Algorithms For Constraint-Based Temporal Reasoning With Preferences

by

Bart Michael Peintner

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2005

Doctoral Committee:

Professor Martha E. Pollack, Chair
Professor Karem A. Sakallah
Professor Michael P. Wellman
Assistant Professor Amy E. M. Cohn

Copyright page

To my wife Liz, whose love and support gave me the strength to push through the low points and struggles that arose while earning this degree. She is a constant inspiration, and has shown me, through her confidence and accomplishments in her own career, what I can achieve in mine.

ACKNOWLEDGEMENTS

First and foremost I thank my adviser, Martha Pollack, who from the beginning had greater confidence in me than I had in myself. I have overwhelming gratitude for her patience, guidance, and constant advocation on my behalf. I owe a great deal of my growth and achievements in this period to her mentorship.

Michael Wellman spent considerable effort reviewing this document, which resulted in a significant addition and more precise claims and explanations. I was encouraged by the genuine interest Karem Sakallah showed in my topic. With the help of Hossein Sheini, he showed me a different way to view my problems, which helped me better understand the strengths and weaknesses of my approaches. I enjoyed my meetings with Amy Cohn, who helped me understand how this work relates to similar problems in Operations Research. My conversations with Satinder Singh Baveja had a significant impact on my general approach to science and the way I evaluate my work.

My conversations with Peter Schwartz and Michael Moffitt helped solidify my ideas and lead me more quickly to conclusions and errors that would have taken me much longer to find. I received similar help from Robert Morris at NASA Ames, who encouraged my early ideas that led to this thesis.

Many thanks to Peter, Michael, Britton Wolfe, Matt Rudary, Mark Schaller, Joe Taylor and Julie Weber for hours of entertainment. I was lucky to have such great people to help me waste time. Many others made my time in the lab enjoyable,

especially Mazin As-Sanie, John Hawkins, Sailesh Ramakrishnan, Andrew Nuxoll, Dirk Colbry and Colleen van Lent.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

ABSTRACT

Recent planning and scheduling applications have successfully used the Temporal Constraint Satisfaction Problem (TCSP) to model events and temporal constraints between them. Given a TCSP, the task is to find a schedule or a set of schedules that satisfy all constraints in the problem. A key limitation of TCSPs and related formulations is that they only represent hard constraints—constraints that are either satisfied or not. Many real-world situations are better modeled with soft constraints—constraints that are satisfied to a particular degree. Soft constraints allow knowledge engineers to express that some situations are preferable to others.

This dissertation augments TCSPs with soft constraints and presents algorithms that find optimal solutions. We represent soft constraints by augmenting hard constraints with a preference function that maps every valid solution to a local preference value that describes how well the constraint is satisfied. We then aggregate the local preference values to attain the value of the entire solution. We studied two aggregation functions, the *sum* and *minimum* functions.

We developed a set of algorithms that optimize with respect to both aggregation functions when preferences are added to each of two common TCSP subproblems: the Simple Temporal Problem (STP) and the Disjunctive Temporal Problem (DTP). We analyzed each algorithm experimentally on random problems and showed that the algorithms for DTPs with preferences integrate practically into a planning and scheduling application called Autominder. Before integration, we modified the algo-

rithms to handle *dynamic* DTPs with preferences, equipping them to solve a series of related DTPs with preferences faster than solving them individually.

In many real-world problems, finding the optimal solution is not necessary—it is more important to find near-optimal solutions quickly. Consequently, our focus was on maximizing anytime performance. We achieved this goal, showing empirically that our algorithms quickly found high-quality solutions even for large problems. For all algorithms, we showed that the cost of adding preferences was minimal, achieving our overall goal of making the use of TCSPs with preferences practical in all situations where hard-constraint TCSPs apply.

CHAPTER I

Introduction

Automated reasoning techniques increase the ability of computers to carry out complex tasks. As computational power decreases in cost and as the interface between people and computers becomes more intuitive, the opportunities for integrating automated reasoning techniques into our daily activities increase. With the ability to delegate more complex tasks to computers comes increased productivity; therefore, research in different subfields of automated reasoning has significant potential for positive real-world impact.

One important subfield of automated reasoning is *constraint reasoning*. This field focuses on tasks such as dividing resources among a set of people or agents; managing schedules for businesses, people, or robots; and verifying that plans and simulations satisfy the constraints of the real world. In general, the tasks involve a set of decisions to be made and a set of constraints that restrict which decisions or combinations of decisions are allowed. What differentiates one constraint reasoning problem from another is the type of decisions and the form of the constraints on those decisions.

Techniques for constraint reasoning have been applied to real-world problems. In some cases, the techniques have resulted in substantial monetary savings. For

example, airlines use constraint reasoning techniques to schedule flights, assign crews to flights, and assign flights to gates [16]. Constraints for this problem come from FAA regulations, union rules, and the physical limitations of planes and airports. While it is possible for people to make these decisions, computer algorithms can systematically explore the space of possible decisions and can therefore produce a set of decisions that maximize safety and revenue while minimizing costs. In other cases, constraint reasoning techniques complete tasks that people cannot perform in a reasonable amount of time. For example, it is nearly impossible for people to verify that modern integrated circuit designs, which contain millions of elements, do not violate physical constraints.

This thesis focuses on a particular type of constraint reasoning problem, that of deciding when a set of events can occur given a set of constraints and preferences on when those events can occur. The events refer to the beginning and end of actions that must be performed, e.g., the start of a commute or the end of lunch. The constraints limit the times at which an event can occur (e.g., the commute must start between 7am and 8am), the durations of actions described by the events (e.g., the start-time and end-time of lunch should be separated by at most an hour), and when multiple events can occur relative to each other (e.g., lunch must end at least an hour before swimming). Preferences encode which times or relative times are better than others (e.g., the commute should start as close to 7 as possible). Given the events, constraints, and preferences, the task is to assign to each event a time that satisfies each constraint and maximizes with respect to the preferences.

Most people use the ability to reason about events and the types of constraints just mentioned on a daily basis. We use this sort of reasoning to schedule our daily tasks, to coordinate the use of a shared vehicle, or to determine when to start

cooking each dish while making dinner. Such reasoning is natural and fairly easy if the situation contains a small number of tasks and a small number of people to coordinate. As the complexity and size of the situation increases, however, it may be more efficient to delegate the problem to a computer that, despite its own limitations, can perform the reasoning faster and possibly better.

In recent years, researchers have developed computational tools and techniques for effectively reasoning about time and temporal constraints. Thus, we already have the ability to delegate many temporal constraint reasoning tasks to computers. In general, each of these new techniques have limits in one or both of the following dimensions: the type and range of reasoning problems it can solve, and the size of the problems it can solve efficiently. Research in this area typically aims to break through these limits in one of two ways: to develop techniques for solving a new type of reasoning problem, or to devise a more efficient method of solving a pre-existing problem. In this research, we have done the former, creating techniques that solve three new types of constraint reasoning problems, all of which involve scheduling events in a way that satisfies temporal constraints and maximizes preference concerning those events.

1.1 Temporal Constraint Satisfaction

Our work builds on recent advances involving Temporal Constraint Satisfaction Problems (TCSPs) [23]. TCSPs are a subclass of the more general Constraint Satisfaction Problems (CSPs) [50, 56, 21], which consist of two elements: variables, which are to be assigned some value from a domain, and constraints, which restrict and allow certain combinations of assignments to a subset of the variables. In TCSPs, the variables are events, and the domain for each event is the set of real numbers

representing time. The constraints in TCSPs exist in one of several forms, which we will later discuss in detail.

Several TCSP subproblems have been used successfully in recent years as components in planning and scheduling applications such as Autominder [64] and NASA's Mars Rover [55]. In planning and scheduling applications, the events in TCSPs represent actions to be scheduled or executed by an agent, and the constraints specify allowable times and temporal differences between the events. The main task for these domains, when given a TCSP, is to assign to each event a time that respects all constraints.

The Autominder system, which provided the motivation for this work, uses TCSPs to manage the daily plans of people with mild memory impairment. Autominder reasons about the plan, the environment, and the user to determine when or if to issue reminders about important steps in the plan, hopefully enabling the user to continue to live independently. The plans managed by Autominder contain actions and constraints that can be encoded as a TCSP.

TCSP solvers are then used to find acceptable times (and ranges of time) for each action in the plan. As the user moves and acts within their environment, Autominder uses sensor information to update the constraints of the TCSP, possibly requiring the TCSP reasoner to reconsider which times are acceptable for each action.

Over the last decade, efficient algorithms have been developed for reasoning about a TCSP and its subproblems, which include Simple Temporal Problems (STPs), binary TCSPs (bTCSPs) [23][1] and Disjunctive Temporal Problems (DTPs) [78, 79]. We postpone a formal definition of the subproblems, which are named based on the

[1] In [23], binary TCSPs were simply called TCSPs. We use bTCSPs for these and reserve TCSP for the general class of problems. Some authors refer to the general class using lower-case letters (temporal constraint satisfaction problems) and refer to the specific subproblem using capital letters.

form of the constraints that compose them, and simply point out that they are listed in order of increasing complexity. Recent work addresses the problems of finding solutions to TCSPs (i.e, finding a valid set of assignments) [23, 78, 79, 3]; checking TCSPs for consistency (i.e., the existence of a solution)[14]; and flexibly executing the steps in plans described by such problems [54, 80, 81].

Only recently, though, has effort been made to increase the expressiveness of TCSPs. The two major extensions include uncertainty, which is the modeling of stochastic or uncontrollable events [86, 53, 84, 81]; and preference, which is the representation of soft constraints in addition to the hard constraints typical in TCSPs [39, 40, 89, 61, 62]. We focus on adding local preferences to TCSPs with the general aim of making the addition of local preferences practical for applications in which TCSPs are already practical. Where previous work has addressed the addition of restricted forms of preferences to STPs, this work contributes methods for adding unrestricted local preferences to the constraints in STPs, bTCSPs, and DTPs.

1.2 Preferences in TCSPs

The need for preferences stems from the limitations of hard constraints. Hard constraints in TCSPs specify the allowable temporal distances between events in a problem. For example, $X_2 - X_1 \in [5, 10]$ specifies that event X_2 must occur 5-10 time units after X_1. Any other temporal distance violates the constraint and represents complete failure. In reality, the violation of a constraint may not represent complete failure; it may simply indicate a less desirable situation.

