also obtained using LINGO.

	Completion Time			
Task	**1 sec**	**3 sec**	**5 sec**	**8 sec**
1	**(0)**	2	2	3
2	1	**(1)**	6	6
3	3	3	4	**(4)**
4	5	5	**(5)**	7

Table 6.3: Penalty Table for Lower Bound Calculation

6.2.4 Pruning Rule

With upper and lower bound defined, we are now in a position to formally define the rule used for pruning. This rule is important to this work, since variations of branch and bound that will be introduced in later chapters depend on modifications to this rule. Branches are pruned according to the following rule. Define the penalty at any node as Π_n. The upper bound on the optimal solution is UB. It can be determined by a heuristic solution initially and modified by better solutions found during the branch and bound process. The lower bound, LB, is the estimate of a solution to the end of a branch. We then prune if the following condition is met:

$$\Pi_n + LB > UB, \tag{6.9}$$

Because of the enormous number of nodes that must be considered in all but very small problems, pruning is necessary for solving this problem efficiently.

6.2.5 Termination Conditions

In the branch and bound algorithm, nodes are explored until they are all *fathomed*, that is all branches are either explored to the leaf node or discarded as non-productive. If, during the process, a solution is found that is better than the upper bound, this becomes the best solution and thus the new upper bound. When all nodes have been fathomed, the algorithm terminates with an optimal solution. The number of sub-optimal solutions found depends on the effectiveness of pruning since we seek to prune all nodes that will not lead to optimal solutions.

One of the problems with branch and bound is the time it may take to find an optimal solution. In the worst case, nearly all nodes might have to be explored—although this is unlikely to occur in practice. However, we can prune additional nodes by accepting a solution which is within some acceptable distance (ϵ) of the lower bound. This leads to an alternate *termination condition* for which we specify the distance from the lower bound desired, and then terminate when this is satisfied.

We define $\epsilon \geq 0$ to be the maximum difference between a lower bound and a terminating solution. Typically, ϵ is expressed as a percentage of the lower bound, however in the formulations to follow it is assumed that ϵ has been converted to an absolute value. We modify the pruning rule of equation (6.9) to prune a node only if it could lead to a solution that could not be better than the current upper bound by an amount ϵ.

As before, define the penalty at any node as Π_n, the upper bound UB, and the LB. We then prune if the following condition is met:

$$\Pi_n + LB + \epsilon > UB \text{ where } \epsilon \geq 0, \tag{6.10}$$

Using ϵ, the termination condition for the branch and bound algorithm is as follows. All nodes that cannot lead to a solution better than the upper bound by ϵ are pruned. The algorithm terminates after all nodes have been fathomed.

6.2.6 Branch and Bound Results

Since branch and bound produces an exact result, simulation is not required to determine its accuracy. However, we did perform several tests to determine the average time required by the algorithm to terminate. In calculating a lower bound, method 1 and method 2 (in §6.2.3) resulted in approximately the same number of nodes being pruned. Because this is so dependent on the specific task set used however, we did not try to test this hypothesis. Indeed, the result is easily controlled by the characteristics of the task set used. We did measure the processing time of method 1 alone, then the processing time for method 1 and method 2 together. Method 2 required more processing than method 1, so we found that the gains due to increased pruning where lost to the additional time to calculate the lower bound.

The branch and bound algorithm using method 3 (lower bound by the assignment problem) was found to be better in the number of nodes pruned. The performance gain, with respect to additional nodes pruned was determined through simulation. Executing the branch and bound algorithm, nodes were tested for pruning by method 1 and method 2, if the node did not meet the criteria for pruning, then the node was tested using method 3 for finding the lower bound. A count was kept of nodes that met the criteria for pruning using method 3, but did not meet the criteria of methods 1 and 2. Results showed that the lower bound using method 3 pruned an additional 0.015% over the other methods. A general algorithm was used to solve the assignment problem, and no effort was made to optimize the algorithm itself. Thus we do make any claim about performance with respect to processing time. This testing was primarily to show the effectiveness of using the assignment problem for computing the lower bound and the effect on state space reduction.

The time required to perform a branch and bound search is, of course, dependent on the number of tasks. For a simple enumerative search, we would expect the time to be non-polynomial and proportional to $O(n!)$. Through pruning, branch and bound should be significantly better. Simulation was used to measure the effectiveness of pruning, and as expected, branch and bound, on average, performed much better than enumerative search. The results are shown in Figure 6.4. The search time is less than $n!$ but greater than n^2 with respect to the number of tasks (n).

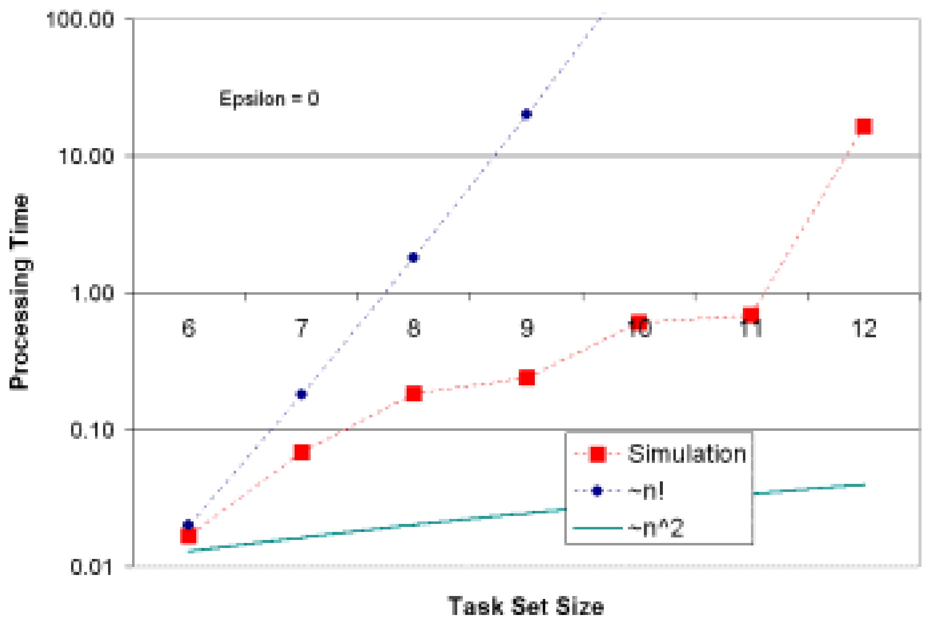

Figure 6.4: Branch and Bound Processing Time

The use of epsilon (ϵ) was used to reduce the search space, with the expectation that processing time would also decrease. Using ϵ equal to 2% of the optimal objective value, for example, decreased the search time by about 50% in a simulation of 100 problem sets. Similar results where found when ϵ was increased from 2% to 5% then to 10%. Beyond that, increasing ϵ to 20% resulted in a smaller improvement in processing time.

The branch and bound technique, as outlined above, is guaranteed to find the optimal solution. A limitation of the branch and bound is that the time required to find a solution is not a polynomial function of task set size. In the worst case, processing time is proportional to $n!$. However, by efficient pruning of nodes, we found that a solution for task sets of twenty tasks can be found in a short time. Pruning depends on the accurate calculation of lower bound. We have presented several methods of doing this, and observed that by solving the AP (assignment problem) more nodes are pruned. However, an efficient implementation of this method is required to compensate for the time required to compute a solution to the AP.

6.3 Heuristic

While integer programming and branch and bound provide a method of obtaining exact solutions, larger problems become difficult to solve in a reasonable amount of time. Regardless of the enhancements to branch and bound, at some point, the number of tasks simply becomes overwhelming. For larger problems it is often desirable to use a heuristic that provides a good solution, but in a greatly reduced amount of time.

The following heuristic has been developed for this problem. In each step of this algorithm, a task is selected to fill the last unassigned position in the sequence. There are several attributes of a task that would make it a candidate to be scheduled last. This can be more easily seen if we consider several simple cases of penalty functions which are illustrated in Figure 6.5.

Zero Slope. In this case, all tasks are equally suitable to be scheduled last. If all tasks have a penalty function which is constant for all completion times ("zero slope" shown in Figure 6.5), then all schedules will produce the same objective value. This is true regardless of the value of the penalty function. This is easily proven. Consider any arbitrary schedule of n tasks and compute the total penalty. Interchanging any two tasks has no effect on the total penalty, since the penalty for any task is a constant regardless of completion time.

Equal Slope, Same Processing Times. The above result can be extended to the case if all tasks have the same processing time and all tasks have a linear function with equal slopes as shown in Figure 6.5. The result is same as above. Choose any arbitrary schedule and interchange any two tasks. The increase in penalty for one task is exactly offset by the decrease in the other.

If, however, the processing time for tasks is not equal, then interchanging tasks will produce a different objective value.

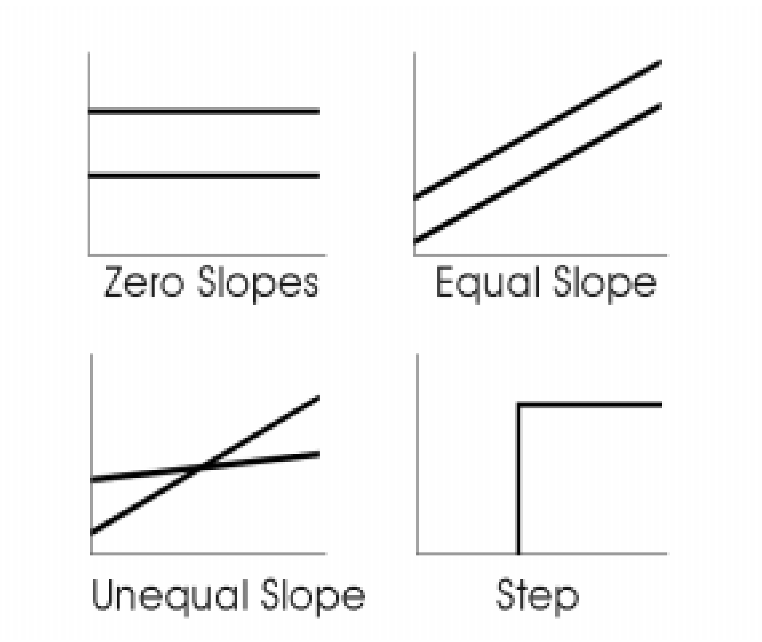

Figure 6.5: Simplified Penalty Functions

Equal Slope, Different Processing Times. In the case where the penalty functions have equal non-decreasing slopes (see Figure 6.5), then in the case where processing times are different, an optimal schedule can be produced by scheduling the task with the longest processing time last. As proof, consider any arbitrary schedule of tasks. Exchange any two adjacent tasks. The completion time of the task that occurs last does not change after the exchange. However, assuming the longer task is last prior to the exchange, the completion time of the task which occurs first will occur later after the exchange. Thus, the increase in penalty of the task now last is less than the decrease in penalty of the task now first. Any exchange of adjacent tasks that moves the shorter task to a position earlier in the schedule will decrease the total penalty.

Unequal Slopes, Same Processing Times. Consider linear penalty functions for tasks where the non-decreasing linear slope of the differs for each task, as shown in Figure 6.5. If all tasks have an equal processing time, then an optimal schedule can be formed by choosing the task with the lowest slope last. Again, choose an arbitrary schedule. Interchange any two tasks such that the slope of the earlier task is lower than the slope of the later task. Thus, the increase of the earlier task when moved to the later position will be less than the decrease of the later task when it is moved to the earlier position since moving a task with a smaller slope results in a smaller change in penalty. Moving the tasks with the lowest slopes to the end of the sequence will provide the optimal schedule.

Single Step. We now consider a penalty function which is a single *step* function, and again this is shown in Figure 6.5. The step function has a penalty value of zero prior to the deadline and a non-zero positive value at and after the deadline. Assume that all tasks have the same value after the deadline, but tasks may have different processing times. This model is identical to tasks with deadlines where the performance measure is the number of deadlines missed. Earliest deadline due (EDD) will create a schedule with zero penalty if such a schedule exists. If no schedule with zero penalty exists, then a different method is required. Schedule tasks choosing the task with the earliest deadline. If the deadline is met, continue by choosing the second task in the sequence from the remaining set of tasks with the earliest deadline. If the task exceeds its deadline, it is then scheduled last [Win93]. This process is continued until all tasks have been scheduled. If task penalty values after the deadline are not equal, then this problem is an instance of the knapsack problem.