Because hard constraints often do not reflect reality, a knowledge engineer specifying bounds on time differences must reason about a trade-off: a wider set of bounds increases the likelihood that the entire set of constraints is consistent, while narrower

bounds keeps actions closer to their ideal times. Therefore, specifying good bounds for a constraint requires knowledge about the real-world situation it models *and* how the bounds will affect the rest of the problem. By allowing the knowledge engineer to directly express preferences—to designate some bounds as more or less desirable than others—this trade-off is avoided. The expert can define a constraint without regard to its affect on the TCSP as a whole.

Adding preferences to a TCSP involves assigning to each constraint a preference function that maps each temporal distance value to a preference value. When temporal constraints contain preference functions, the problem is changed from simply finding a consistent assignment for events to finding an assignment with optimal value. The value of an assignment is defined by a function that aggregates each constraint's *local preference value*, which in turn is calculated by the constraint's preference function. The aggregation function, called the objective function, can vary: different objective functions require different algorithms of varying complexity.

Two types of aggregation functions are found often in the temporal preference literature: maximin[2] [39, 40, 61] and utilitarian [52, 62]. With the maximin function, the value of the entire assignment is equal to the *lowest* of all local preference values. The maximin function often permits very efficient algorithms for finding the optimal assignment. With the utilitarian function, the value of the entire assignment is equal to the *sum* of all local preference values. Algorithms with utilitarian optimality functions tend to have much greater complexity than their maximin counterparts, making algorithms that find approximate (i.e., potentially suboptimal) solutions necessary for practical use.

[2] "Maximin" has also been termed "Weakest Link Optimality" [39].

1.3 Overview of Work

In this research, we developed, analyzed, and tested algorithms for solving TCSPs with Preferences (TCSPPs). We studied the general problem and several of its special cases that are of practical interest for planning and scheduling applications. The space of problems we considered has three dimensions: hard constraint type, preference function properties, and objective function. We considered: hard constraint types found in STPs, bTCSPs, and DTPs (described fully in Section 2.2); unrestricted, semi-convex, and convex preference functions; and the maximin and utilitarian objective functions. Different combinations of these attributes require distinct algorithms to fully exploit available structure. Table 1.1 summarizes the space of problems this work addresses.

Dimension	**Range**
Hard constraint type	STP, bTCSP, DTP
Preference function	unrestricted, semi-convex, convex
Objective function	maximin, utilitarian

Table 1.1: The space of problems addressed by this research.

We name these problems primarily based on their hard constraint type, e.g., STPs with Preferences (STPPs) and DTPs with Preferences (DTPPs). Where necessary we include the preference type, e.g., STPPs with semi-convex functions.

Parts of this space have already been explored, particularly the parts corresponding to finding optimal solutions of STPPs with restricted preference functions. This research explored the remaining parts of the space, which apply to Autominder and similar applications.

Solving DTPs is known to be NP-Complete, and adding preferences to the DTP formalism only increases the complexity. Therefore, there exists a danger that even if the DTPP algorithms developed exploit all possible structure in the problem (i.e.,

the algorithm that finds the optimal solution the fastest), the algorithms still may not be useful in practical applications. Therefore, we adopt the following requirement for each DTPP algorithm: for applications in which the use of DTPs is feasible, the use of DTPPs must be feasible as well.

The problem space in Table 1.1 outlines several well-defined constraint optimization problems. Each problem takes as input sets of events, constraints and preferences and outputs a set of assignments or ranges of assignments. The main concerns when designing algorithms for these problems include completeness, speed, solution quality, and (as mentioned above) anytime performance. An additional concern, in the context of real-world applications such as Autominder, is that the algorithms can adapt to small changes in the problem. Specifically, the algorithms should be able to use information learned solving one problem to help it more quickly solve a modified version of the same problem. Therefore, for all of the algorithms we define, we show how to make them dyanmic—that is, we show the algorithms can be modified to adapt to a sequence of changes to the original problem.

1.4 Hypotheses

When this research was proposed, we listed a set of hypotheses we planned to test. In this section, we repeat the list. In Chapter VII, we report the result of each hypothesis.

1. Greedy search is effective for finding high-quality, approximate utilitarian solutions to STPPs.

   (a) For STPs with *semi-convex* preference functions and less than 10 preference levels per function, greedy search will find solutions with a value of greater than 80% of optimal for 95% of problem instances.

(b) For STPs with *unrestricted* preference functions and less than 10 preference levels per function, greedy search will find solutions with a value of greater than 70% of optimal for 95% of problem instances. This property will hold in cases where the number of local minima in each preference function is less than 5.

We use 10 preference levels because the resolution the number provides is more than sufficient for our Autominder domain, whose functions are specified by people. Larger numbers of preference levels affects our algorithms, but not dramatically. We chose 5 as the maximum number of local minima in the functions because we believe it is an upper bound for the number of local minima in any realistic preference function. In Autominder, this value rarely exceeds 1.

2. Finding maximin optimal solutions to DTPPs requires only slightly more effort than solving DTPs with hard constraints in the worst case. Before stating the precise hypotheses, we need to define a few terms. First, our algorithm requires that the range of preference functions be discretized into a finite number of bins, which we call *preference levels*. Second, we define a *level DTP* of a DTPP, which is a DTP constrained to only include solutions with a maximin value greater than some specified preference level. Thus, if the preference function range is discretized into L levels, there will exist at most L level DTPs.

   (a) For DTPs with *semi-convex* preference functions, the *additional* cost of finding a maximin optimal solution over finding a solution to the level DTP at the *solution level* is cubic in the number of events in the DTPP for the worst case.

(b) For DTPs with *unrestricted* preference functions, the *additional* cost of finding a maximin optimal solution over finding a solution to the *largest* level DTP is cubic in the number of events in the DTPP for the worst case.

3. Although finding utilitarian solutions to DTPPs has extremely high worst-case complexity, an anytime algorithm can be constructed that is clearly superior to a leading algorithm that solves DTPs without preferences [79]. Specifically, there will be no situation in which the DTP algorithm outperforms the DTPP algorithm.

4. The DTPP algorithms (maximin and utilitarian) can be made *dynamic*, meaning that the modified algorithm can solve a sequence of related DTPPs faster than the unmodified algorithm can solve the individual DTPPs separately.

1.5 Contributions

This work makes the following contributions, most of which follow directly from the specific hypotheses above:

1. Greedy-search heuristics that find approximate utilitarian solutions to STPs with *semi-convex* and *unrestricted* local preference functions. The analysis of these heuristics identifies which heuristics perform best and at what cost. Before our work, all research concerning preferences in TCSPs has focused on preference functions that are either convex or semi-convex. (Chapter III).

2. An anytime and complete algorithm for finding exact utilitarian-optimal solutions to STPs with *unrestricted* local preference functions (Chapter III).

3. An algorithm to find maximin optimal solutions to DTPPs. This algorithm allows the low-cost addition of preferences to applications for which DTPs with hard constraints are practical (Chapter IV).

4. An anytime and complete algorithm for finding utilitarian-optimal solutions to DTPPs using STPP heuristics (Chapter V).

5. Data structures and algorithms for solving and updating dynamic DTPPs, where constraints change as time progresses. Information from previous searches is leveraged to quickly find new optimal solutions after one or more constraints change (Chapter VI).

6. An analysis of relevant algorithms on a real world application, Autominder (Chapter VI).

Although the concept of using preferences in TCSPs was not introduced in this work, we were the first to add preferences to DTPs and the first to handle unrestricted preference functions in any form of TCSPP. We believe the primary reason these problems had not been attempted earlier involved efficiency concerns: the extra expressiveness, while desired, is often not worth the high computational cost that accompanies it. We addressed this concern by developing algorithms that have only incremental costs over current algorithms (contribution 3), by developing heuristics that find approximate solutions quickly (contributions 1,2,4, and 5), and by utilizing the common structure that exists in problem sequences found in the real world (contribution 6).

In general, these contributions will enable temporal reasoners used in planning and scheduling applications to extend their representational power. We demonstrate this in Chapter VI, where we incorporate the dynamic version of the DTPP algo-

rithms into Autominder.

The overarching result of this work is that adding preferences to TCSPs is now practical in situations where their hard constraint counterparts are practical. Autominder provides an excellent case study for evaluating the tangible results of this work.

1.6 Examples

To motivate the need for preferences we describe an example from the domain of Autominder, a system that currently uses DTPs (the most expressive hard TCSP subproblem) to manage the daily plans for persons with memory impairment [64]. One driver of the work reported in this dissertation has been the need to express preference when defining the daily plans for Autominder's intended users, a process that highlights the trade-off navigated by a knowledge engineer when forced to specify constraints with hard bounds.

We will use three situations to illustrate the use and need for STPPs and DTPPs. The first two are derived from domains in which Autominder operates, while the third is based on the Mars Rover application [52]. The first presents an STP with semi-convex preference functions, the second is a DTPP, and the third is an STP with unrestricted preference functions.

1.6.1 STP with semi-convex preference functions

An elderly woman must exercise for 25 uninterrupted minutes in late afternoon (3-5pm) shortly after taking heart medication, and before a friend visits from 3:45 to 4:15. It is best if the exercise ends well before the visit to allow her to recover. The task is to model this scenario with a TCSP and use it to determine when to take the medicine and when to begin exercising. To represent this situation, we use a

Simple Temporal Problem (STP) with six events: a temporal reference point (TRP), which is used for stating absolute (clock-time) constraints; an event representing the TakeMeds event (assumed instantaneous); and events representing the start and end of the Exercise and Visit actions. Notice that an event corresponds to a single point in time (i.e., it has no duration).

Figure 1.1 shows an STP encoding of this example. This representation is considered an STP (and not a binary TCSP or DTP) because each constraint in the problem restricts the difference between two events to lie within a single interval. This distinction will be elaborated in Section 2.2. The first constraint states that TakeMeds must occur 5 to 20 minutes before ExerciseStart, the second expresses when ExerciseEnd can occur relative to the visit, the third constrains the starting time of the visit, and so on.

Events	
TRP (3pm)	TRP
TakeMeds	T
ExerciseStart	E_S
ExerciseEnd	E_E
VisitStart	V_S
VisitEnd	V_E

Constraints
$E_S - T \in [5, 20]$
$V_S - E_E \in [5, \infty]$
$V_S - TRP = 45$
$V_E - V_S = 30$
$T - TRP \in [0, \infty]$
$E_E - E_S = 25$

Figure 1.1: STP encoding of the Autominder example.

This STP is not a particularly interesting one. Finding an assignment to all events that satisfies the constraints is fairly easy by inspection or by using the algorithms described in Section 2.3. However, two of its constraints help motivate the need for preferences in TCSPs.

Consider the first constraint between events T and E_S. We assume its parameters are chosen by a medical expert who understands the effects of the medicine in relation to exercise. In the STP, the interval $[5, 20]$ represents the allowable time difference

between the TakeMeds event and starting the Exercise action. Any other difference, say 4, 21, or -30, is not allowed and represents an inconsistency or failure in the STP. Note that in this STP, a difference of -30 is no worse than a difference of 4; both are considered failures of equal degree.

Instead of choosing hard bounds, the doctor may want to express that the ideal time difference between TakeMeds and ExerciseStart is 12 and that the widest bound possible is the interval $[0, 24]$. Such a preference can be represented, for example, as a piece-wise linear function shown in Figure 1.2(a).

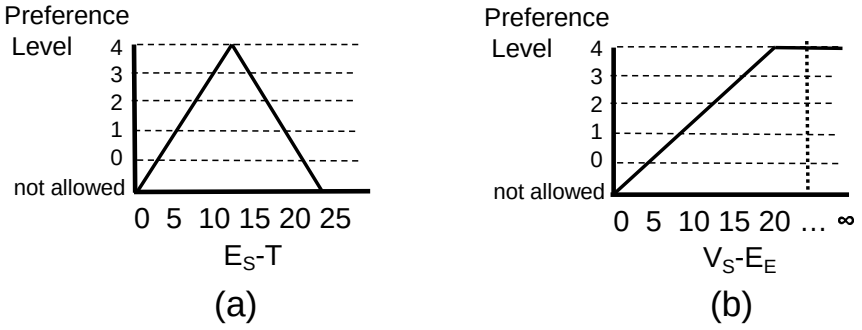


Figure 1.2: Two preference functions for two STPP constraints in the Autominder STPP example. (a) The preference function for the constraint between the TakeMeds event and the start of Exercise step. (b) The preference function for the constraint between the end of Exercise and the start of Visit.

As stated in the introduction, specifying hard bounds on time differences requires the expert to reason about a trade-off between increasing the likelihood that the entire STP is consistent and keeping actions closer to their ideal times. Expressing preferences avoids the trade-off: the expert can express the local relationship without regard to its effect on the problem as a whole.

A second constraint, between the events E_E and V_S, can also be improved by adding a preference function. The constraint between E_E and V_S specifies that exercise must end at least 5 minutes before the visitor arrives. What it does not

specify is that higher values for this temporal difference are preferable over lower ones, which, in our example, corresponds to the woman wanting as much time to recover from exercising as possible. Using a preference function, we can elaborate the constraint to say that finishing at least 20 minutes ahead or more is most desirable, while finishing barely before the visitor arrives is the least preferable possibility. A function that expresses this is shown in Figure 1.2(b).

1.6.2 DTP with preference functions

Our example of a Disjunctive Temporal Problem (DTP) extends the previous example into one more typically found in Autominder. We change the constraint that requires exercise to finish at least 5 minutes before the visitor arrives to allow the exercise action to start *after* the visitor leaves as well. To model this situation, we need the ability to express disjunctions in the constraints. Specifically, we want to express that exercise must end at least 5 minutes before the visit begins *or* exercise must begin anytime after the visit ends. The result, which is a DTP constraint, is as follows:

$$V_S - E_E \in [5, \infty] \vee E_S - V_E \in [0, \infty]. \tag{1.1}$$

Two more constraints now need to be added: the first to prevent the TakeMeds activity from occurring during the visit:

$$V_S - T \in [0, \infty] \vee T - V_E \in [0, \infty], \tag{1.2}$$

and the second to make sure exercise finishes by 5pm:

$$E_E - TRP \in [-\infty, 120]. \tag{1.3}$$

After the addition of the constraint in (1.1), we need to redefine the preference function for the modified constraint to express how desirable the second option (exercising after the visit) is relative to the first. To express preferences in a DTP

constraint, we define a preference function for each option, or disjunction, in the constraint. In this case, we can leave the function defined for the first option unchanged, and say that the second option has a preference of 2, which makes it equally desirable to finishing exercise 10 minutes before the visit. For completeness, we show the functions for both disjuncts in Figure 1.3.

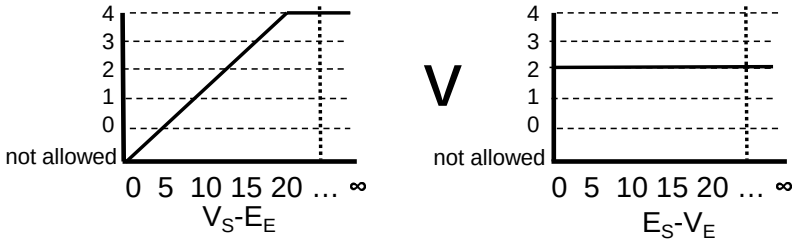


Figure 1.3: Two preference functions for a single DTPP constraint in the Autominder DTPP example.

The constraint in (1.2) is a special type of constraint that occurs often in planning applications, called a *non-overlapping constraint*. Such constraints are really qualitative, not quantitative, but we represent them quantitatively.

In this example, we will choose to not define a preference function for constraints (1.2) and (1.3). Not defining a preference function is equivalent to having a preference function in which every temporal distance has an infinite preference value or zero preference value, depending on the nature of the objective function.

In this example, the preference functions were piece-wise linear functions, mapping intervals to each preference level. In general, however, any function, discrete or continuous, that maps temporal differences to preference values is acceptable. Our algorithms convert all functions into a canonical form that allows all preference function types to be processed in the same way.

1.6.3 STP with unrestricted preference functions

We now describe a very simple example of an STPP with non-convex and non-semi-convex functions based on the Mars rover domain [52]. For this situation, two events need to be optimally scheduled: the start- and end-time of a single experiment (events S and E). The experiment must begin some time after the instrument it requires becomes available (event A, set to time 0). Although the experiment can start immediately once available, it is preferable that some time separates A and S to allow the instrument to cool. This preference is expressed as MC1 (Mars constraint 1) in Figure 1.4(a); the horizontal axis represents the difference between events A and S, while the vertical axis represents the local preference value of each temporal difference.

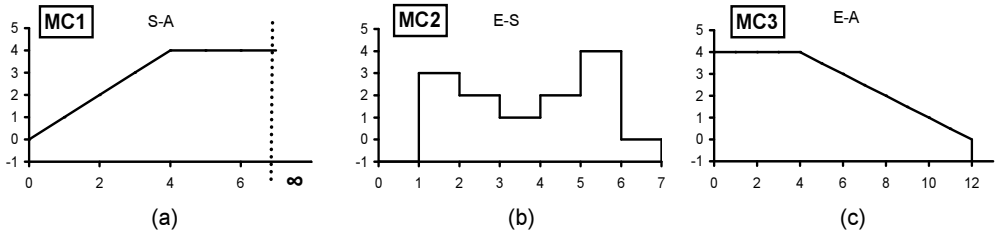


Figure 1.4: Example preference functions for the Mars Rover example. The horizontal axes represents the temporal difference between two events, while the vertical axis represents the local preference value of each temporal difference.

The scientific value of the experiment increases irregularly with time spent running it (E-S), but is mitigated by the significant power usage of the instrument. The net value is expressed by the preference function in Figure 1.4(b). Finally, since other experiments can begin once the current experiment ends, it is preferred that the experiment finish as early as possible. The function in Figure 1.4(c) expresses this relationship.

A legal schedule in this example is one in which $E < 12$ and $1 \leq E - S < 7$; that is, one in which all preference functions map to a nonnegative value. For a schedule $\{A = 0, S = 3, E = 6\}$, constraints MC1, MC2, and MC3 have respective preference values 3, 1, and 3, for a utilitarian value of 7. An optimal schedule, with a utilitarian value 10, is $\{A = 0, S = 4, E = 5\}$.

1.7 Organization of Dissertation

This thesis is organized by the algorithms that constitute the main contributions of our work. In Chapter 2, we introduce formalisms, concepts, and algorithms related to the remaining chapters. The chapter briefly surveys work in temporal reasoning; defines the Temporal Constraint Satisfaction Problem (TCSP) and several of its special cases; gives an overview of existing algorithms for solving TCSPs and TCSPs with preferences; and gives an overview of soft constraints in non-temporal CSPs.

Chapter 3 describes the problem of finding utilitarian-optimal solutions to STPs with unrestricted preference functions. We present two algorithms for solving this problem: one simple approach based on greedy search, called `STPP_Greedy`; and an extension to that approach that is complete, called `GAPS`.

Chapter 4 describes the problem of finding maximin-optimal solutions to DTPs with unrestricted preference functions. We present our solution to this problem, called `DTPP_Maximin`, and show that its worst-case complexity barely exceeds that of algorithms for solving DTPs with hard constraints.

Chapter 5 describes the problem of finding utilitarian-optimal solutions to DTPs with unrestricted preference functions. Our solution to this problem, called `GAPD`, borrows heavily from the techniques and lessons learned in Chapters 3 and 4.

Chapter 6 shows how the `DTPP_Maximin` and `GAPD` algorithms from Chapters 4 and

5 can be integrated into the Autominder application. We discuss how Autominder actually solves a series of DTPPs, called a dynamic DTPP, and discuss modifications to `DTPP_Maximin` and `GAPD` that improve their efficiency for solving dynamic DTPPs.

An empirical analysis is given in Chapters 3-6 that characterize the performance of the respective algorithms. The analysis in Chapter 6 shows that the algorithms created can be practically applied to real-world problems.

Finally, Chapter 7 summarizes this work by restating the contributions and by analyzing the results for the testable hypotheses. The chapter includes a brief discussion of a few general lessons learned in this study and a few directions for future work.

CHAPTER II

Background

This chapter introduces formalisms, concepts, and algorithms related to the work in this dissertation. We start with a brief survey of work in temporal reasoning. Section 2.2 defines the Temporal Constraint Satisfaction Problem (TCSP) and several of its special cases. Section 2.3 gives an overview of existing algorithms for solving TCSPs and provides significant detail for some concepts and techniques used within the algorithms we developed. Section 2.4 describes how to augment TCSPs with preference functions, discusses different aggregation functions useful in finding optimal solutions, and justifies why we have chosen to focus on two particular aggregation functions. Section 2.5 covers existing algorithms for solving TCSPs with preferences. We finish the chapter with an overview of soft constraints in non-temporal CSPs, including recent work on qualitative preferences.