The purpose of the above classes of penalty functions is to show that different characteristics suggest different scheduling criteria. Notice that tasks with discontinuous functions (the step function here) must be scheduled differently than tasks with continuous penalty functions although they appear to be similar to the continuous functions. While not all of these characteristics exactly apply to the multiple step function that used in this model, however, some of these characteristics can be inferred.

For the problem that we are considering here, the penalty functions may be combinations of the characteristics discussed above. The discrete penalty function that we propose can be relaxed to approximate a continuous function, which is accomplished by linear interpolation between the steps of the penalty function. Figure 6.6 shows how this is done. Such a relaxation has some benefit in scheduling tasks, but from the above discussion, it is clear that the interpolated function is not equivalent to the original. While no single approach using simplified functions applies, all can be used in a scheduling strategy, and we propose to use a combination of several key criteria. Of course, an optimal solution cannot be found in polynomial time as the problem as described in §6.1.1 is $\mathcal{NP}$-hard. Alternatively, any heuristic that executes in polynomial time cannot guarantee an optimal solution. Such is the case with our heuristic.

Combining these strategies, we present a composite heuristic. The schedule assigns tasks to a position, beginning with the last position in the sequence. A list

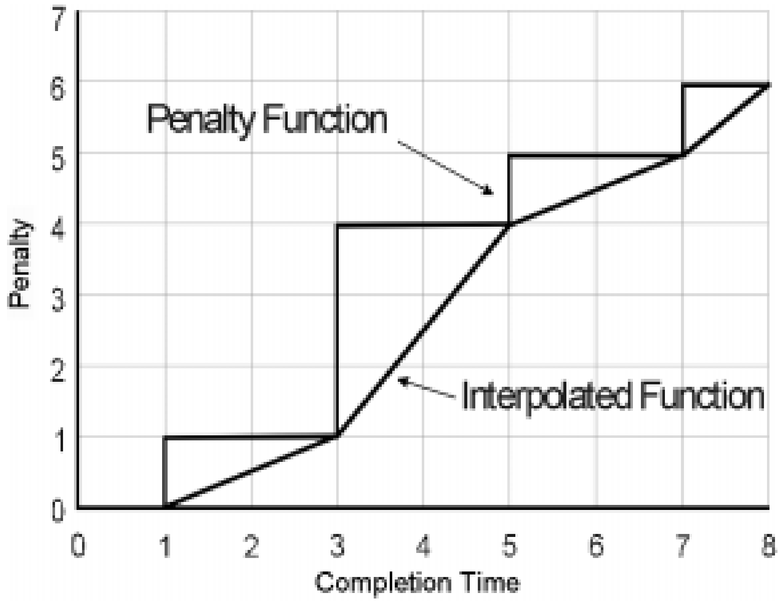

Figure 6.6: Interpolated Penalty Functions

is constructed by ranking all unscheduled tasks based on processing time, penalty function slope and value at the completion time for the position being scheduled. Referring to the model given in §6.1.1, we seek to find a sequence of tasks which minimizes the sum of the penalties for each task.

The heuristic creates a sequence by assigning to tasks to the last unassigned position in the sequence. In scheduling a task in the next unassigned position in the sequence, the heuristic uses the sum of three characteristics for each task. These characteristics are computed for the completion time of the position to be filled. Tasks are assigned values based on the following characteristics: shorter processing time, lower slope and lower penalty. In calculating the slope, the step function penalty function is replaced with a piecewise linear function such that breakpoints of the function are the penalty values at the deadlines of the original step function (recall

Figure 6.6). Using this new function, a slope can be calculated at the completion time of the position to be scheduled. Tasks are ranked based on the weighted sum of these three characteristics. The task with the best (lowest in this case) ranking is assigned to the position.

For this heuristic, we define several additional parameters. Let U be the set of unscheduled tasks and T the total processing time of tasks in U. Initially, U contains all tasks, that is $U = \{1 \ldots n\}$

Next, let P_{ic} denote the penalty of completing task i at completion time c. A "relaxed" penalty function P'_{ic} is also defined. This function returns a value of penalty that is a linear interpolation of penalty value for completion times that fall between deadlines. Again, refer to Figure 6.6 which shows both the actual penalty function and relaxed penalty function. P_{ic} is normalized by dividing it by the maximum value of P_{ic} over all unscheduled tasks at each completion time.

A "slope" parameter m_i for task i is used by the heuristic to consider the change in penalty for change in completion time. The slope is calculated as the change in relaxed penalty P'_{ic} between the current completion time, and the completion time less the task processing time. This difference is divided by the task processing time:

$$m_i = \frac{P'_{ic} - P'_{i(c-t_i)}}{t_i}.$$

It is intended to both capture the implied slope of the penalty function in the region of the current completion time and the task processing time. This is calculated for each task at each scheduling decision.

A schedule S is created by choosing a task from the unscheduled set, U. Initially, U contains all tasks, while S is empty. Also, begin with the last task, $n = N$. The following is the heuristic to create a schedule S:

- **Step 1.** To choose the task to be executed next, n, first compute the completion time as:

$$c_n = \sum_{i \in U} t_i$$

- **Step 2.** Rank tasks by the following metric:

$$V_i = \alpha P_{ic_n} + \beta S_{ic_n}$$

- **Step 3.** Choose task with lowest value of V_i, and place at beginning of sequence in S. Remove the task from U. Decrement n.

- **Step 4.** Repeat Steps 1 through 3 until U is empty.

The weighting factors, α and β are determined empirically. A more complete definition of this heuristic is presented in pseudocode form in Appendix A.

6.4 Computational Results

A number of simulations were run to determine the effectiveness of this heuristic. Task sets were randomly generated; the number of tasks the number of deadlines, the time of the deadlines, and the penalty associated with the deadline were all chosen using a random function. The composite heuristic was used to construct a schedule. For comparison, additional heuristics were also employed. These additional heuristics focused on one of the characteristics discussed above, for example "slope."

Random Task Sets. For each solution, a task set is required. A task set was randomly generated with the characteristics shown in Table 6.4. The bounds, for example U(6 – 18) are stored in a file, which allows them to be easily changed for specific tests. As in Table 5.1 the notation U(6 – 18), means that the random values for this attribute are distributed uniformly from 6 to 18.

Task Attribute	**Distribution**
Number of Tasks	U(6 – 18)
Number of Deadlines	U(2 – 6)
Task Processing Time	U(2 – 12)
Penalty Increment	U(2 – 9)
First Deadline before Completion Time	U(2 – 10)
Last Deadline after Completion Time	U(2 – 10)

Table 6.4: Random Distribution of Task Attributes

The penalty at each deadline was determined by adding the "penalty increment" to the penalty at the previous deadline (or add to zero for the first deadline). The times for deadlines are based on the completion time of the task and assume an arbitrary schedule (a random ordering of tasks). These times are expressed as the amount of time the first deadline occurs before the completion time, and the amount of time last penalty occurs after the completion time. Using these attributes and the random values generated, a unique problem set is generated for the scheduling simulation.

The flow diagram for the simulation (AP_BAT.EXE) is shown in Figure 6.7. The program actually spawns three programs, including LP.EXE, a program that generates an instance of a data set (TSP.LDT) from the specification file (TEST.SPC). A new test set is generated on each pass. The program LP.EXE then performs a branch and bound solution and runs four heuristics. The results are appended to the data set file (TSP.LDT). LINGO is the second program to run, it reads a model (TSP.LNG) along with the data (TSP.LDT). LINGO stores the solution objective and sequence in two files (OBJ.DAT and SOLU.DAT respectively). LINGO is not executed for tasks sets of greater than 10 tasks. Finally, the third program, STORE.EXE, is run. It reads results from the three data files: TSP.LDT, OBJ.DAT and SOLU.DAT. Data from these files is parsed and appended to a comma-delimited file (RESULTS.CSV). RESULTS.CSV is directly readable by the spreadsheet program, Excel[1]. Excel is used to analyze the data following the completion of the simulation. Parameters are passed to the high level program AP_BAT.EXE that specify the number of iterations, the components to be executed and several other parameters to control the simulation. Typically, our simulation test runs include at least 1000 iterations.

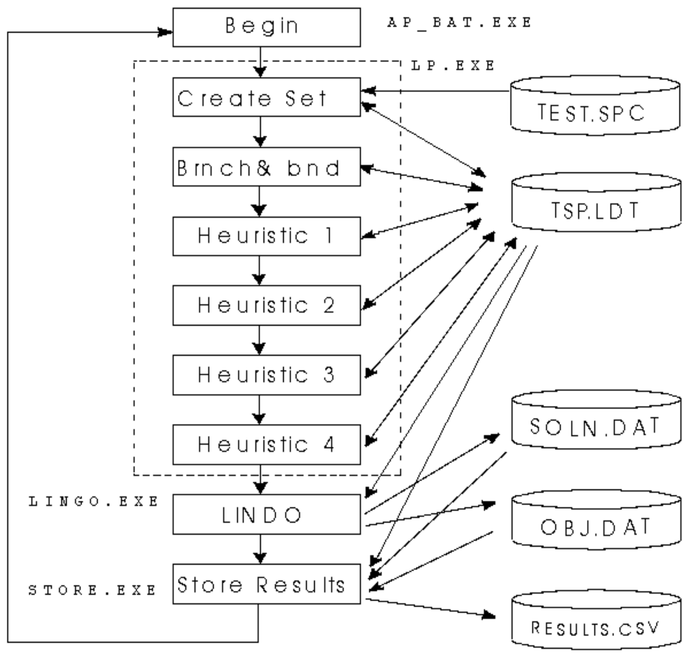

Figure 6.7: Simulation Flowchart

The randomly generated task set was processed by the branch and bound algorithm discussed above for an exact solution. To keep processing time reasonable, task sets were limited to fewer than twenty tasks.

For each task set randomly generated, the following results were obtained and

[1]Excel is a registered trademark of the Microsoft Corporation

stored as one row in the results file, RESULTS.CSV.

- A listing of the task set.
- The objective value and sequence generated by heuristic #1. This composite heuristic schedules by the weighted sum of absolute penalty and penalty slope.
- The objective value and sequence generated by heuristic #2. This heuristic uses a relaxed penalty, then schedules by lowest penalty first.
- The objective value and sequence generated by heuristic #3. This heuristic uses penalty, then schedules by lowest penalty first.
- The objective value and sequence generated by heuristic #4. This heuristic uses the slope in the region of the completion time, then schedules by greatest slope first.
- The objective value and sequence generated by heuristic #5. This heuristic uses the processing time of each task. Shortest processing time tasks are scheduled first.
- The objective value and sequence generated by branch and bound. Tasks are scheduled in accordance with the results of the branch and bound search. Also listed is the number of cuts or nodes pruned by the branch and bound algorithm.

The results are shown in Figure 6.8, which shows the distribution of heuristic errors. For this plot, the minimum objective found by any heuristic was compared to the optimal objective obtained by branch and bound. The plot of Figure 6.8 is a histogram which shows the number of occurrences (frequency of tasks) for each value of error (difference between optimal and heuristic objective value). While in nearly all cases the heuristic performed quite well, there are a small number of cases where it performed poorly. This is a result of the heuristic using specific characteristics of task penalties to choose tasks for the sequence. For task sets where these characteristics do not correlate with the best sequence, the heuristic will not perform well. Thus, we conclude the heuristic is useful, and as expected, performance cannot be guaranteed.