2.1 Temporal Reasoning

Reasoning about time is a central task in AI research. Past research in temporal reasoning is often divided into two main categories: reasoning about action and change using logical formalisms, and reasoning about temporal constraints. In one respect, the work in this thesis spans this dividing line, since we are using tempo-

ral constraints to reason about the actions in a plan. However, our work directly descends from the temporal constraint literature.

A good survey on the field as a whole can be found in [13].

2.1.1 Logic-based Temporal Reasoning

Much of the initial work in temporal reasoning was motivated by the need to express change and the effects of action in logical formalisms such as propositional logic and first-order logic. The situation calculus (SC) [46] extends first-order logic by adding an argument to each predicate that indicates the *situation* in which the predicate applies. A situation is the state of the world at a particular time. Elements of the world model that vary their state by situation, called fluents, provide a mechanism for expressing actions and their instantaneous effects.

The difficulty of expressing which fluents do or do not change after the execution of an action became known as the *frame problem* [51, 33]. Addressing the frame problem became the focus for later work on SC and related formalisms [59, 18, 74, 67].

SC essentially folds time into a logical formalism, defining special arguments and predicates to encode time without affecting the underlying logical formalism. A method that gives time special status is *reification* [2], which defines special time-related predicates that accept propositional formulas as arguments. For example $HOLDS(\psi, I)$ designates that the formula ψ holds over the time interval I.

An alternative to SC is the Event Calculus [41], which explicitly models the *changes* between situations rather than each situation. The world state is represented as a set of properties (analogous to fluents) whose values change in response to events. In EC, properties persist, holding their value until changed by some event, thus avoiding the frame problem. EC associates events and properties to particular

time intervals [41] or time points [71] rather than to more abstract times as done by SC. The result is that EC defines events in realistic linear time, whereas SC defines a partial order of situations, allowing hypothetical situations to be modeled. Since the representations of EC and SC are orthogonal, attempts have been made to unify the two [63, 66].

A more thorough survey of logic-based temporal reasoning can be found in [83].

2.1.2 Constraint-based Temporal Reasoning

The second branch of temporal reasoning deals more directly with time. Temporal entities such as time points and time intervals are used to represent the occurrence of events, the duration of actions, and times at which some fact holds true. Constraints over the temporal entities define the relations between the events or actions the entities represent. The field of constraint-based temporal reasoning is usually divided between approaches involving qualitative constraints and metric (quantitative) constraints. Underlying both of these categories is the Constraint Satisfaction Problem (CSP).

Constraint Satisfaction Problems

CSPs [21] are defined by a set of variables V, a set D containing a domain for each variable, and a set of constraints C defining acceptable combinations of assignments to each variable. The domains in D can be discrete and finite (e.g., $\{red, blue, green\}$), discrete and infinite (e.g., all integers), or continuous (e.g., all reals). Each constraint in C is defined by a set of variables $S(C) \subseteq V$, called the *scope* of the constraint, and a relation $R(C)$ that defines whether each possible assignment to the variables in $S(C)$ is acceptable.

Given a CSP, the task is to find some set of assignments to the variables that sat-

isfy all constraints. Finding such an assignment is usually accomplished through some combination of constraint propagation and search. Constraint propagation involves combining existing constraints to infer new constraints or to eliminate assignment combinations from existing constraints. Search is usually executed by sequentially assigning a value to each variable until some constraint is violated; then, backtracking occurs and new assignments are tried. The backtracking search is aided by pruning techniques and heuristics that order the assignments in a way that more quickly leads to a solution.

A thorough discussion of CSPs, constraint propagation, and search has been published by Dechter [21]. Particular techniques related to the algorithms in this thesis are presented in Section 2.3.

Qualitative constraints

The most notable instance of qualitative constraint-based temporal reasoning is Allen's Interval Algebra (IA) [1]. IA defines a set of 13 basic relations that can exist between two time intervals: *before, after, meets, met_by, overlaps, overlapped_by, starts, started_by, during, contains, finishes, finished_by, equals*. In this algebra, intervals are mapped to CSP variables and the binary constraints between two intervals define an allowable subset of the 13 relations. For example, two intervals I_1 and I_2 could have the constraint $C_{12} = \{before, meets, overlaps\}$. Given a set of intervals and a set of constraints, the task is to place the intervals on a time-line in a way that satisfies the constraints.

Solving an IA problem is NP-complete [88], so simpler algebras began to appear. The Point Algebra (PA) [87] uses time points as the principal time entity and defines three possible relations between two time points: $\{precedes, equals, follows\}$. Be-

cause this algebra is less expressive (cannot express <>), polynomial-time constraint propagation algorithms (path consistency) can solve the underlying CSP.

Other algebras, including van Beek's Continuous Endpoint algebra [82] and Nebel and Burchert's ORD-Horn algebra [57] similarly allow tractable solvers at the expense of some expressive power.

Quantitative constraints

The first work on quantitative constraint-based temporal reasoning was the Time-Map [20], which is a graphical representation of binary constraints between events. The nodes of the graph represent events, while the edges represent constraints over the temporal difference between the times at which the events occur. This concept was expanded and formalized into a quantitative temporal algebra by Dechter, Mieri, and Pearl [23]. This algebra defines the set of allowable time differences between two time points. The set is organized into a set of intervals, as in $X_j - X_i \in \{[a_{ij1}, b_{ij1}], ..., [a_{ijn}, b_{ijn}]\}$, where X_i, X_j are time points. This algebra led to the Temporal Constraint Satisfaction Problem, which underlies much of the work in this dissertation. Section 2.2 gives details on this problem and associated algorithms.

2.1.3 Temporal Reasoning under Uncertainty

Just as time was first encoded implicitly into logical formalisms, so was time first encoded implicitly into probabilistic formalisms. Time was encoded into Bayesian Networks [58] by creating random variables whose values represented time points or intervals. Approaches in this line included discrete Time Nets [35], Temporal Nodes Bayesian Networks [5], Probabilistic Temporal Networks [72], Berzuini's Network of Dates [6], and the continuous Time Net [36].

Other formalisms give time first-class status by creating sequences of non-tempor-

al random variables, as in Markov chains, Hidden Markov Models, and Dynamic Bayesian Networks [19]. The addition of time in these cases is analogous to the way Situation Calculus adds time to first-order logic.

Recently, uncertainty has been addressed in constraint-based temporal reasoning approaches. A special case of the Temporal Constraint Satisfaction Problem, the Simple Temporal Problem (STP), has been extended to handle uncertain and uncontrollable temporal constraints [53, 81, 89, 69].

2.2 Temporal Constraint Satisfaction Problems

We now formally define three subclasses of TCSPs: Simple Temporal Problems (STPs), Disjunctive Temporal Problems (DTPs), and binary Temporal Constraint Satisfaction Problems (bTCSPs).

2.2.1 Simple Temporal Problem

The most restricted common subclass of TCSPs is the Simple Temporal Problem (STP). An STP is a pair $\langle X, C \rangle$, where X designates a set of variables corresponding to events, and C is a set of m binary temporal constraints. Each variable in X has as its domain the set of real numbers representing time, so the semantics of assigning value d to variable x is that event x should occur at time d. Each constraint in C limits the assignment of times to exactly two events to respect the following relationship: $x - y \in [a, b] : x, y \in X; a, b \in \Re$.

Notice that in our STP example (Figure 1.1) all constraints are of this form. All STPs can be represented by a network where each event is a node, each arc is a constraint, and each arc-label is the allowable temporal difference for a single constraint. Figure 2.1 shows the network that represents our example STP.

A *solution* to an STP (or any TCSP) is an assignment of a time to each event

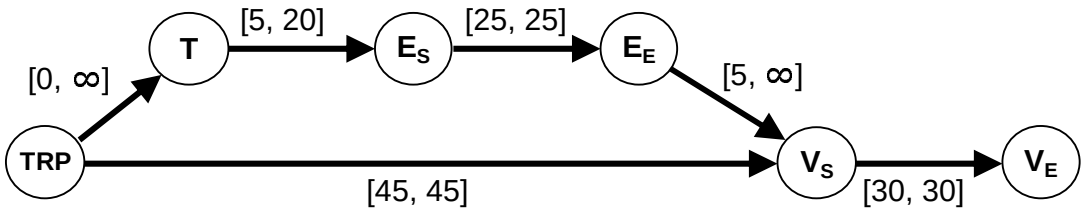


Figure 2.1: A network representing the example STP defined in Figure 1.1.

that satisfies all constraints. For our example, the assignment $a = \{TRP = 0; T = 0; E_S = 5; E_E = 30; V_S = 45; V_E = 75\}$ is one of many solutions to the STP depicted in Figure 2.1. We refer to the set of all solutions to an STP S as $sol(S)$. We write a solution to STP S as $\{x_i = d_i : x_i \in X\}$. An STP is said to be *consistent* if at least one solution exists. Conversely, an STP is *inconsistent* if there is no assignment that will satisfy all constraints.

Consistency-checking is the process of determining whether a solution exists without necessarily finding that solution. Consistency-checking in an STP can be cast as an all-pairs shortest path problem (APSP) in the corresponding network: the STP is consistent if there are no negative cycles in the all-pairs graph. This check can be performed in $|X|^3$ time using any APSP algorithm.

A by-product of this particular consistency-checking method is the *minimal network*. A minimal network is the tightest form of an STP that contains all solutions; one STP is as tight as another if each constraint in the first represents a subset of the temporal differences in the second. Since this concept is integral to many algorithms in this dissertation, we define it formally:

Definition 1 (Minimal Network)**.** *A* **minimal network** *for STP $S = \langle X, C \rangle$ consists of an STP $M(S) = \langle X, M \rangle$, where $M = \{M_i : M_i$ is as tight as $C_i\}$. $M(S)$ is the tightest form of S that allows all solutions to S: $sol(M(S)) = sol(S)$. Tightening*

any constraint in $M(S)$ would cause $sol(M(S)) \neq sol(S)$.

The key property of a minimal network is that every temporal difference in a minimal network constraint participates in some solution of the STP. In other words, for any minimal network constraint M_i and any value $d \in [c_i, d_i]$, there exists some solution in which $x_i - y_i = d$. Thus, tightening any constraint would exclude at least one solution.

A single solution can be extracted from the minimal network in $|X|^2$ time [23].