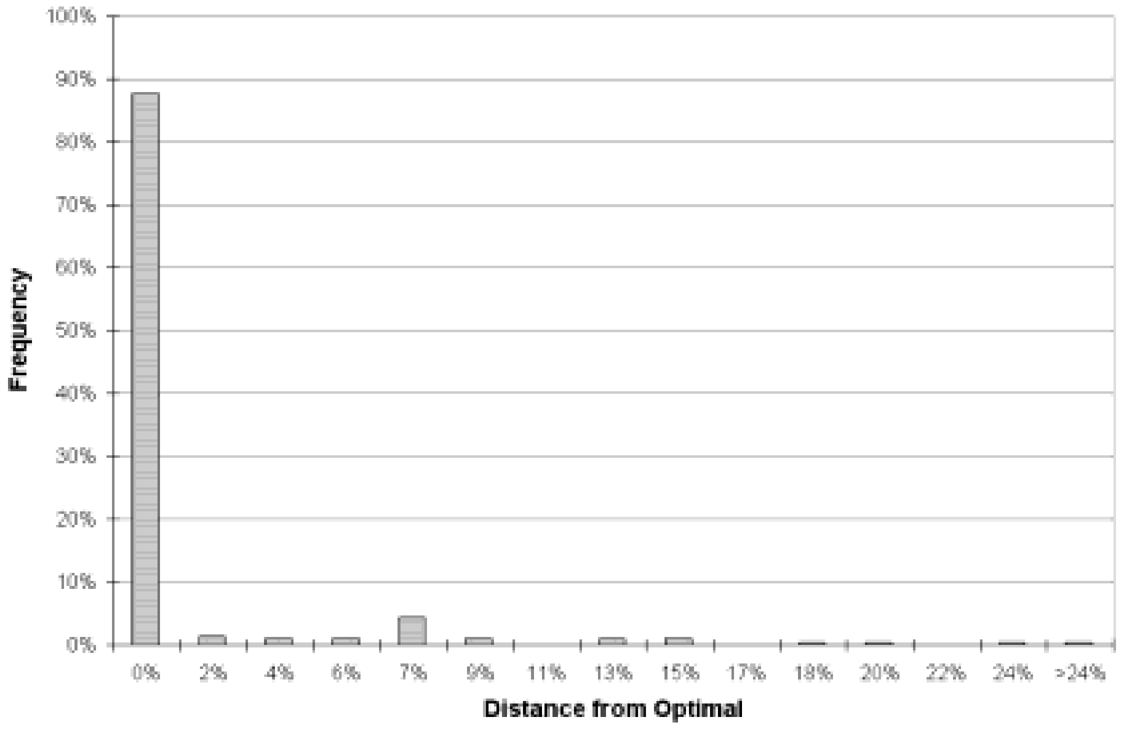

Figure 6.8: Histogram, Heuristic Accuracy

The results of the simulation are useful only to the extent that the task sets are representative the actual task sets that will be used by the heuristic. There are no performance guarantees for the heuristic and if it is to be employed in a specific application, further simulations are warranted. Indeed, the heuristic includes several weighting factors, and these should be adjusted to fine tune performance for any real application. The results also show that the combined heuristic was nearly always better than any of the single criteria heuristics, although it often occurred that one or two single criteria heuristics should match the combined heuristic.

Chapter 7

Tasks with Temporal and Logical Constraints

In Chapter 6 we explored the problem of multiple temporal deadlines. In this chapter we consider the effects of additional requirements imposed by logical constraints between tasks. Typically, the addition of logical constraints would make the problem simpler by reducing the size of the search space. Unfortunately, this is not necessarily the case here. We seek a solution that *permits the violation* of logical constraints, and therefore we cannot automatically discard portions of the state space that include logical constraint violations.

We will begin this chapter by examining the scheduling problem where there are only logical constraints and no temporal constraints. This is a fairly simple problem under the assumption that there are no cycles in the precedence graph. With this assumption, an exact solution can be easily obtained. We then discuss the problem of minimizing deadline penalties as in Chapter 6, but with the additional requirement that all logical constraints are met. In this case, the problem *is* simpler due to a reduction in the search space. Finally, we generalize the problem to allow logical constraint violations and propose several strategies to solve this problem.

7.1 Tasks with Logical Constraints

Prior to considering the problem of tasks with both temporal and logical constraints, we will first discuss logical constraints only. By comparison to the problem of meeting temporal constraints, logical constraints can be met with much less difficulty.

As in previous chapters (for example §3.2.1), we model logical constraints using precedence ordering. Each task may be related to any other task through a partial ordering, with a penalty associated for task pairs that violate the precedence constraint. The magnitude of the penalty reflects the error or imprecision introduced to the system (or its output) as a result of performing tasks out-of-order. Penalties are cumulative and, under the assumptions of a metric space, the total imprecision introduced into the system as a result of precedence violations is bounded by the sum of imprecision introduced by each violation alone. As in earlier cases, we assume that penalties are not transitive with respect to precedence constraints.

7.1.1 Model

Consider a set of tasks $1, 2, \ldots, N$. Tasks may be partially ordered, that is, a task may have one or more predecessor tasks. Partial ordering is expressed through precedence constraint pairs as: $a \prec b$. The relationship is transitive, i.e., if a is a predecessor to b, and b is a predecessor to c, then a is a predecessor to c. We require that there be no *cycles* in the partial ordering, that is, a task cannot be contained in its predecessor set.

In practical applications cycles rarely exist, thus we do not consider the problem where cycles may occur. If we assume there are no cycles in the precedence constraints, then a schedule with no precedence violations must exist. We make this assumption for all models described here, but we will revisit this assumption in the concluding remarks.

Partial ordering among tasks reduces the number of possible sequences. For example, in a set of tasks with no precedence relationships, there are $n!$ possible sequences. However if i tasks are partially ordered as $1 \prec 2 \prec \ldots \prec i$, then there are $\frac{n!}{i!}$ possible sequences. If there are two partially ordered sequences, i tasks and j tasks in length, then the number of possible schedules is: $\frac{n!}{i! \cdot j!}$. For example, consider a set of 12 tasks. Assume that four tasks are partially ordered as follows $1 \prec 2 \prec 3 \prec 4$ and three tasks are partially ordered as $5 \prec 6 \prec 7$. The remaining tasks $8 \ldots 12$ are not ordered. The number of possible schedules of tasks is $\frac{12!}{5! \cdot 4!} = 3.3$ *million*. This is significantly less than the number of schedules ($12! = 480$ *million*) with no partial orderings. These rules can be used to reduce complex combinations of precedence chains, but clearly, partial ordering can greatly reduce the number of possible orderings of tasks.

Formulation. For the set of tasks $1, 2, \ldots, N$, define $x_{ij} = 0$ if task i completes before task j and 1 otherwise. Penalty ξ_{ij} is assessed if i is scheduled after j. Then the objective measuring the quality of a schedule with respect to precedence constraints is:

$$Min \sum_{i=1}^{N} \sum_{j=1}^{N} \xi_{ij} x_{ij}, \tag{7.1}$$

The objective is to minimize the total penalty, incurred by a schedule.

Procedure. As discussed previously, the precedence relation is transitive, but penalties are not. Since an optimal schedule must exist, there is no need to consider any schedule which results in a penalty, $\xi > 0$. Thus the penalty values assigned to ξ may be limited to $\{0, 1\}$. The following procedure, which is based on a simplified version of Lawler's Method will find a schedule that contains no precedence violations.

- Step 1. Show all constraints as a directed graph where each task is a single node, and each constraint is a directed arc. Figure 7.1 is an example of such a graph.
- Step 2. A task is *ready* if it is the last node of a set of tasks. That is, if it has no successor tasks. We will create a sequence starting with the last task. In Figure 7.1, tasks 3, 4 and 7 are ready and any may be chosen to be last in the sequence.
- Step 3. Choose any ready task and place first in the partial sequence (that is, prior to any tasks already scheduled). When a task is chosen, remove it from the graph. Its immediate predecessor task (or tasks)now becomes ready provided they do not have any other successor tasks. In Figure 7.1 if we choose task 7, then task 6 becomes ready. If we instead choose task 3, task 2 does not become ready since it is still a prececessor to task 3.
- Step 4. Repeat Step 3 until all tasks are scheduled. Note that *any* ready task may be chosen. For the example of Figure 7.1, a valid sequence is then $5 \rightarrow 2 \rightarrow 6 \rightarrow 1 \rightarrow 7 \rightarrow 4 \rightarrow 3$, among others.

Note that while we have chosen randomly among the ready tasks, Lawler's Method provides a basis for choosing among ready tasks through a temporal measure. We will consider such a technique in later sections.

This procedure finds a schedule that meets all precedence constraints; however, it finds only one. In future models, we will require not only all optimal schedules, but all near-optimal solutions as well.

Branch and bound solution. To find all solutions, an enumerative algorithm is required, and we use the same branch and bound approach used for temporal constraints in §6.2.

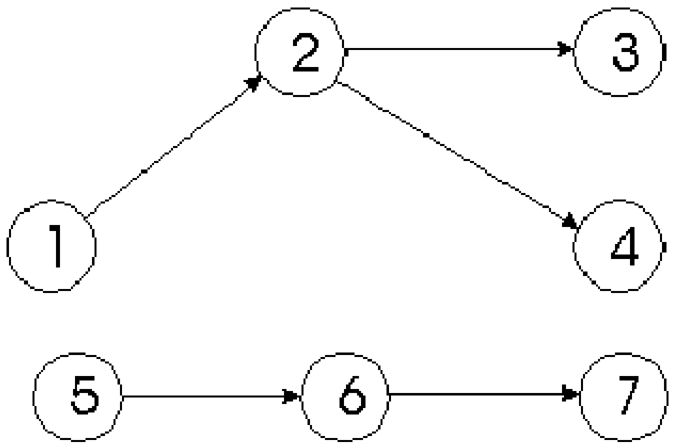

Figure 7.1: Network Diagram for Logical Constraints

Define the precedence penalty at any node as Ξ_i and the maximum allowable penalty as Ξ_{max}. We then perform a branch and bound search for all solutions, pruning nodes if the following condition is met:

$$\Xi_i \geq \Xi_{max} \tag{7.2}$$

This search will result in all schedules for which the precedence penalty is less than Ξ_{max}. If $\Xi_{max} = 0$, then branch and bound will find all optimal schedules, that is all schedules with no violations. For $\Xi_{max} > 0$, all schedules within Ξ_{max} of optimal will be found. If $\xi \in \{0, 1\}$ then the objective simply represents the number of violations permitted. By allowing $\xi \geq 0$, we can prioritize constraints.

This procedure gives us a method to find all schedules that permit some bounded (by Ξ_{max}) imprecision (or total precedence violation).

7.2 Alternate Criteria Approach for Tasks with Temporal and Logical Constraints

With solutions for the scheduling problems described for timing constraints alone (§6.2 and §6.3) and logical constraints alone (§7.1), we now focus on problems where both types of constraints are present, and specifically an alternate criteria approach to solve it.

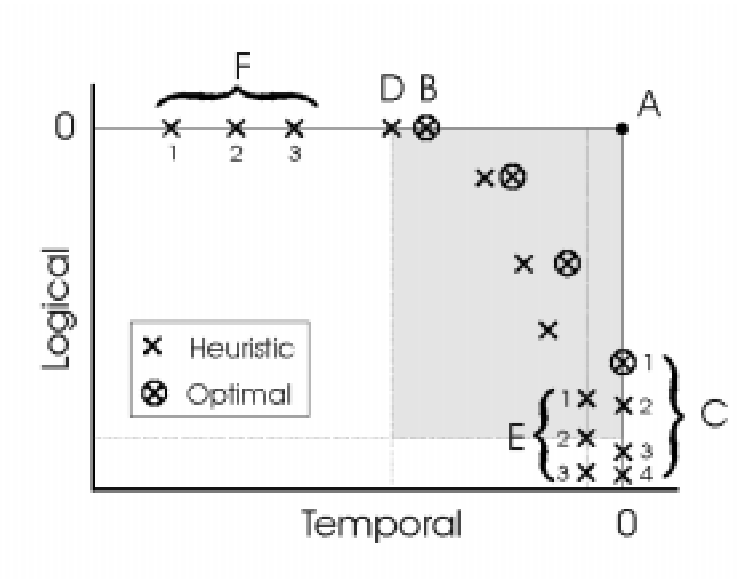

Figure 7.2: Frontier of Solutions

Consider the typical frontier diagram of Figure 7.2. Point A represents a solution that *dominates* all other possible solutions, that is, no point can be better than A in both measures. Of course, such a point may not exist. However, if A does exist there is no need to consider a trade-off between logical and temporal constraints since there is a schedule, and possibly several, that meets them both. From our previous discussions about the frontier (Chapter 2), if point A exists, it is a unique point on the frontier.