2.2.2 Binary Temporal Constraint Satisfaction Problem

A binary Temporal Constraint Satisfaction Problem (bTCSP) extends an STP by allowing multiple intervals in binary constraints:

$$x - y \in \{[a_1, b_1], ..., [a_n, b_n]\}.$$

This expression constrains the distance between x and y to be in one of the intervals listed. The extra expressiveness comes at a high cost: determining consistency for a binary TCSP is NP-hard [23]. Note that a bTCSP constraint can be viewed as a disjunction of STP constraints over the same two events: $x - y \in [a_1, b_1] \vee x - y \in [a_2, b_2] \vee \ldots \vee x - y \in [a_n, b_n]$.

In the DTP example in Section 1.6.2, we do not have any binary constraints with more than one interval. For an example of this case, imagine that the elderly woman's visitor will arrive at either 3:45 or 4:15, depending on whether the person has already seen a particular episode of Matlock. In this case, the constraint in question would become $V_S - TRP \in \{[45, 45], [75, 75]\}$. Such constraints can still be represented as a graphical network, where the arc-labels are sets of intervals rather than a single interval.

It is useful to note that if for every constraint in a bTCSP, a single interval is selected (i.e a single disjunct), the set of selected intervals form an STP. The resulting STP is called a *component STP* of the bTCSP, and is a key construct in algorithms for solving bTCSPs.

2.2.3 Disjunctive Temporal Problem

A Disjunctive Temporal Problem (DTP) [78] removes the binary restriction of bTCSPs, resulting in constraints with the following form:

$$(x_1 - y_1 \in [a_1, b_1]) \vee (x_2 - y_2 \in [a_2, b_2]) \vee \cdots \vee (x_n - y_n \in [a_n, b_n]).$$

The bTCSP is a subcase of the DTP in which $x_1 = x_2 = \ldots x_n$, and $y_1 = y_2 = \ldots y_n$. Hereafter, we will write disjunctive constraints as a set of STP constraints (the 'or' will be implied).

To satisfy a DTP constraint, only one of its disjuncts must be satisfied. A solution to a DTP is an assignment that satisfies at least one disjunct in each of its constraints. The concept of component STPs is key to algorithms that solve DTPs as well.

DTPs are useful in Planning and Scheduling applications because they can represent the so-called *promotion/demotion constraints* that occur frequently. A promotion/demotion constraint expresses situations in which one process cannot be running at the same time as a second process. The first process either must finish before the second process begins (promotion) or it must start after the second process ends (demotion). This is similar to the relationship between the exercise action and the visit in our example. In addition, plans and schedules often express that activities can occur at multiple times or that multiple activities can be assigned to a single time slot; disjunctive constraints model both of these situations well.

2.3 Overview of TCSP Algorithms

We now sketch some of the basic algorithms for the chief operations on TCSPs.

2.3.1 Checking consistency of and solving STPs

To check the consistency of an STP is simply to determine whether a solution exists, while solving an STP can have one of several meanings: to find a single satisfying assignment, to find all satisfying assignments, or to find the minimal network, which is a compact representation of all satisfying assignments. Finding the minimal network is the default meaning of "solving an STP" in the planning and scheduling context because the minimal network represents a flexible schedule—a range of allowable times for each event represented.

Recall from above that the minimal network is the tightest form of an STP that contains all solutions in the original problem. To produce the minimal network, each temporal difference in each constraint that does not participate in some solution is removed.

The most common method for finding the minimal network involves calculating an alternate representation called a *d-graph*, which is a complete directed graph where each node is an event of the STP and each arc is the smallest temporal distance between the events it connects. Since the minimal network contains a subset of the information in the d-graph, computing the d-graph for an STP essentially computes the minimal network. The d-graph represents the all-pairs shortest path distance between events in the STP and can be calculated using standard shortest path algorithms, including the `Floyd-Warshall` ($O(|X|^3)$) and `Bellman-Ford` ($O(|X||C|)$) algorithms [17]. When the STP does not have a solution (i.e., it is inconsistent), the shortest-path algorithm will find a cycle with a negative path length.

Other algorithms solve STPs using constraint satisfaction techniques, such as path-consistency and arc-consistency, which are sufficient for determining consistency in TCSPs. `PC-1` and `PC-2` [45] repeatedly enforce path-consistency among triples of events, tightening the temporal distance between events until the minimal network (d-graph) is found. `PC-1` is equivalent to `Floyd-Warshall`, producing the same result and requiring the same amount of time. Both of these algorithms have best-case performance equal to that of the worst case. `PC-2` has the same worst-case performance, but improves on the average case.

AC-3 and AC-3cc

Several consistency algorithms using arc-consistency have been developed as well. These algorithms perform worse than path-consistency algorithms in the worst case, but often do better on average. A well-known arc-consistency algorithm is `AC-3` [45]. Although `AC-3` applies to general CSPs, it produces stronger results when applied to STPs: it proves consistency or inconsistency. In Chapters III and V, we use a modified version of the `AC-3` algorithm that is suitable for propagating small changes to the STP [12]. Therefore, we present this modified algorithm, which we also call `AC-3`, in Figure 2.2.

`AC-3` works by maintaining an interval for each event, called the *assignment window*, that represents the allowable times to which it can be assigned. One special node, called the *temporal reference point*, starts with a time window of $[0, 0]$, while the others start with $[-\infty, \infty]$. The algorithm continuously shrinks the assignment windows while processing a set of constraints, ensuring that the time window for each event is consistent with all constraints that affect it. When any time window becomes invalid (i.e., the upper bound is less than the lower bound), the STP is known to be

```
AC-3(new_arcs)
    // each arc ∈ new_arcs is defined by ⟨from, to, lb, ub⟩
    // each node maintains a window of allowable assignments ⟨LB, UB⟩
    queue = new_arcs
    WHILE (queue not empty )
        current-arc = Pop(queue);
        IF Revise(current-arc)
            modNodes = all nodes whose bounds were modified by Revise
            queue =  queue ∪  modNodes \ current-arc
        ELSE
            RETURN INCONSISTENT
        END
    END
    RETURN CONSISTENT

Revise(arc)
    // increase LB of To node if necessary
    arc.To.LB = max(arc.To.LB, arc.From.LB + arc.LB)
    // decrease UB of To node if necessary
    arc.To.UB = min(arc.To.UB, arc.From.UB + arc.UB)
    // increase LB of From node if necessary
    arc.From.LB = max(arc.From.LB, arc.To.LB - arc.UB)
    // decrease UB of From node if necessary
    arc.From.UB = max(arc.From.UB, arc.To.UB - arc.LB)

    // check for inconsistency
    IF (arc.To.LB > arc.To.UB) or (arc.From.LB > arc.From.UB)
        RETURN false
    ELSE
        RETURN true
```

Figure 2.2: `AC-3`, an incremental STP consistency-checking algorithm [45, 12].

inconsistent since no assignment to that event is possible. If the intervals cannot be further shrunk, then the algorithm exits.

As input, `AC-3` is given a set of constraints, called `new_arcs`, that are to be propagated through the network. If the goal is simply to check the consistency of a particular STP, then the entire set of constraints in the STP is given as input. However, the algorithm can be called with only a subset of the constraints given as input to check whether the addition of new constraints causes inconsistency. This

ability to propagate constraints individually is valuable when the constraints in the problem change over time. Each time a constraint tightens, it alone must be given as input, and will produce the same result as if all constraints had been propagated at once. The greedy search algorithms presented in Chapter III capitalize on this ability.

In the worst case, each call to the `Revise` method will reduce the interval of a single event by one unit of time. The complexity of `AC-3` is $O(h|C|)$, where h is the temporal horizon defined by the largest upper bound of any event. While the worst-case complexity can be greater than the algorithms described previously, `AC-3` runs very fast in practice when solutions exist. However, when the network is inconsistent and conditions are not favorable, the running time can be very close to the worst case. The unfavorable condition occurs on certain STPs that are "barely" inconsistent. Figure 2.3 illustrates such a condition.

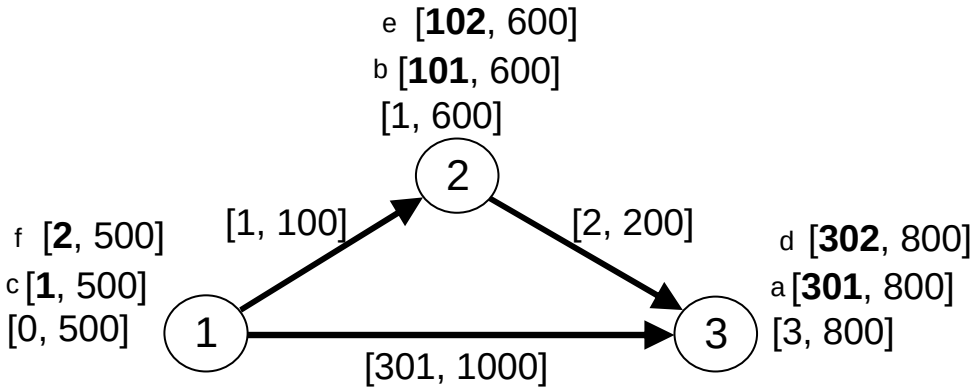


Figure 2.3: A simple example of an inconsistent STP that causes looping in the `AC-3` algorithm. The constraints and the initial event bounds are given; the letters *a-f* indicate the order in which the event bounds are updated during `AC-3`. Notice how the bounds of each event (i.e., events 1,2, and 3) tighten by only 1 unit after each revision. (reproduced from [12])

Notice that the simple 3-event STP in Figure 2.3 is trivially inconsistent: The upper bound of arcs $1 \rightarrow 2$ and $2 \rightarrow 3$ sum to 300, which is less than the lower bound

of arc $1 \rightarrow 3$. However, `AC-3` will not discover the inconsistency until it processes each constraint many times. Notice that the time window of event 1 starts at [0,500] and only reduces by 1 each time it is processed: it will require close to 500 revisions of each constraint before `AC-3` detects the inconsistency.

Fortunately, the unfavorable condition can be detected and circumvented by maintaining two additional structures during the execution of `AC-3`: the *upper dependency graph* and the *lower dependency graph*. Each dependency graph maintains a record of which constraints "caused" the lower or upper bound of each time window. The graphs contain a node for each event and at most 1 outgoing arc from each node. If an event's time window has not yet been modified, there is no arc emanating from the event's node. However, as soon as the lower (upper) bound of the event's time window has been modified by the Revise method in `AC-3`, an arc is created in the lower (upper) dependency graph from the node to the node that caused the modification.

The purpose of the dependency graphs is to detect cycles similar to the one illustrated in Figure 2.3. Any time such a cycle exists, there will be a corresponding cycle in one of the dependency graphs. In Figure 2.3, the cycle was in the lower dependency graph, as evidenced by the repeated modification of all lower bounds.

Checking for cycles in the dependency graph requires time linear in the number of events, since at most 1 arc eminates from each node. A cycle check is performed at the end of the Revise function, adding complexity to the algorithm, but dramatically descreasing average case performance. We call the resulting algorithm `AC-3` with cycle check (`AC-3cc`). The original authors did not give it a name.

2.3.2 Checking consistency of and solving DTPs

Recall that a DTP or bTCSP can be viewed as a collection of component STPs, where each component STP is formed by selecting one STP constraint (i.e., one interval) from each bTCSP or DTP constraint. Given this viewpoint, checking the consistency of the full DTP amounts to searching for a consistent component STP. Each solution to a component STP will also be a solution to the complete DTP or bTCSP as well. This search can be cast as a meta-CSP problem, where each variable is a disjunctive constraint, each value is a single STP constraint within the disjunctive constraint, and the implied constraint is one that allows only sets of disjuncts that form consistent STPs.

This formulation underlies the `Forward and Go-Back` algorithm of Dechter *et al.* [23]. Although this algorithm was originally designed to solve bTCSPs, it applies directly to DTPs as well, since it does not exploit the binary property of the constraints in bTCSPs. An algorithm that does exploit the binary property of bTCSPs exists [14], but since the algorithm does not apply to DTPs, we do not describe it here. Consequently, in the reminder of the dissertation, we shall refer only to DTPs; the reader should keep in mind that everything we write also applies to the special case of bTCSPs.

The `Forward and Go-back` algorithm works by searching the space of *partial* component STPs within a given DTP, where a partial component STP consists of STP constraints extracted from a subset of the DTP constraints. It builds a partial STP one constraint at a time, adding a constraint if the current set is consistent, and backtracking otherwise, looking for complete component STPs that are consistent. (A component STP is complete if it includes a disjunct from every constraint in the original DTP.) The original version of the algorithm found *all* consistent component

STPs and returned the union of their minimal networks, which represents the most flexible schedule possible given the constraints in the problem. For many applications, however, it is not necessary to find the most flexible schedule of the entire DTP—it suffices to find the minimal network of a single consistent component STP, since it contains at least one solution to the DTP.

Figure 2.4 shows the `Forward and Go-back` algorithm modified in two ways. First, it returns the minimal network of the first consistent STP found. Second, the bookkeeping aspects of the algorithm have been modified to make it more closely resemble current DTP solvers (e.g., [79]). Because of these bookkeeping modifications, a line-by-line comparison of our version of the algorithm—called `Solve-DTP`—with the original one reveals few similarities. However, they search through the space of component STPs in the same order.

Solve-DTP(ci)

```
1.  newDisjunct = getNext(ci); // Choose next unprocessed disjunct in DTP constraint ci
2.  IF newDisjunct does not exist
3.      IF ci = 0 return failure // no solution exists
4.      reset(ci); //Mark all disjuncts in constraint ci as unprocessed
5.      RETURN Solve-DTP(ci-1); // Try next disjunct in previous DTP constraint
6.  END
7.  cSTP = cSTP ∪ newDisjunct
8.  IF Consistent-STP(cSTP)
9.      IF ci = |C| − 1 RETURN Minimal Network of selected STP
10.     RETURN Solve-DTP(ci+1); // Try first disjunct in next DTP constraint
11. ELSE
12.     cSTP = cSTP \ newDisjunct;
13      RETURN Solve-DTP(ci); // Try the next disjunct in this DTP constraint
14. END
```

Figure 2.4: `Solve-DTP`, a modified version of the `Forward and Go-back` algorithm [23].

The algorithm operates on a DTP represented as a list of m DTP constraints, each of which is a set of STP constraints. The algorithm maintains a partial component

STP, called `cSTP`, which always contains a set of disjuncts with at most one disjunct from each DTP constraint. Each call to the recursive algorithm selects a disjunct from a DTP constraint denoted by the constraint index *ci* and adds it to `cSTP`. This process continues until an added disjunct causes `cSTP` to be inconsistent or until the size of `cSTP` reaches m. When an inconsistency occurs, the new disjunct is removed and another one is tried. If `cSTP` has size m, `cSTP` represents a solution to the DTP and is returned.

The algorithm uses three internal functions: `Consistent-STP`, which checks `cSTP` for consistency; `getNext`, which retrieves an unprocessed disjunct from a DTP constraint; and `reset`, which marks all disjuncts in a constraint as unprocessed. Any of the aforementioned algorithms will work for the `Consistent-STP` function. The external call to `Solve-DTP` should pass in the integer '1' for *ci*, indicating that it starts the search with the first DTP constraint.

Line 1 of the algorithm chooses the next disjunct from the ci^{th} constraint. If all disjuncts from a constraint have already been tried (line 2), lines 3-5 check for the exit condition, reset the ci^{th} constraint and backtrack to try the next disjunct of the previous constraint.

If another disjunct does exist, Line 7 adds it to `cSTP`. Line 8 then checks the consistency of `cSTP`. If it is both consistent and complete (line 9), then a solution has been found. If consistent but not complete, then the algorithm is called recursively to find a disjunct from the next DTP constraint (line 10). If inconsistent, the new disjunct is removed from `cSTP` (line 12) and method is called again for constraint *ci* so that its next disjunct can be considered (line 13).

The algorithm's time complexity depends on the number of component STPs in the DTP, which in turn depends upon the number of DTP constraints and the

number of disjuncts in each. If the DTP contains $|X|$ events and m constraints with a maximum of k disjuncts per constraint, there exists $O(k^m)$ STPs to check for consistency. In the worst case, where no consistent STP exists, the total complexity is $O(k^m \cdot |X|^3)$.

The worst case is rarely achieved, however, because of pruning techniques that have been developed for CSPs and modified for use in DTPs, including Conflict-Directed Backjumping, Semantic Branching, Removal of Subsumed Variables, and No-good Recording [79]. Each of these methods can be incorporated with few modifications into the `Solve-DTP` algorithm, and, although we do not provide detailed explanation, they can similarly be adopted into `DTPP_Maximin`, which is described in Chapter IV.

2.4 Preferences in TCSPs

The idea of preferences in the context of temporal constraint satisfaction problems was introduced in [39]. This work defined the formalism and related it to previous work on soft CSPs, which are the more general (non-temporal) analog of TCSPs with Preferences.

To extend an STP to an STP with preferences (STPP), each constraint is augmented with a preference function that maps each value in the interval identified by that constraint to a *preference value*, a scalar measure of the value's desirability. Hence the constraints in an STPP have the following form:

$$\langle x - y \in [a, b], f \rangle$$

where

$$f : [a, b] \rightarrow \Re.$$

To extend a DTP to a DTP with preferences (DTPP), we simply add preference functions to the STP disjuncts within each STP constraint. Of course, preference functions are not required for *all* constraints. In practice, problems will most likely contain a mix of constraint types: STP constraints, STPP constraints, DTP constraints, and DTPP constraints. Constraints without preferences must be satisfied, but they do not contribute to the objective function. We design algorithms for the most complex case, in which all are DTPP constraints, but we ensure that when simpler constraints exists, the algorithms leverage the simpler structure.

As mentioned in the introduction, the addition of preferences changes the problem from a satisfaction problem to an optimization problem. The task is to find the set of assignments that maximize some objective function. We can evaluate an assignment with respect to a particular constraint using the constraint's preference function. For example, if we have the constraint $\langle X_j - X_i \in [a, b], f \rangle$, and an assignment $X_i = 10$ and $X_j = 22$, then the *local preference value* for the constraint is $f(12)$.

Definition 2 (local preference value)**.** *The* **local preference value** *for a disjunct* $C_{\langle k,d \rangle} \equiv \langle X_j - X_i \in [a, b], f \rangle$ *under some solution* $S = \{X_0 = t_0, X_1 = t_1, \ldots, X_m = t_m\}$ *is* $v_{\langle k,d \rangle}(S) = f(t_j - t_i)$*. The* **local preference value** *for a constraint* $C_k \equiv \bigvee_d C_{\langle k,d \rangle}$ *is* $v_k(S) = \max_d v_{\langle k,d \rangle}(S)$*.*

To evaluate an entire solution, we aggregate the local preference values in a way that suits the particular application. Given some binary, associative, and commutative aggregation operator $\times$, optimally solving a TCSP with preferences requires a search for the assignment S that maximizes (or minimizes) the following aggregation function:

$$\bigtimes_{k \in C} v_k(S).$$

In general, different aggregation operators require different algorithms. However, a given algorithm may require only a set of properties in an aggregation operator (e.g., that it is idempotent), and can therefore be applied when maximizing any aggregation operator that embodies those properties [7].

In this dissertation, we assume that a preference value of zero represents the lowest possible preference that is still feasible. Higher values represent stronger preference.

2.4.1 Types of Aggregation

The local preference functions provide the basis of our aggregation function by mapping solutions to a local preference value for a single constraint. What remains is to aggregate the local preference values into a single value representing the overall desirability of a particular schedule. The choice of aggregation functions has been studied extensively in other contexts under labels including "social choice" [4] and "multiattribute utility" [38]. In general, this work aims to define criteria that aggregation functions must meet in order to be consistent and fair and to define functions that meet this criteria. Here, we briefly survey several aggregation functions found in the literature.

The first aggregator we describe is the minimum function, which sets the global value of a solution to the minimum value of any local utility (e.g., local preference value). Optimization using this function is termed *maximin* because the goal is to maximize the minimum local utility.

A problem with maximin arises because a given problem often contains many solutions that have optimal value. With respect to "minimum" objective function, each of these solutions is equivalent. But there are cases in which some "optimal" solutions are obviously better than others. For example, consider a three constraint

TCSP containing two maximin optimal solutions of value 4: s1: $\{4, 8, 9\}$, and s2: $\{4, 4, 4\}$. In most straightforward interpretations, s1 is preferable to s2 because each local preference value in s1 is as good or better than the preference values in s2. In other words, s1 *dominates* s2.

Definition 3 (Dominated solutions)**.** *A solution s1* **dominates** *another solution s2 iff (1) for each constraint in the TCSP the local preference value for s1 is greater than or equal to the local preference value for s2 and (2) for at least one constraint the local preference value for s1 is greater than the local preference value for s2.*

When given a set of optimal solutions (maximin or other) to a given problem, all solutions dominated by another solution in the set can be removed from the set. The resulting set contains the *Pareto optimal* solutions to the TCSP. A Pareto optimal solution is one in which increasing the local preference value of any constraint will require a decrease in some other constraint's local preference value to remain consistent.