If instead, we assume that such a point does not exist, then we are left with the problem of finding a *set* of points, which are shown in Figure 7.2. The points shown as "$\otimes$" represent the exact solution to the frontier. Points shown as "$\times$" are not precisely on the frontier.

In this section, we seek to find the frontier of points that are optimal in both logical and temporal measures. However, we have already discussed methods that might provide some of these points. We now discuss the points that might be found by these methods. The solution for tasks with temporal constraints only was discussed in §6.2. Branch and bound was used to find optimal schedules with respect to deadline penalties. Points marked C_1, C_2, C_3 and C_4 represent a sample set of points that might be found for some set of tasks. These points are optimal for meeting deadlines, and one (C_1) has the lowest precedence penalty of the four points shown. The heuristic discussed in §6.3 finds one point that has a low value of deadline penalty. Assume, for example, there are three such points. They are marked E_1, E_2 and E_3. The three points are shown to indicate that the point found is without regard to precedence ordering, the heuristic will find one, and it is not known which will be the one found.

Similarly, in §7.1, we find points optimal with respect to precedence ordering. These are shown, also by example, as F_1, F_2, F_3, D, B. Branch and bound will find all these points, the procedure in §7.1.1 will only find one.

In the following two sections, we continue our search for frontier end points

7.2.1 Schedule Tasks to Minimize Deadline Penalty such that all Precedence Constraints are Met

This section describes the *alternate criteria* approach for solving a problem where tasks have both logical and temporal constraints. In this section, we seek to find the schedule with the least deadline penalty given that all precedence constraints be met. The concept of alternate criteria was described in Chapter 2. The method we describe is motivated by Lawler's method, which is a well known algorithm that schedules tasks to minimize a regular measure of performance such that all precedence constraints are met [Fre82]. To implement Lawler's method an algorithm that optimally schedules tasks with temporal constraints is required. We do not have such an algorithm. However, we can apply Lawler's method if we choose tasks by the non-optimal procedure described in §6.3. This will result in an application of Lawler's method that meets precedence constraints, but does not guarantee an optimal solution with respect to temporal constraints. We propose the following heuristic:

Consider a set of tasks with known processing times and a penalty function as described in Chapter 6. Also assume that the tasks are partially ordered. We seek to find a sequence of tasks such that deadline penalty is minimized, subject to all precedence constraints being met. The procedure for doing so is the same as described for the heuristic of §6.3, except only ready tasks are considered at each stage of the procedure. It is also similar to the procedure described in §7.1.1, except ready tasks are chosen by a heuristic and not randomly. The result of this procedure is a sequence that meets precedence constraints and focuses upon minimizing the deadline penalty as a secondary criteria. Referring to the frontier shown in Figure 7.2, we have found a point such as D. This point, as shown in Figure 7.2 may not actually be on the frontier, since a better schedule (point B) with respect to deadline penalties might exist. However, the point we have found is a lower bound for B. This will be useful when we consider an exact solution.

7.2.2 Schedule Tasks to Minimize Precedence Violation Penalties, such that Deadline Penalties are Minimal

The problem to minimize precedence violations such that temporal performance is optimal is the reverse of that problem of §7.2.1. This problem is somewhat more difficult than the previous problem, and although we cannot solve this problem directly, we can bound the solution such that the result will be useful in algorithms to follow. Using the heuristic of §6.3, we can find a schedule that ignores all precedence orderings, this will be one of points E_1, E_2 or E_3 in Figure 7.2. This solution gives the best estimate for temporal penalty, with some arbitrary value for precedence violations. If we assume that the heuristic point is no worse than the actual frontier point with respect to precedence penalties, then this heuristic point can be used to bound an enumerative search for the frontier. We discuss this issue further in §7.6.

7.3 Tasks with Multiple Deadlines and Precedence Constraints

With methods to find, or at least bound, the end points of the frontier, we can now further refine those methods to find the entire frontier. In this section, we combine the problems of tasks with temporal constraints and tasks with logical constraints and apply a bi-criteria solution. This model, introduced in §3.3, is the most general model that we will consider.

7.3.1 Model

Consider a set of tasks $1, 2, \ldots, N$. Each task i has processing time t_i and a set of deadlines d_{ik}, for $k = 1, 2, \ldots, B_i$. Associated with each d_{ik} is a penalty π_{ik}. We order deadlines such that $d_{i(k-1)} < d_{ik)}$ and impose a penalty π_{ik} if, for task i, deadline d_{ik} is missed, but deadline $d_{i(k-1)}$ is met (where $k > 1$). Further, $\pi_{i(k-1)} \leq \pi_{ik}$, where $k > 1$.

L_{ik} is used to indicate if task i completes prior to the k^{th} deadline, d_{ik}. $I_{ij} = 1$ if task i is the immediate precedessor to task j, zero otherwise. $x_{ij} = 0$ if task i completes before task j and 1 otherwise. Precedence penalty $\xi_{ij} \geq 0$ is incurred when task j scheduled before task i. Then the objective is:

$$Min \sum_{i=1}^{N} \sum_{k=1}^{B_i} \pi_{ik} L_{ik}, \tag{7.3}$$

subject to the following constraints:

$$\sum_{\substack{i=1 \\ i \neq j}}^{N} I_{ij} = 1, \quad \forall\, i \tag{7.4}$$

$$\sum_{\substack{j=1 \\ i \neq j}}^{N} I_{ij} = 1, \quad \forall\, j \tag{7.5}$$

$$c_i - c_j + p_j < M(1 - I_{ij}), \quad \forall\, i \neq j, j \neq 1 \tag{7.6}$$

$$c_1 = 0 \tag{7.7}$$

$$c_i - \sum_{k=1}^{B_i} d_{ik} L_{ik} \leq 0, \quad \forall\, i \tag{7.8}$$

$$\sum_{k=1}^{B_i} L_{ik} = 1, \quad \forall\, i \tag{7.9}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{N} \xi_{ij} x_{ij} \leq B \tag{7.10}$$

$$c_i - c_j < M x_{ij}, \quad \forall\, i, j \tag{7.11}$$

$$I_{ij} \in \{0,1\}, x_{ij} \in \{0,1\}, L_{ik} \in \{0,1\} \tag{7.12}$$

The objective (7.3) is to minimize the total penalty incurred by the schedule. This problem is again an instance of the TSP and is the same as the formulation of §6.1.1 but with additional constraints (7.10) and (7.11). These two additional constraints bound the total penalty to be less than B. This formulation will find a single frontier point such that the precedence penalty is less then B. To find the entire frontier, the problem must be solved by iterating on values of B from 0 to the maximum precedence penalty (or until a deadline penalty of zero is achieved).

7.4 Branch and Bound Approach for Finding the Frontier

The model presented in §7.3.1 combines both temporal and logical constraints into a single model. We seek to find bi-criteria solutions for a frontier that are best in temporal and logical performance as discussed in §3.2.5. In the previous sections, we have discussed methods to find one or more frontier points. We now extend those methods to find the complete frontier.

In this section, we describe a branch and bound algorithm to find the frontier. Recall that since branch and bound is an enumerative method it will find all points on the frontier. We seek to find methods to effectively reduce the search space so problems can be solved in a reasonable amount of time.

Search space reduction with temporal constraints. Define $\delta_T > 0$ to be a threshold for pruning nodes with respect to temporal performance. That is, at any node, if the penalty so far plus the estimate to any leaf node (LB) cannot result in a solution for deadline penalty better (lower) than δ_T, the node is pruned. The pruning threshold (δ_T) is set to find all optimal plus all sub-optimal solutions that might be included in the frontier. Thus, δ_T is set using the value for deadline penalty found in §7.2.1, which is an estimate of the minimal deadline penalty given that all precedence constraints are met. This is an end point on the frontier. This threshold reduces the search space, however, it not reduced nearly so much as was done in §6.2, especially when using the factor ϵ to further reduce the search space. Consequently, the processing time will increase.

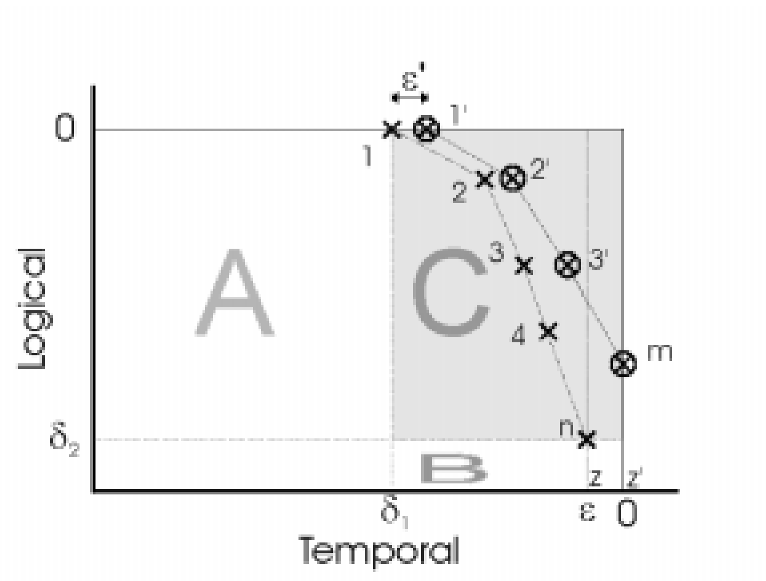

Figure 7.3: Search Space Regions

Recall the pruning rule (6.9). This rule is modified to expand the search space to include the frontier as follows. Define the temporal penalty at any node as Π_n. The lower bound from the node to any leaf is LB. We then prune if the following condition is met:

$$\Pi_n + LB > \delta_T \text{ where } \delta_T \geq 0, \tag{7.13}$$

The temporal threshold, δ_T, must be large enough to ensure that the search space includes the complete frontier, but ideally, no larger. This region is shown as $B \cup C$ in Figure 7.3. The heuristic of §7.2.1 determines point 1 in Figure 7.3. This point can be used for δ_T since its magnitude is at least as great as the magnitude of the optimal point marked $1'$ in Figure 7.3.

With the pruning rule given by the pruning rule (7.13), the search space is reduced to $B \cup C$. The region of B however, does not include the frontier, so we would prefer not to search this area either.

Search space reduction with precedence constraints. In §7.1.1 we discussed the problem of finding an optimal solution with just precedence constraints. The rule for pruning in this case is given by pruning rule (6.10). In a manner similar to the treatment for temporal penalties, we modify this rule to expand the search space to $C \cup A$ as shown in Figure 7.3.

Define the penalty at any node as Ξ_n, and $\delta_L > 0$ to be a threshold for pruning nodes with respect to logical performance. The lower bound, $LB = 0$ is the estimate of a solution to the end of a branch, note that we can always find a partial solution (to a leaf node) with no precedence violations. We prune if the following condition is met:

$$\Xi_n > \delta_L \text{ where } \delta_L \geq 0, \tag{7.14}$$

Again, the choice of δ_L is important. The threshold specified by δ_L must be large enough to ensure that the search space is sufficiently large to include the whole frontier. This region is shown as C in Figure 7.3. The heuristic of §7.2.2 determines point n in the figure, which is used for δ_T. If a suitable bound as described in §7.2.2 cannot be found, then the entire state space in respect to precedence violations $(B \cup C)$ must be searched.

Search space reduction with logical and temporal constraints. Using δ_T with the pruning rule (7.13), the search space is reduced to the region of Figure 7.3 shown as $C \cup B$. Using δ_L with the pruning rule (7.14), the search space is reduced to the region $C \cup A$. By using these two rules together, we can further restrict the search to only region C, which contains the frontier. The pruning rule is as follows.

Define the penalty at any node as Π_n for deadline penalties and Ξ_n for precedence penalties. The lower bound is LB_t for deadline penalty at any node. (Recall that LB_p, the lower bound for precedence penalties is zero at any node). We then prune if the following condition is met:

$$\Pi_n + LB_t > \delta_T \;\; OR \;\; \Xi_n > \delta_L \text{ where } \delta_T, \delta_L \geq 0, \tag{7.15}$$

The branch and bound algorithms described so far, in this and previous chapters have had a single objective and consequently, a single pruning rule. By searching with two objectives, we can prune using two rules. At each node, it is a simple matter to evaluate the objective function for both temporal and logical measures. Branches are pruned when the conditions of rule (7.15) are met.

Lemma 2: *An enumerative solution over all the solution space will find the frontier.*

Proof: By the assumption of non-decreasing deadline penalties we can eliminate from consideration all non-delay schedules. Thus, an optimal schedule can be represented by a sequence of tasks. Since enumeration considers all ordering of tasks, there can exist no schedules better than what is found by enumeration. □

Lemma 3: *If there exists, in the two dimensional search space of Figure 7.3, a schedule with minimal precedence penalty given that deadline performance is optimal* (x_1, y_1) *and a second schedule for which deadline penalty is minimal given that all precedence constraints are met* (x_2, y_2)*. Then, for each of these points the deadline penalty and precedence penalty can be calculated, and these two points define a rectangle which includes the frontier.*

Proof: There can be no point, (x, y), outside the rectangle with deadline penalty less than the minimum $(x < x_1)$ since we assume we have already found a schedule with the minimum penalty. Likewise, there can be no point outside the rectangle with precedence penalty less than minimum penalty ($y < y_2$). There also can be no point, (x, y), with a deadline penalty less than x_2 since (x, y) could not be on the frontier. Similarly, a point (x, y) with precedence penalty y_1 could also not be on the frontier.

$\square$

Thus, there does not exist a point that is greater in one measure and strictly better in the other measure than any point on the frontier calculated by branch and bound. Branch and bound, then, finds the best solution for the frontier where tasks that have both temporal and logical constraints. The frontier is a set of points, from which an optimal point can be selected. As with previous applications of branch and bound algorithms discussed in this dissertation, an exact solution is found, but processing time increases rapidly with task set size. The complexity of branch and bound is O(n!), which can lead to very long processing times.

Pruning of non-productive branches typically reduces that time significantly, but without guarantee. The use of the factor, δ, in the branch and bound rules increases the processing time over pruning when limited to optimal solutions. Simulation was performed to measure the effect of δ. For each data set, the full enumerative solution was found. The search space was then limited to the maximum value temporal penalty, the task set again solved and the time recorded. With this pruning limit, the region that includes the complete frontier will be searched. The pruning limit was then changed to reduce the space by 50%, that is, the limit was set to the average of the optimal and worst case frontier points with respect to deadline penalty. The reduction in search time was approximately 62% in a simulation of 20 task sets. If the points were distributed uniformly in the region, we would expect a reduction of one half. In the region of the frontier, we would expect points to be distributed in only the lower left corner of the region.

The branch and bound algorithm we describe finds all points on the frontier in no particular order. We begin the algorithm with two points that bound the frontier

$(0,0), (\delta_T, \delta_L)$. At each stage of processing, the computed frontier is a lower bound on the actual frontier. The longer the algorithm processes the better the solution. This continues until the algorithm terminates with an optimal solution.

In the next section, we explore a heuristic for solving this problem. It too, is based on an algorithm from previous sections, and like the previous algorithm, it cannot guarantee an optimal result.

7.5 Heuristic Based Construction of the Frontier

Since branch and bound cannot guarantee reasonable processing times for anything but small task sets, it is useful to have a heuristic that can provide a solution with less processing time, that is, in polynomial time. Such a heuristic has been developed, but as is usually the case, we cannot either guarantee the optimality of the result, nor can we guarantee that the heuristic will be acceptable for all applications. If the processing time to be allocated to the heuristic is limited, and we are willing to "fine tune" the heuristic for the specific application, this heuristic approach may be acceptable.

The heuristic here is based on the heuristic of §7.2.1. Recall that heuristic scheduled tasks to minimize deadline penalty such that all precedence constraints were met. We now wish to permit precedence violations, but such that deadline penalties are still minimized. We do this by allowing, or forcing, certain precedence violations to occur.

Beginning with no precedence violations we use the heuristic to find a schedule with minimal deadline penalty by the procedure described in §7.2.1. We then allow certain precedence violations to occur, but only so many as to cause a precedence violation penalty of 1. We then iterate this procedure to allow a precedence penalty of $2, \ldots M$, where M is the penalty for all precedence violations.

The key to the success of this heuristic is the selection of precedence constraints to violate. Choosing a set of precedence violation to result in a given penalty is an instance of the knapsack problem and very similar to the problem considered in Chapter 4. In the makespan problem of Chapter 4, tasks were selected for a knapsack based on a "weight" expressed as penalty, and a "value" measured by the processing time. Here, the weight is still the penalty, but the "value" is something more complicated. In fact, the value of violating a precedence constraint is proportional to the decrease in temporal penalty that would result if the tasks were to be scheduled out of order.

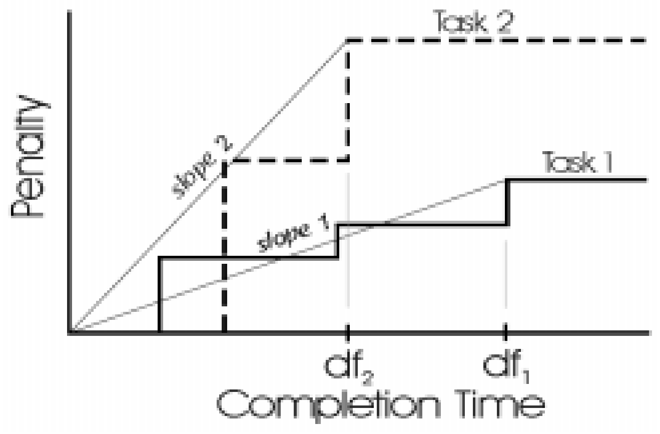

Figure 7.4: Penalty Benefit for Task Exchange

We define a heuristic measure of the advantage of scheduling two tasks in a precedence relation out of order. We shall refer to this value as the *Temporal Benefit Value* (TBV). TBV_{ij} is the temporal value obtained by relaxing the $i \prec j$ constraint and processing task j before i. Figure 7.4 shows graphically how the TBV is calculated.

A slope for each task deadline penalty function is calculated using the point

$(0,0)$ and the time and penalty at the final deadline for the task. These are shown in Figure 7.4 as df_1 for task 1 and df_2 for task 2. The slope values are subtracted, depending on the precedence ordering. If precedence ordering requires $task_1 \prec task_2$ with some penalty, then take the Temporal Benefit Value as $TBV_{1,2} = slope_2 - slope_1$ or 0 which ever is greater. If there is no precedence ordering specified, then TBV is set to 0, and there is no temporal benefit to violating the precedence relationship. But if $TBV_{1,2} > 0$ then there is an advantage to violating the precedence ordering $task_1 \prec task_2$. At the start of the heuristic, the TBV_{ij} is calculated for each task pair as $slope_j - slope_i$ to create a matrix of TBV_{ij} values. TBV_{ij} is used as the "value" in finding a knapsack solution. Observe that if $TBV_{ij} > 0$ then $TBV_{ji} = 0$

Procedure:

- Step 1. For a precedence penalty target to $B = 0$ (see model 7.1.1), perform the heuristic of §7.2.1. The solution is a candidate frontier point. Add this first point to the frontier.

- Step 2. Increment B by 1. Using dynamic programming (as described in Chapter 4) find the best knapsack solution to choose one or more precedence pairs for a knapsack of the target penalty B. Use the precedence penalty for "weight" and the TBV_{ij} for "value". For chosen precedence pairs "reverse" the precedence order and again schedule using the heuristic of §7.2.1. However, calculate the precedence penalty using the original precedence ordering.

- Step 3. The solution is a candidate frontier point. Add this point and test frontier for excess points as above. That is, add the point to the frontier if no better points already exist on the frontier. If a point is added to the frontier, then any points that are not as good as the added point must be removed. Recall the definition of a frontier point in §3.2.5 determines if a point is "better" than another.

- Step 4. If the precedence target is the maximum precedence penalty possible (sum of all penalties), then algorithm is complete. Otherwise, go back to step 2.

Using this heuristic, a frontier, which specifies the trade-off between logical and temporal requirements, is created. An optimal schedule is chosen from the frontier which exemplifies the relative importance of each. This heuristic suffers both same advantages and disadvantages for heuristics presented previously. Briefly, this heuristic has the advantage that it executes rather quickly—compared to branch and bound—and is not so adversely affected by task set size. However, it is not guaranteed to find an optimal solution, and it may require modification to optimize it for a specified application.

A more complete definition of this heuristic is presented in pseudocode form in Appendix B.

7.6 Enhancements to Branch and Bound

In the previous sections of this chapter, we have proposed both a branch and bound and heuristic solutions, each with advantages and disadvantages. In this section we explore some techniques to improve branch and bound by more effective pruning and also, to use the heuristic to further bound the branch and bound algorithm.

The Milestone Approach. In Chapter 2 we discussed the milestone method for computing where imprecise results are acceptable [LLS$^+$91]. Recall that the milestone method employs an iterative approach to calculations. At any time in the calculation, a result is available. The precision of the result increases with time, but if time is limited, then an imprecise result can be used. This method is directly applicable to our use of the branch and bound algorithm. The algorithm described in §6.2 searches the state space for points on the frontier. As new points are found, previous points may be deleted. When the algorithm terminates, all nodes have been fathomed, and the entire frontier is found. During the process, however, frontier points are discovered in an arbitrary order. Each successive frontier point provides a better lower bound on the frontier, which is also an estimate of the frontier. Beginning with the two points provided by the heuristic—and which is the first approximation of the frontier—each additional point found by branch and bound refines that approximation. This process is meets the criteria of the milestone approach. We can terminate the branch and bound algorithm at any time and return an approximation of the frontier, where this approximation is a lower bound to the exact solution.

Enhanced Pruning. We discuss in §6.2 the pruning conditions to reduce the search space to a region that includes the frontier. With this reduction, the search requires less time. However, with each point found on the frontier, the search space can be further restricted. Recall that points are added to the frontier if, for a candidate point, a better frontier point does not exist. Figure 7.5 shows the situation. Assume points a, b, c and d are points currently on the frontier. Any new point that falls in the "shadow" of any existing point cannot be on the frontier. The shadow is the regions marked A, B, C and D for points a, b, c and d respectively. The fact that there are regions that cannot contain a frontier point suggests further opportunities for pruning. For some point (t, p) where t is the deadline penalty value and p is the precedence penalty value, can prune nodes if the following condition are met:

$$\Pi_n + LB_t > t \;\; AND \;\; \Xi_n > p \tag{7.16}$$

where Π_n and Ξ_n are the total deadline and precedence penalty evaluated at the

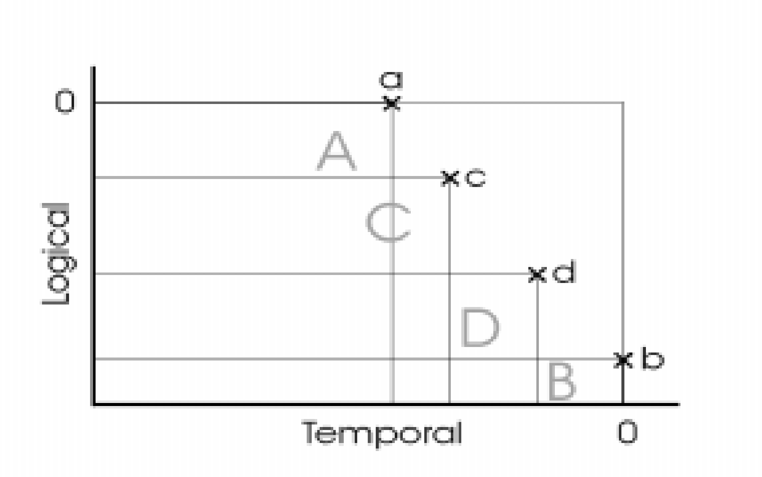

Figure 7.5: Enhancements to Pruning

node and LB_t is the lower bound for the deadline penalty to end of the branch. This pruning rule is addition to the pruning rule (7.15). The condition of pruning rule (7.15) is "AND'ed" with an instance of pruning rule (7.16) for all frontier points. For a large number of frontier points, the pruning rule can become quite large. Frontier points that are close to other frontier points will only provide a small reduction in the search space, thus some points can be eliminated from the rule without minimal impact.