Stratified egalitarian (SE) optimality combines the maximin optimality criterion with the concept of Pareto optimality [52]. SE-optimal solutions are the subset of maximin optimal solutions that are not dominated by any other maximin optimal solution.

A third function is the utilitarian function, which sets the global value of a solution to the sum of all local utilities. This often-used function makes sense when the individuals' only purpose is to serve a global good. In our Autominder application, for example, each constraint influences the schedule for a single individual.

When some individual entities are more important than others, a weighted function of local preference values is appropriate. The utilitarian function is easily modified to include weights as factors for each local preference value. Alternatively,

weights can be added to a multiplicative function, as in the weighted product of powers function [76]:

$$f(u) = \left(u_1^{w_1} \cdot u_2^{w_2} \cdot \ldots \cdot u_n^{w_n}\right)^{-\sum_i w_i}.$$

Maximizing the multiplicative function is equivalent to maximizing the sum of the logarithms of the factors, which means that an algorithm that finds utilitarian optimal solutions will work for both the sum and multiplicative aggregation functions. We are mostly concerned about the case in which all individuals are of equal weight, so we do not address the weighted case.

Although the multiplicative function can be transformed into an additive one, there are situations in which the multiplicative function better fits the situation. For example, if the preference values represent independent probabilities of success, then the product of the local preference values would correspond to the probability of success for the entire problem. However, this has little relevance to the TCSPPs in this dissertation.

Another criterion used for preference aggregation is called *lexicographic* optimality [34]. To find a lexicographic optimal solution to a problem with several distinct atomic criteria (e.g., our local preference values), an ordering of the criteria is chosen and then each criterion is maximized in that order. The method works by first finding the set of all solutions that maximize the first criterion. Then, from that set, the set that maximizes the second criterion is found. This continues until all criteria have been maximized or until the set reduces to a single solution. This type of aggregation is useful when individual criteria are much more important than other criteria (e.g., safety is usually maximized before considering other choices.) Again, for the problems this work addresses, all local preference values (i.e., criteria) have

the same weight, so we will not focus on lexicographic optimality.

Two other criteria, *minmax regret* [44] and *competitive ratio* [11] use a *regret function* that characterizes the value of a solution relative to some maximum possible value. Although the specific definition of a regret function is in terms of states and actions, the form that matches our situation is simple: the regret of a single constraint regarding some solution s is a function of the constraint's local preference value, v_c, and the maximum possible preference value, v_cMax, of that constraint for any solution. For minmax regret, the regret function is $R_c(v_c, v_cMax) = v_cMax - v_c$, where for competitive ratio, the regret function is $R_c(v_c, v_cMax) = v_cMax/v_c$. In both cases, the goal is to find the solution that minimizes the total regret for all constraints. Intuitively, this corresponds to searching for the solution that upsets everyone the least. Given the simplified form of these criteria, both are isomorphic to the maximin aggregation function.

Since the goal of this work was not to further the study of what makes the best aggregation function, we chose to focus on two functions that have received attention in the most related literature and that make intuitive sense for our application. We focus first on the maximin function because it is so conducive to efficient algorithms. Previous work has shown that STPs with semi-convex preference functions can be maximin-optimally solved in polynomial time [39].

We focus second on the utilitarian function because it subsumes the multiplicative function and because of its applicability to Autominder and related systems in which all constraints are defined with respect to a common goal. The DTPs in Autominder represent the plan of an individual user, so it makes intuitive sense to maximize the total of all local preference values.

2.4.2 Properties of local preferences

In general, optimally solving even STPs with unrestricted preference functions (both maximin and utilitarian) is NP-Hard [39], so exploiting the properties of particular functions is necessary to design tractable algorithms.

In the introduction, we stated that we would be focusing on two types of restricted preferences: convex and semi-convex. A convex preference, in the context of soft constraints, is one that satisfies the following property:

$$f(\frac{1}{2}(t_1 + t_2)) \geq \frac{1}{2}(f(t_1) + f(t_2))^1.$$

A semi-convex preference is one in which the set $\{x : f(x) > l\}$ forms at most a single interval for any value l. A semi-convex preference satisfies the following property:

$$f(t_1) = f(t_2) \Rightarrow \forall t \in [t_1, t_2], f(t) \geq f(t_1).$$

The semi-convex property is useful for an algorithm that efficiently finds a maximin solution to STPPs [39].

Other more restrictive properties of preference functions may enable very efficient algorithms, but may not apply to many situations. These include square functions (equivalent to a weighted TCSP), step functions, tent functions, Gaussian functions, or piecewise linear functions.

Restrictions can also be made on the location of preference functions. A polynomial algorithm for finding utilitarian-optimal solutions to STPPs exists when the preference functions are restricted to exist only on constraints between the temporal reference point and another event [42]; in other words, when preferences exist only on the absolute times for each event and not the relative times. Although this for-

[1]This is actually the definition of a concave function, but is considered a convex preference because it can be represented as a convex relation.

malism is a simple restriction of STPPs, the author gave it a new name: extended STPs.

The need for unrestricted preference functions arise most often in the form of *repulsive preferences*, which indicate a preference that some time or set of times should be avoided. For example, when describing the preferred time for driving to work, someone may express that "the drive should be as far away from 8am as possible".

2.5 Existing STPP Algorithms

We now briefly review two previous algorithms for solving STPPs. The first algorithm we describe finds maximin optimal solutions to STPs with semi-convex preference functions [39]. The algorithm, called `WLO` (for Weakest Link Optimality), works by choosing some preference value α and then projecting the STPP constraints onto an STP constraint that contains the temporal differences with preference α or higher. If the resulting STP is consistent (checked with a standard STP algorithm), then a solution with maximin value α is found. Using a binary search through all possible α, the optimal solution is found. If the range of the preference functions is discretized into a set A, the binary search through preference levels will find the consistent STP with the highest preference level in $O(\log(|A|) \cdot |X|^3)$ time.

As stated before, the problem with maximin optimality is that there are often many maximin solutions which are not Pareto optimal. To address this problem, the same authors developed a modification of the algorithm that finds the stratified egalitarian optimal solutions [40]. This modification, however, requires that the preference functions be convex, not just semi-convex.

The modified algorithm, called `WLO+`, finds the Pareto set of optimal solutions

by using multiple calls to WLO. After each WLO iteration, a special set of constraints called the Weakest Link Constraints (WLC) are identified. Each element of WLC is satisfiable only at the optimal level found by WLO. The WLC set is replaced with constraints that restrict the STPP to the current set of solutions and are given preference functions that map all temporal differences in the constraints to the highest preference level. The STPP is then ready for the next iteration. The algorithm iterates until all constraints have been members of WLC or until the STP returned by WLO represents a single solution. When all preference functions are convex, WLC is guaranteed to be nonempty at each iteration. Each iteration removes a set of dominated solutions from the minimal network.

We briefly mention that when all preference functions are convex and piecewise linear, the STPP can be represented as a linear program and solved by highly optimized LP solvers [52]. This formulation of the problem leads to a utilitarian optimal solution. Morris et al showed empirically that the WLO+ algorithm produced solutions that were on average within 90% of the utilitarian optimal solution, suggesting that WLO+ was a feasible option for computing near-optimal utilitarian solutions in polynomial time.

2.6 Expressive Power of STPPs, bTCSPPs, and DTPPs

From the definitions in Section 2.2, it is clear that an STP is a special case of a bTCSP, which itself is a special case of a DTP. When preferences are added, however, the lines between these formalisms blur. For example, consider the STPP constraint in Figure 2.5(a). If the area below level L (indicated as area C) is removed, the remaining areas A and B form a two-disjunct bTCSPP constraint shown in Figure 2.5(b).

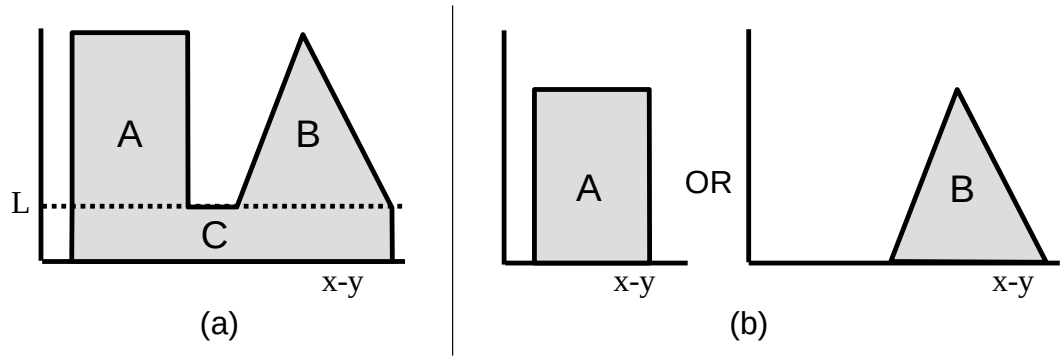


Figure 2.5: An STPP constraint that is a bTCSPP constraint when chopped at level L.

Likewise, the bTCSPP constraint in Figure 2.5(b) can be converted into an STPP constraint by adding a "false bottom" to the function, producing the constraint in Figure 2.5(a). Any bTCSPP solution that satisfies the bTCSPP constraint in Figure 2.5(b) with a local preference value of v will satisfy the corresponding STPP constraint at value $v+L$. Conversely, any solution that satisfies the STPP constraint in Figure 2.5(a) with a local preference value $q > L$ will satisfy the corresponding bTCSPP constraint at value $q - L$.

Using this conversion strategy, we can show that a bTCSPP can be searched for maximin-optimal solutions using an STPP algorithm. The proof (and the two following it) assume without loss of generality that the lowest feasible preference value for all constraints is 0 and that each preference function returns -1 for all values outside of its domain (i.e., all infeasible values).

Theorem II.1. *The problem of finding maximin-optimal solutions to bTCSPPs can be cast as a problem of finding maximin-optimal solutions to STPPs.*

Proof. Given a bTCSPP with constraints of the form $\bigvee_d \langle x-y \in [a_d, b_d], f_d \rangle$, such that $\forall p\ b_p < a_{p+1}$, create for each bTCSPP constraint an STPP constraint $\langle x-y \in [a, b], f \rangle$, where $a = \min_d a_d$, $b = \max_d b_d$, and $f(t) = 1 + \max_d f_d(t), \forall t \in [a, b]$.