Frontier End Point. Finally, we consider the problem discussed in §7.2.2. Recall that if the heuristic point was not a good estimate of the actual frontier point, and specifically, if the heuristic estimate of precedence penalty was less than actual, the search space might be reduced too much. This leads to the possibility that a point or points at the end of the frontier might be missed. An addition to the pruning rule (7.15) can correct this situation. By not pruning nodes which fall into the region marked F in Figure 7.6, we can be guaranteed to find all frontier points, including points near the end of the frontier.

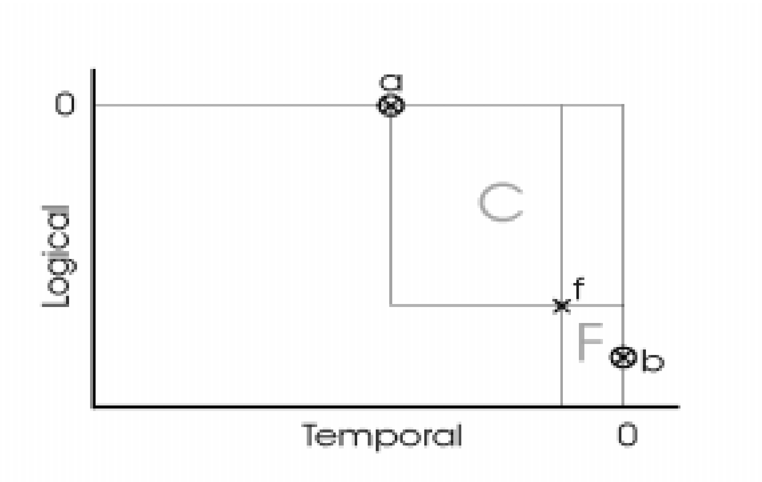

Figure 7.6: Expansion to Include All of Frontier

The change to pruning rule (7.15) is as follows: under the same assumptions and definitions of pruning rule (7.15) prune if the following condition is met:

$$(Pi_n + LB_t > \delta_T \;\; OR \;\; \Xi_n > \delta_L) \;\; AND \;\; Pi_n + LB_t < \Pi_h) \tag{7.17}$$

where Π_h is the value of the deadline penalty determined by the heuristic of point marked as point f in Figure 7.6, this point was shown as point E_2 in Figure 7.2. As a result of this addition to the pruning rule, region F will be searched in addition to region C, and should an optimal point at b exist, it will now be found.

Chapter 8

Conclusions

The trade off between logical and temporal consistency is a trade off that, in the appropriate situations, can provide schedules that best meet system performance goals. We have shown that the trade-off is valid and theoretically sound. Logical and temporal requirements use orthogonal measures of performance, and each measure can be varied through different schedules.

While the techniques discussed here apply to scheduling problems where both logical and constraints can be simultaneously be met, the strength of this work is in real-time systems that have logical constraints that must adapt to overload conditions. This requires that some temporal or some logical constraints may not be met. In systems were it is unacceptable to violate logical constraints or miss task deadlines, this approach would not be feasible. However, in time critical systems for which meeting deadlines is a requirement, then it may be acceptable to violate logical constraints. If the cost of missing a deadline is higher than the cost of performing a task out of order, then it is acceptable to use a schedule that violates logical consistency.

For this work to be useful in scheduling tasks there are five major assumptions that must be met:

- First, there must be two different measures of performance.
- Second, different schedules produce different levels of performance in the different measures.
- Third, the precedence relations relate to penalties in a metric space.
- Fourth, it is acceptable to trade-off performance in one measure for better performance in the other.
- Fifth, Deadline penalty and precedence penalty functions are known.

While we apply this work to two measures of performance, it can be extended to more than two. For any number of measures the above assumptions must hold. Additional measures of performance complicate the problem, thus it is advantageous to limit the model to only those measures that we desire to trade-off.

For this problem, we provide the general framework and a collection of algorithms and heuristics to provide solutions. Branch and bound methods are general in nature and can be applied directly, regardless of the individual characteristics of the application. Heuristics, however, must be crafted for the application since they generally rely on application specific characteristic that can be used to create a schedule.

We have assumed throughout this work that deadline and penalty functions are known. Without the knowledge of these functions it is not possible make a trade-off. Stankovic [Sta88] discusses the origins of deadlines, and how they are applied in a real-time system. Deadlines are usually driven by some external event, such as a truck leaving the loading dock. Likewise multiple task penalties are derived from several events, perhaps a morning shipment and an afternoon shipment. Deadline penalties arise from the costs of missing the deadline. These may be related to actual monetary costs, such as the additional costs of shipping by air if the deadline for ground shipment is missed. Penalties for precedence violations are somewhat more complex. Here, the costs relate to the task done incorrectly and the consequence on the whole job. Such costs may be monetary, such as the cost of rework of parts that do not meet quality standards due to incorrect processing. Possibly, the penalty is related to the increased risk of making the wrong decision based on imprecise data, such as from a database query. More difficult is the equivalence between logical and temporal penalties. In some cases where both temporal and logical penalties are expressed in monetary units, then the equivalence is well defined. But in general, this is not the case. Deadlines relate to timeliness while precedence relates to quality of output. These measures do not equate directly—but instead vary based on the desires of a customer or the philosophy of the system user. This issue relates to the simple question of getting something done on time, or doing it perfectly—which is more important and by how much?

A related issue is the "scale" at which the trade-off is applied. We have generally assumed low level tasks, which simplifies the problem and justifies the assumption of non-preemptable tasks. A disadvantage of making a trade-off with low level tasks is that it is more difficult to construct deadline and penalty functions, since the effect of these functions is harder to understand in the larger process. If we can assume non-preemptable tasks, then the methods discussed here are scalable to larger tasks, which may actually consist of larger work units. Non-preemptable tasks is a valid assumption in many applications. We leave the issue of preemptability of tasks as future work.

Although we have shown that many applications can be supported by our results, our unified scheduling technique may be inappropriate for many other applications. Throughout this work, we have assumed that deadlines are of critical importance to the system. If this assumption does not hold, then there is no justification to violate precedence constraints. We expect that the system has the potential to be in an overload condition, that the penalties for deadlines and precedence ordering are known and it is possible to trade off one for the other. If this is not the case, then this method is not appropriate.

As a result of this research, we have accomplished the following goals:

- Established a model for the characterization of real-time tasks in a database, FMS, mission planning, or similar system which might benefit from a trade off between temporal and logical constraints.

- Applied penalty functions with certain restrictions to represent temporal constraints. We showed that these penalty functions can serve as a useful proxy for a generalized temporal function and provide a means to tailor timing constraints to individual tasks.

- Defined a model for logical consistency using precedence ordering as a generalization of pessimistic concurrency controls. We have shown that violations of logical concurrency constraints introduce a bounded amount of imprecision into the system.

- Defined a model which provides different levels of temporal and logical performance based on the schedule of tasks. We showed that a set of efficient schedules exists that makes an explicit trade-off between logical and temporal consistency. We proved that the optimum schedule is included in that set.

- Using a incremental approach, we developed increasingly complex models. We also developed heuristics that will find an optimal schedule from the set of efficient schedules in polynomial time.

- Developed simulations to evaluate the scheduling methods and heuristics developed.

- We measured the performance of the heuristics and evaluated them for application to real-time computing. Through simulation of practical applications using randomly generated task sets, we verified the utility of the heuristics, and more generally, the performance gain by allowing schedules that explicitly make the trade-off between temporal and logical consistency.

Through the efforts of the work, we have shown the possibility of trading-off temporal constraints for logical constraints. Not only have we shown this trade-off to be sound, but we are able to explicitly define the trade-off. In support of this concept, we have extended the concept of *value functions* to include logical constraints modeled as precedence constraints. This work will be useful in the specific applications discussed, such as real-time databases, scheduling in FMS's and mission control for military applications. In addition, there are many other real-time applications that will benefit from this work—in fact, any application where the five assumptions discussed at the beginning of this chapter hold.

Chapter 9

Future Work

In the sections that follow we outline several areas that are ready for further exploration.

9.1 Delay Schedules

We have assumed throughout this work that the penalty function is non-decreasing. This has major implications for scheduling in that a non-decreasing penalty function insures that no delay schedule can be better than all schedules without delays, that is, if a delay schedule is optimal, there exists a non-delay schedule that is also optimal. A delay schedule is one that has idle time between tasks. If a penalty function for some task can decrease in time, over some portion of the scheduling horizon, then a better schedule might be achievable by *delaying* the task, that is, by the addition of idle time between the task and its predecessor task.

Delay schedules present a number of problems for a scheduler. First of all, an optimal schedule cannot be determined by enumeration. Because it executes in $n!$ time, enumeration is not all that useful, but at least in the case of non-delay schedules, it will provide an optimal schedule. We have used enumeration in this work to successfully evaluate heuristic methods. With delay schedules, the use of enumeration will be less useful.

Although delay schedules present several difficult problems, there may be some advantages provided by decreasing penalty functions in specific applications. Without resorting to delay functions, we presently handle the issue of tasks that may not be *ready* to execute by not introducing them to the system until they are ready. This forces tasks to delay execution until they are ready, but does not allow a penalty to be taken for execution slightly earlier than its ready time. Since this type of trade-off is central to our work, this method of delaying task until ready is not completely satisfactory.

9.2 Non Feasible Schedules

Since, in one sense, we assess penalties for late and out-of-order tasks; no schedules are infeasible, just expensive. However, if a penalty function has an infinite penalty (or an arbitrarily high *Big M* penalty) following some deadline, then we can define infeasible schedules as schedules that have a total deadline penalty which is infinite (or above M).

We can assume that a feasible schedule exists, but by assigning infinite penalties we create the possibility of in feasibility. In real-time applications infeasible schedules present serous problems, and at the least require some sort of exception handling.

9.3 Requirement to Execute all Tasks

In the static scheduling case of the general model introduced Chapter 3, the issue of whether all tasks must be executed is not a concern. Low penalty tasks, which might otherwise be deleted, can simply be moved to the end of the schedule. But in the dynamic scheduling case, such low penalty tasks are at risk of being delayed indefinitely. This is acceptable, from the standpoint of the scheduling policy, but at some point, the system needs a *garbage collection* function to clean out old tasks that are not likely to be executed. Failure to remove these stale tasks will increase scheduling overhead over time.

In our work here, we have alternately permitted and denied the possibility of deleting tasks when their execution is no longer of any benefit to the system. The important consideration is that we allow tasks to be deleted when it is clear that they are no longer viable, but do not permit deletion simply to avoid a penalty. The use of value functions, such as proposed by Locke [Loc86] provides a mechanism for deleting tasks. Tasks are deleted when their execution would add no value to the system. With penalty functions, however, the issue is not so clear. Penalty functions are non-decreasing, so the longer we wait before deleting a task, the higher the penalty. This argues for deleting all tasks at the start of the schedule to minimize the penalty. Such a strategy is clearly unacceptable. With penalty functions, a policy for deleting tasks is required, assuming that deleting tasks is acceptable. Such a policy is not difficult to formulate, for example, we may simply allow that tasks may be deleted after their final deadline, but not before. Note that in the static scheduling problem, task deletion is not an issue, since such tasks that are candidates for deletion can instead be processed at the end of the schedule. It is the dynamic case, were more tasks are likely to be added to the set, that deletion becomes an issue.

9.4 Generalized Penalty Functions

We have used in this work, a penalty function that is a non-decreasing multiple step function. We have already discussed the implications of non-decreasing functions, here we discuss the constraint of step functions. There are several other options for penalty functions including piece-wise linear functions and continuous functions. The difficulty in this scheduling arises from the discrete nature of the solution. A discrete number of tasks must be assigned to a discrete number of positions in a schedule. As we have seen, this is an instance of the Traveling Salesman Problem, a problem known to be $\mathcal{NP}$-hard. The problem would be must easier solved if part of one task could be scheduled in part of one position in the schedule, that is, solvable by *linear programming.* The discrete formulation forces this problem into a class of problems solved as *integer programs.* We can *relax* the discrete constraint for deadlines, which would appear to simplify the problem by removing the integer constraints. However, the basic integer constraints of tasks and position in schedule remain.

The heuristic and branch and bound algorithms do not require a multiple step function. However, the use of either continuous or piecewise linear functions is left for future work.

9.5 Bound on Processing Time for Branch and Bound Algorithm

As we have discussed, there is a trade-off between using an algorithm which provides a optimal solution and an algorithm that executes in polynomial time. Using the branch and bound techniques we have developed, processing time to schedule up to twenty tasks may be reasonable. But in the worst case, processing time can be much longer. The use of an ϵ can be used to decrease the search time in return for a sub-optimal schedule. If the processing time for the branch and bound is monitored, then if the algorithm processing exceeds some limit, either the schedule determined by the fast heuristic can be used, or ϵ increased and the branch and bound run again.

This is one avenue for future work. Further investigation of the milestone approach through simulation to determine processing times is suggested.

9.6 Dynamic Scheduling

The work we have discussed here uses a model of *static* scheduling. That is, all task attributes are known to the scheduler *a-priori*. Thus the scheduler, prior to building a schedule has all the information necessary to make the right decisions. In real-time systems, however, it is more likely that new tasks will enter the system requiring system resources. In this situation of *dynamic* scheduling, the system must adapt to the new information and re-schedule the tasks remaining to be processed.

While dynamic scheduling may be the more realistic approach in a real application, we chose not to consider it for this work. Dynamic scheduling complicates the issue since it requires that the scheduler adapt and re-schedule new tasks. For this work we chose to focus on the core issues of temporal versus logical scheduling goals and not further cloud this issue with the additional requirements imposed by dynamic scheduling.

Looking toward future work, it is desirable that this work evolves to include dynamic scheduling. In its simplest form dynamic scheduling can be accomplished by simply repeating the static scheduling algorithms at periodic intervals or whenever task attributes change that would affect the schedule. However, this approach may not be the most efficient. Continuous re-scheduling using the static scheduling model will impose a high processor overhead on the system degrading performance.

The implementation of a dynamic schedule algorithm can take advantage of information from the previous schedule, and this information can be used to find the next schedule. However, the techniques we have developed here do not require a large amount of the storage during execution. For example the *depth first search* that we have employed needs only keep track of where it is in the tree and several attributes of the candidate schedule.

In our branch and bound algorithm we seed the algorithm with a candidate solution derived from a heuristic. There is no requirement for the accuracy of the seed, however, the better the seed, the more quickly branch and bound is likely to find a good solution which is then used to prune other nodes. In a dynamic scheduling environment, an optimal solution from a previous schedule could be used to seed the current solution. Unfortunately, it does not seem that this approach would provide significant improvement. The use of a seed schedule only provides modest improvement in branch and bound processing time, a seed schedule can quickly be calculated by heuristic and finally the schedule from a previous schedule is likely to contain different tasks to be scheduled.

Perhaps a more promising approach would be the modification of our heuristic to accommodate dynamic scheduling. In this way, completed tasks could be deleted from the system and new tasks inserted into the schedule such as to minimize penalty. The low complexity of this algorithm makes it attractive computationally. At periodic intervals, the schedule created using this method could be validated using a fast static scheduling heuristic. Or, at less frequent intervals, a more complete static scheduling algorithm could be run to create a new baseline schedule.

9.7 Repetitive Tasks

We have, in this work, ignored the possibility that some tasks are repetitive. Instead, we assume in this static case, all tasks are known to the scheduler and need to be executed only once. Repetitive tasks are more relevant to the dynamic case, were we might *release* tasks periodically as they become ready. However, in a system were there are a number of repetitive tasks, it might be desirable to accommodate them independently of the other tasks. For example, an heuristic, such as the rate monotonic algorithm [LL73] could be used to schedule repetitive tasks, then using the methods discussed in this work, schedule the remaining tasks in the gaps between repetitive tasks. This alone is probably not sufficient, in that it may be desirable to permit the trade-off between logical and temporal constraints with both repetitive and non-repetitive tasks. Indeed, we it may be acceptable to miss execution of a repetitive task occasionally since it will be executed again at a later time.

9.8 Stochastic Scheduling

In our work, we have assumed that all processing times are known and fixed. In reality, processing times, well as the task set and penalty functions may be not so clear. Stochastic scheduling assumes that these task attributes are specified by probability functions rather than fixed numbers. This is a area that as many implications for the scheduling algorithms and is an area which should receive more research.

9.9 Future Applications

In this dissertation we have described a method to schedule tasks with both temporal and logical constraints in real-time systems. Much of the motivation comes from the systems that in "overload" condition where not all constraints can be met. However, this is not meant to be exclusive, since this work applies to any real-time systems where are timing requirements are in conflict with logical constraints, even if the system is not in overload. The frontier provides an accurate, intuitive and useful metric for scheduling tasks under these conditions.

In many applications, developers have struggled with the problem of handling timing requirements in the presence of logical constraints. Relaxed alternate criteria solutions, penalty functions contrived to enforce precedence are two of the ways this problem has heretofore been approach. With this work, solutions to problems of this type should be more directly and more easily solved.

References

[BHG86] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, New York, 1986.

[Bol90] Bela Bollobas. *Graph Theory, An Introductory Course.* Springer-Verlag, 1990.

[Cla90] Raymond Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, Carnegie-Mellon University, 1990. PhD Dissertation, Department of Computer Science, Carnegie-Mellon University.

[CLB94] C. Chuen-Lung and R. Bulfin. Scheduling a single machine to minimize two criteria: Maximum tardiness and number of tardy jobs. *IIE Transactions*, 26(5):76–84, Sep 1994.

[cpl94] CPLEX Optimization, Inc. *CPLEX User Manual*, 1994.

[DiP95] Lisa Cingiser DiPippo. *Object-Based Semantic Real-time Concurrency Control.* PhD thesis, University of Rhode Island, 1995. PhD Dissertation, Department of Computer Science, the University of Rhode Island.

[FO89] Abdel Aziz Farrag and M. Tamer Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[Fre82] S. French. *Sequencing and Scheduling, An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood Limited, 1982.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database system. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[HP94] A. Hariri and C. Potts. Single machine scheduling with dealines to minimize the weighted number of tardy jobs. *Management Science*, 40(12), 1994.

[Jen96] D. Jensen. Real-time manifesto. Published on the Internet: http//www.realtime-os.com/rtmanifesto/, 1996.

[JS95] G. Jones and M. Sodhi. A method for describing operation sequences in flexible manufacturing systems. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, July 1995.

[KM92] Tei-Wei Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Real-Time Systems Symposium*, December 1992.

[KM93] Tei-Wei Kuo and A. K. Mok. A semantics-based protocol for real-time data access. In *Real-Time Systems Symposium*, December 1993.

[Law78] E. L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[lin94a] Lindo Systems, Inc. *LINDO User Manual*, 1994.

[lin94b] Lindo Systems, Inc. *LINGO User Manual*, 1994.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLKS85] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem*. John Wiley and Sons, 1985.

[LLS$^+$91] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computation. *IEEE Computer*, 24(5), May 1991.

[Loc86] Douglas Locke. *Best-Effort Decision Making For Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, 1986. PhD Dissertation, Department of Computer Science, Carnegie-Mellon University.

[Lyn83] Nancy A. Lynch. Multilevel concurrency – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[PDPW94] JJ Prichard, Lisa Cingiser DiPippo, Joan Peckham, and Victor Fay Wolfe. RTSORAC: A real-time object-oriented database model. In *Proceedings of the International Conference on Database and Expert Systems Applications*, Sept 1994.

[Pin95] M. Pinendo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[PM88] Joan Peckham and Fred Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, Sept. 1988.

[Ram93] Krithi Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[RP] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. to appear in *Transactions on Knowledge and Data Engineering.*

[SM90] P. Toth S. Martello. *Knapsack Problems; Algorithms and Computer Implementations.* John Wiley and Sons, 1990.

[Sta88] John Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10), October 1988.

[VCY95] G. Vairaktarakis and L. Chung-Yee. The single-machine scheduling problem to minimize total tardiness subject to minimimum number of tardy jobs. *IEE Transactions*, 27:250–256, 1995.

[Win93] W. Winston. *Operations Research, Applications and Algorithms.* Duxbury Press, 1993.

[WYP92] K. Wu, P. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of the International Conference on Data Engineering*, 1992.

[YWLS94] P. S. Yu, Kun-Lung Wu Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

Appendix A

Heuristic to Schedule with Temporal Requirements

This appendix includes the pseudocode for the heuristic described in the text for creating a schedule of tasks with temporal constraints (§6.3). The main subroutine (Schedule_tasks), is the entry routine. It creates a schedule of tasks using one of five user specified heuristics.

This pseudocode loosely follows the conventions of the "C" programming language, with some minor additions for clarity. Key words (C commands) are capitalized. Variable names are all lower case; however, some liberties have been taken with declaration, initialization and passing arguments in subroutines. Subroutine names are lower case with first letter capitalized. The FOR loop structure is expressed as: `DO FOR loop_counter = initial TO final`. The DEFINE keyword defines only variables and structures which are global. Comments use the conventional C notation of `/* comment */`. Except for the entry subroutine (Schedule_tasks), all subroutines return a single value. The entry subroutine returns a list (schedule[]) as a passed argument.

```
DEFINE big_M;  /* integer representation of "infinity" */
DEFINE number_of_tasks;
DEFINE number_of_deadlines;

DEFINE STRUCTURE
{
   task_number;
   objective;
}schedule[number_of_tasks];

DEFINE STRUCTURE
{  /* task attribute and temporal penalty table */
   task_time;
   deadline_array[number_of_deadlines];
   penalty_array[number_of_deadlines];
} task_list [number_of_tasks];


SUBROUTINE  Get_index(task_number, completion_time)
{  /* returns index into penalty_array for completion time, that
      is, returns pointer to task penalty corresponding to task
      completion time */

   DO WHILE {task_list[task_number].deadline[i]<completion_time
            AND i<number_of_deadlines} i++;
   RETURN (i);
}

SUBROUTINE Calculate_penalty(task_number, index)
{  /* returns penalty for a task given the index into penalty
      array */

   /* if past last deadline, penalty is Big_M */
   IF(index == number_of_deadlines) RETURN (Big_M);

   ELSE RETURN (task_list[task_number].penalty[index];
}

SUBROUTINE Calculate_smooth_penalty(task_number,index)
{ /* returns penalty for task given the index into penalty
     array, but assuming a linear interpolation between penalty
     steps (see text) */

   /* if past last deadline, penalty is Big_M */
   IF (index == number_of_deadlines) RETURN (Big_M);

   /* note, if first deadline is met, then first point is (0,0)*/
   ELSE IF(index == 0)
   {
      y = task_list[task_number].penalty[index];
      x = task_list[task_number].deadline[index];
   }

   /* for any other deadline met, find delta x and delta y from
      previous deadline point */
   ELSE
   {
      y = task_list[task_number].penalty[index]
          - task_list[task_number].penalty[index-1];
      x = task_list[task_number].deadline[index]
          - task_list[task_number].deadline[index-1];
}

   /* if x is zero, then "interpolated penalty" is just penalty*/
   IF (x == 0) RETURN task_list[task_number].penalty[index];

   /* return linear interpolated penalty */
   ELSE RETURN (y/x * (completion_time
         - task_list[task_number].deadline[index])
         + task_list[task_number].penalty[index]);
}

SUBROUTINE Calculate_completion_time()
{  /* calculates completion time for task next to be scheduled,
      since last task in sequence is scheduled first, initially,
      the completion time is sum of all task processing times.
      As tasks are scheduled, completion time is the sum of all
      unscheduled task processing times */

   DO FOR n=0 TO number_of_tasks
   {
      /* check to see if task is scheduled */
      DO FOR m=0 TO number_of_tasks
      {
         IF (schedule[m].task_number==n) scheduled = TRUE;
      }
      IF NOT (scheduled)
            completion_time += task_list[n].task_time;
      }
}

SUBROUTINE Start_time(task_number, completion_time)
{  /* given a completion time, finds the start time of the task,
      if it were to be scheduled next */

   RETURN (completion_time - task_list[task_number].task_time);
}

SUBROUTINE Maximum_penalty(completion_time)
{  /* for a given completion time, return the penalty value for
      the task with the greatest penalty */

   DO FOR n=0 TO number_of_tasks
         max_penalty = MAX (get_penalty(get_index(n,completion_time)),
            max_penalty);

   RETURN (max_penalty);
}

SUBROUTINE Get_slope(index_start, index_end, task_number)
{  /* find the "slope" of the penalty function in the region of
   the deadline */

   RETURN (Calculate_smooth_penalty(task_number,index_end)
              - Calculate_smooth_penalty(task_number,index_start))
              / task_list[task_number].task_time);
}

SUBROUTINE Test_for_scheduled(task_number)
{  /* tests the task (task_number) to see if it has been already
      scheduled */
   DO FOR n=0 TO number_of_tasks
      IF(schedule[n].task_number == task_number) RETURN(TRUE);

   RETURN(FALSE);
}

SUBROUTINE Task_ready(task_number)
{  /* NOTE:  this subroutine will be replaced by a different
      routine when precedence constraints are considered in
      Appendix B */

   RETURN(TRUE)
}

SUBROUTINE Choose_task(position, criteria, completion_time)
{  /* for a given completion time, returns task_number of task
      with best (lowest) score based on the criteria specified.

      criteria: 1 = lowest composite penalty (requires "alpha")
                2 = lowest penalty
                3 = lowest smoothed penalty
                4 = lowest slope in region of completion time
                5 = shortest processing time */

   min_score = big_M;
   min_score_task_number = NULL;

   DO FOR n=0 TO number_of_tasks
   {
      IF(Test_for_unscheduled(n) == TRUE AND Task_ready(n))
      {
         slope = get_slope(get_index(n,get_start(n,
                   completion_time),(get_index(n,completion_time),n);

         penalty = calculate_penalty(n,get_index(n,completion_time));


         BEGIN CASE
         {
            /* composite penalty (alpha chosen empirically) */
            CASE(criteria == 1) score = alpha * slope
                       + penalty/maximum_penalty(completion-time);

            /* penalty */
            CASE(criteria == 2) score = penalty;

            /* smoothed penalty */
            CASE(criteria == 3) score =
                        Calculate_smooth_penalty(n,get_index(n,
                        completion_time);

            /* slope */
            CASE(criteria == 4) score = slope;

            /* processing time */
            CASE(criteria == 5) score=task_list[n].task_time;
         }

         IF(score < min_score)
         {
            min_score = score;
            min_score_task_number = n;
         }
      }
   }

   schedule[position].task_number = min_score_task_number;
   schedule[position].objective   = min_score;
   RETURN;
}

SUBROUTINE Schedule_tasks(task_list, schedule, criteria)
{  /* creates a schedule of tasks based on the specified
      criteria.  Tasks are defined by task_list[], schedule is
      returned in schedule[] as a list of task_number */

   DO FOR n=number_of_tasks TO 0
   {
      Choose_task(n, criteria, Calculate_completion_time());
```

```
            objective = objective + schedule[n].objective;
    }
    RETURN(objective);
}
```

Appendix B

Heuristic to Schedule with Temporal and Logical Requirements

This appendix includes the pseudocode for the heuristic described in the text for scheduling a set of tasks that have both temporal and logical constraints. (§7.5). The main subroutine (Find_frontier), is the entry routine. It returns a list of frontier points, from which the best schedule can be chosen. The heuristic described here effectively creates a list of ready tasks. Then the heuristic of Appendix A is called to schedule tasks for temporal performance, but with the modification that only tasks that are *ready* can be scheduled. In order to accomplish this modification, the subroutine `Task_Ready` included in this appendix is substituted for the subroutine `Task_ready` originally included in Appendix A.

This pseudocode loosely follows the conventions of the "C" programming language, with some minor additions for clarity. Key words (C commands) are capitalized. Variable names are all lower case; however, some liberties have been taken with declaration, initialization and passing arguments in subroutines. Subroutine names are lower case with first letter capitalized. The FOR loop structure is expressed as: `DO FOR loop_counter = initial TO final`. The DEFINE keyword defines only variables and structures which are global. Comments use the conventional C notation of `/* comment */`. Except as noted, all subroutines return a single value—those that don't instead modify one or more of the globally defined data structures.

```
DEFINE STRUCTURE
{  /* this structure defines all precedence constraints.  The array
      entry (weight[x][y]= penalty) defines the penalty for
      violating the constraint x<y).  Penalty = 0 when no constraint
      is specified.  Value is the temporal benefit for violating the
      constraint x<y.  Value[x][y] = 0 if there is no temporal
      benefit for violating x<y, or if weight[x][y] = 0 */

   weight[number_of_tasks][number_of_tasks];
   value [number_of_tasks][number_of_tasks];
} prec_constraints[max_kps_size]


DEFINE STRUCTURE
{  /* this structure is an set of arrays.  An array contains
      the set of precedence violations required
      (kps_solutions[].selected[x][y] = 1 if x<y selected) to meet
      some penalty value.  There is an array for each penalty value,
      from 0 to the maximum.  */

   selected[number_of_tasks][number_of_tasks];
} kps_solutions[max_kps_size];

DEFINE STRUCTURE
{  /* task attributes and temporal penalty table */
      task_time;         /* not used in this heuristic */
   deadline_array[number_of_deadlines];
   penalty_array[number_of_deadlines];
} task_list [number_of_tasks];

DEFINE STRUCTURE
{
   task_number;
   objective;
```

```
}schedule[number_of_tasks];

DEFINE STRUCTURE
{
   schedule[number_of_tasks];
   /* frontier point (x,y) */
   x;
   y;
} frontier[];

DEFINE number_of_tasks;
DEFINE number_of_deadlines;

SUBROUTINE Create_TBV()
{  /* Using the temporal penalty functions, determines the
      value of violating a precedence constraint.  No value
      is returned */

   DO FOR i=0 TO number_of_tasks {
   DO FOR j=0 TO number_of_tasks
   {
      IF(prec_constraints.weight[i][j] > 0)
      {
         /* there is a logical benefit, what is the
            temporal benefit */
         slope_i = task_list[i].penalty_array[number_of_deadlines-1]
                 / task_list[i].deadline_array[number_of_deadlines-1];
         slope_j = task_list[j].penalty_array[number_of_deadlines-1]
                 / task_list[j].deadline_array[number_of_deadlines-1];
         /* temporal benefit is difference is slopes -- if > 0 */
         benefit = slope_j - slope_i;

         IF(benefit > 0)
            prec_constraints.value[i][j] = benefit;
   }}
   RETURN();
}


SUBROUTINE Test_for_frontier(x,y)
```

```
{  /* checks new point to see if should be on frontier,
      if it is, also checks to see any points now should be
      removed.

      A point is on the frontier when no other point is better
      in one measure and better or equal in the other

      Modifies the frontier list (frontier[]), returns NULL */

   /* see if point should be added (no point clearly better) */
   add_point = TRUE;
   DO FOR(n=0 to frontier_index)
   {
      IF(x < frontier[n].x AND
            y <= frontier[n].y) add_point = FALSE
   }

   /* should point be added? */
   IF(add_point == TRUE)
   {
      /* add point */
      frontier_index++;
      frontier[frontier_index].x = x;
      frontier[frontier_index].y = y;
      frontier[frontier_index].schedule[] = schedule[];

      /* check each point against new point */
      DO FOR(n=0 to frontier_index)
      {
         /* new point is better? */
         IF(frontier[n].x < x AND frontier[n].y <= y)
         {
            /* remove old point */
            frontier[n].x = NULL;
            frontier[n].y = NULL;
frontier[n].schedule[] = NULL;
         }
      }
   }
   RETURN();
```

```
}

SUBROUTINE Task_ready(task_number, kps_number);
{  /* NOTE:  this subroutine replaces the routine of the same
      name in Appendix A.

      A task is ready when it has no unscheduled successor tasks,
      where a successor task is defined by the original
      precedence constraints, or overridden by forced violation
      of precedence constraints as specified in the array
      kps_solutions(kps_number) */

   ready = TRUE;
   DO FOR(n=0 to number_of_tasks)
   {
      IF(prec_constraints.weight[task_number][n] > 0
         OR kps_solutions[kps_number].selected[task_number][n] == 1)
      {
         /* task_number is not ready if task n is not
            already scheduled */
         ready = FALSE;
         DO FOR(m=0 number_of_tasks)
         {
            IF(schedule[n].task_number == n) ready = TRUE;
         }
      }
   }
   RETURN(ready);
}

SUBROUTINE Array_and(destination, n, knapsack_size - n)
{  /* AND's two knapsack result arrays and puts the result in
      destination.  Destination is either a knapsack solution
      array or a temporary array */

   DO FOR i=0 TO number_of_tasks {
   DO FOR j=0 TO number_of_tasks
   {
      destination[i][j] = kps_solutions[n].selected[i][j]
            AND kps_solutions[knapsack_size-n].selected[i][j];
      IF(destination[i][j] == 1) value = value +
            prec_constraints.value[i][j];
   }}
   RETURN(value);
}

SUBROUTINE Combine_previous(knapsack_size, kps_solution);
{  /* begin by trying all solutions consisting of two previous
      solutions (including the solution for zero = null set),
      and pick the best, add to kps_solutions and return
      logical value */

   DO FOR n = 0 to knapsack_size - 1
   {
      IF(Array_and(temp, n, knapsack_size - n) > logical_value)
      {
         logical_value = Array_and(kps_solutions[knapsack_size],
               n, knapsack_size - n);
      }
   }
   RETURN(logical_value);
}

SUBROUTINE Knapsack_weight(knapsack_size)
{  /* Returns the current "weight" of knapsack solution */

   DO FOR i=0 TO number_of_tasks {
   DO FOR j=0 TO number_of_tasks
   {
IF(kps_solutions[knapsack_size].selected[i][j]==TRUE)
      {
         weight = weight + prec_constraints.weight[i][j];
      }
   }}
   RETURN(weight);
}

SUBROUTINE Add_more_weight(logical_value, knapsack_size)
{  /* Trying to add more un-selected constraint violations
```

```
      (weights), such that weight fits, and it has the
      highest value.  Logical_value is returned. */

   weight = Knapsack_weight(knapsack_size)
   weight_added = TRUE;
   DO WHILE(weight_added==TRUE)
   {
      weight_added==FALSE;
      DO FOR i=0 TO number_of_tasks {
      DO FOR j=0 TO number_of_tasks
      {
         /* see if it fits and it's available */
         IF(prec_constraints.weight[i][j] + weight < knapsack_size
               AND kps_solutions[knapsack_size].status[i][j] == FALSE;
         {
IF(prec_constraints.value[i][j] > max)
            {
               max = prec_constraints.value[i][j];
               max_i = i;
               max_j = j;
            }
         }
      }}
      kps_solutions[knapsack_size].status[max_i][max_j]=TRUE;
      logical_value = logical_value
            + prec_constraints.value[max_i][max_j];
weight = weight + prec_constraints[max_i][max_j];
      weight_added = TRUE;
   }
   RETURN(logical_value);
}

SUBROUTINE Dynamic_program(knapsack_size)
{  /* given a set of smaller knapsacks (kps_solutions[k]), find
      the optimal knapsack for a knapsack of size knapsack_size.
      Note that when a previous solution is used, any "stones"
      already selected, cannot be selected again.

      This subroutine modifies the frontier[] structure, it
      returns logical_value. */
   /* find solution by combining all combinations of previous
      solutions */
   logical_value = Combine_previous(knapsack_size);

   /* keep trying to add one more un-selected constraint
      violation */
   logical_value = Add_more_weight(logical_value, knapsack_size)

   RETURN(logical_value);
}

SUBROUTINE Schedule_tasks_prec(task_list, frontier, prec_constraints)
{  /* Creates a frontier of optimal schedules given task definitions,
      timing constraints and precedence constraints. */

   /* Create the table which gives the temporal "benefit" for
      violating a precedence constraint */
   Create_TBV();

   /* Use dynamic programming to choose
   DO FOR kps=0 TO max
   {
      logical_value = Dynamic_program(kps);

      /*  given a set of constraints to meet and to *not* meet,
          find the best temporal schedule */
      temporal_value = Schedule_tasks(task_list, schedule, criteria);

      /* Check to see if this point is on the frontier */
      Check_frontier[temporal_value][logical_value];
   }
   RETURN();
}
```