By definition, any solution S to the STPP with maximin value of q must satisfy all STPP constraints at level q. Specifically, for each STPP constraint $\langle x - y \in [a, b], f \rangle$, $f(x - y) \geq q$ given S. If $q > 0$, then $\max_d f_d(t) > -1$, which means exactly one disjunct of the corresponding bTCSPP constraint is satisfied at level $q - 1$ (recall that all disjuncts in a bTCSPP constraint represent non-overlapping regions, since $\forall p \ b_p < a_{p+1}$). Therefore, any STPP solution with maximin value of $q > 0$ is a solution to the original bTCSPP with maximin value $q - 1$. □

Using a similar construction, we can prove the same for the utilitarian case.

Theorem II.2. *The problem of finding utilitarian-optimal solutions to bTCSPPs can be cast as a problem of finding utilitarian-optimal solutions to STPPs.*

Proof. Given a bTCSPP with constraints of the form $C_k \equiv \bigvee_d \langle x - y \in [a_d, b_d], f_d \rangle$, such that $\forall p \ b_p < a_{p+1}$, create for each bTCSPP constraint a set of STPP constraints (one for each disjunct) $C_{\langle k,d \rangle} \equiv \langle x - y \in [a_d, b_d], g_d \rangle \, , \forall d$, where

$$g_d(t) = \left\{ \begin{array}{ll} M + f_d(t) & \text{if } f_d(t) \geq 0 \\ 0 & \text{otherwise} \end{array} \right\} \forall t$$

and M is some constant that exceeds the maximum possible utilitarian value for the bTCSPP.

For any solution S to the bTCSPP, exactly one disjunct in each constraint will be satisfied because, again, the disjuncts are mutually exclusive. Consider a bTCSPP constraint C_k in which disjunct d is satisfied at level $f_d(x - y) = v$ given S. Since $f_d(x-y) \geq 0$ given S, $g_d(x-y) = M+v$ for the STPP constraint $C_{\langle k,d \rangle}$. For the STPP constraints that correspond to the remaining disjuncts in C_k, $g_d(x - y) = 0$ given S. Therefore, given S, the sum of all local preference values for STPP constraints created from C_k is $M + v$. If there are m constraints in the bTCSPP and the

utilitarian value of S is u, then the utilitarian value of S in the corresponding STPP is $m \cdot M + u$.

Any solution S to the corresponding STPP with utilitarian value $q \geq m \cdot M$ will have a utilitarian value of $q - m \cdot M$ in the original bTCSPP because exactly one STPP constraint derived from each bTCSPP constraint will be satisfied. For the set of STPP constraints that correspond to a single bTCSPP constraint, having more than one concurrently satisfied in S is impossible since the disjuncts are mutually exclusive; having zero satisfied is of course possible, but the utilitarian value of S would reduce to a value less than $m \cdot M$. □

The proof to Theorem II.2 exploits the fact that all disjuncts in a bTCSPP constraint are mutually exclusive. However, the proof does not exploit the fact that the constraints are binary. Therefore, DTPPs in which all constraints have mutually exclusive disjuncts can be reduced to STPPs as well.

Theorem II.3. *The problem of finding utilitarian-optimal solutions to DTPPs whose constraints have mutually-exclusive disjuncts can be cast as a problem of finding utilitarian-optimal solutions to STPPs.*

Proof. Proof to Theorem II.2, mutatis mutandi. □

Theorem II.3 addresses a subclass of DTPPs that is very important to this dissertation. We motivated the need for DTPPs with the need for promotion/demotion constraints that are common in planning and scheduling applications. Since promotion/demotion constraints have mutually exclusive disjuncts, many planning and scheduling problems fall into this class.

The theorems in this section show that when we allow unrestricted local preference functions, the distinction between the three problems is less clear than in the

non-preferences case. Although the analysis may suggest that we only need STPP algorithms for all cases, in practice there are reasons to develop separate algorithms. First, for the cases not covered above, including finding maximin-optimal solutions to DTPPs and finding utilitarian-optimal solutions to non-mutually exclusive DTPPs, separate algorithms are clearly needed. Second, in practice problems described as STPPs with unrestricted local preferences will often have significantly fewer disjunctions than problems described as DTPPs. STPP algorithms (like the ones in this dissertation) that exploit this fact would not work well on DTPPs converted in the manner described above. Third, note that the conversions presented above work by adding significant amounts of infeasible search space to the STPP. The added space is marked infeasible by adding a constraint on the solution value (e.g., utilitarian value must be greater than $m \cdot M$).

2.7 Soft CSPs

Soft CSPs are a generalization of TCSPPs in the same way that CSPs generalize TCSPs. Therefore, several concepts that arise when finding the optimal solutions for TCSPPs exist in soft CSPs as well. For example, in the soft CSP (SCSP) literature, different algorithms and analyses exist for different aggregation functions. In fact, they usually define the problem based on the function used: weighted CSPs use utilitarian optimality; fuzzy CSPs usually use maximin optimality; probabilistic and possibilistic CSPs use the multiplicative function; and lexicographic CSPs use lexicographic optimality [47].

A seminal paper on soft CSPs defines a *c-semiring*, which is a special semiring that can represent all SCSPs and associated aggregation functions just mentioned [7]. The paper defines each constraint as a tuple $\langle def, con \rangle$, where con is the set

of variables included in the constraint (its scope, according to [21]), and def is a mapping from all possible assignments to the variables in *con* to a value in the set A. A c-semiring is a five-tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, where A is a value set such that $\mathbf{0}, \mathbf{1} \in \mathbf{A}$, $+$ is an additive operator used to intersect the def's of multiple constraints with common variables, and $\times$ is a multiplicative operator used to aggregate the defs when combining constraints. The authors in [7] prove many properties based solely on the properties of the c-semiring, allowing any type of SCSP to be categorized and matched with similar SCSPs. Using this information, properties discovered in a single type of SCSP often easily transfer to others. Most of the useful generalizations made in this paper apply to aggregation operators ($\times$) which are idempotent, such as the minimum function. Little is proved about non-idempotent aggregators, such as the sum function.

Table 2.1 lists several common soft CSP types and the associated c-semirings.

Name	A	$+$	$\times$	**0**	**1**
Classic CSP	$\{false, true\}$	$\vee$	$\wedge$	false	true
Fuzzy CSP	$[0, 1]$	max	min	0	1
Weighted CSP	$\Re+$	min	sum	∞	0
Probabilistic CSP	$[0, 1]$	max	$\times$	0	1

Table 2.1: Common soft CSPs and associated c-semirings (compiled from [7]).

Even the classic CSP (with no soft constraints) fits into the c-semiring framework: the value set contains only "false" and "true". The operator $\wedge$ ensures that the combination of any set of constraints is "true" only if every constraint in the combination is "true".

Fuzzy CSPs [70, 24] use a c-semiring that corresponds to the maximin optimality referenced often in this dissertation. Weighted CSPs [43] combine constraint values using the sum function, resulting in a utilitarian evaluation of a solution. However,

the constraint values in weighted CSPs usually refer to costs, which means the goal is to minimize the sum of values. In our work, the goal is to maximize the sum.

Probabilistic CSPs [27] are slightly different in that the "softness" refers to the existence of the entire constraint, rather than in the particular value combinations of the constraints. Each constraint c applies only with probability p_c. The probability of a particular schedule being invalid is the combined probability of all constraints it violates.

There exist other types of soft CSPs, such as Possibilistic CSPs [25] and Lexicographic CSPs [28], but they have no relevance to this dissertation.

A recent article has formally described which types of binary soft CSP problems have tractable solutions and which do not [15]. Since STPs contain only binary constraints, this work is applicable to this dissertation. It confirms what was already known about STPs: finding a maximin-optimal solution is tractable when preference functions are semi-convex, and finding a utilitarian-optimal solution is tractable when the functions are convex.

General solution techniques applicable to other SCSPs are applicable to TCSPPs as well. Branch-and-bound techniques are common for algorithms that systematically search for optimal solutions (e.g., [70]), while local search and genetic algorithms are used to find possibly suboptimal solutions. Approximate solution techniques also find suboptimal solutions, but maintain an upper bound, giving an indication of how close the found solution is to optimal [47].

Qualitative Preferences

The soft CSPs just described encode preference in much the same way as do TCSPPs. Because of the structure of TCSP constraints, preference can be repre-

sented as simple functions, whereas most soft CSP formulations generally must be described in tables. In both cases, however, the values assigned to each assignment combination in a constraint must be a metric value, which can be difficult to elicit in practice. To account for this problem, recent work has focused on representing and reasoning about qualitative preferences, such as "I prefer red wine to white wine". Given the lack of quantitative values, it is not possible to evaluate a given solution to the CSP on its own merits; its value can only be expressed in terms of relative value—whether it is preferred over other solutions.

The formalism underlying most of the work on qualitative preferences is the CP-net [8, 9]. The CP-net provides a graphical representation of relative preferences and conditional preferences. Inference on a CP-net yields an assignment to all variables that is Pareto optimal, but does not give a quantitative evaluation of the assignment.

The CP-net has been extended to the multi-agent case [29, 68], the case in which cycles exist in the graph encoding the preferences [31, 65], and the case in which variables have differing levels of importance [10].

CHAPTER III

Finding Utilitarian-optimal Solutions to STPPs

Although the problem of finding maximin-optimal solutions to STPPs has been solved or at least studied, solving them using utilitarian optimality is fairly unexplored territory. The only portion of the space previously studied is the case in which the preference functions are both convex and piece-wise linear [52]. In this chapter, we remove these restrictions and describe two algorithms that search for utilitarian optimal solutions to STPPs with arbitrary local preference functions.

The first algorithm, called `STPP_Greedy`, is an extremely fast algorithm that uses greedy search techniques to find solutions whose utilitarian value average over 80% of the value of the optimal solution. We describe several heuristics useful in the greedy search and experimentally show which perform best on randomly generated networks. We also show how the heuristics can be combined to improve results.

The second algorithm, called the Greedy Anytime Partition algorithm for STPPs (`GAPS`) [62], is a complete, anytime, and memory-boundable algorithm that uses `STPP_Greedy` as an inner call. We demonstrate the superb anytime properties of `GAPS` and show that it finds optimal solutions faster than a branch-and-bound solution to the problem. The results show that `GAPS` is well-suited for practical applications requiring STPPs: it finds high-valued solutions very quickly and finds the optimal

solution in time comparable to an algorithm not designed for anytime performance. While GAPS itself has little value for our Autominder application (Autominder requires the disjunctive constraints found in DTPs), it provides the core of the GAPD algorithm, which is described in Chapter V.

We begin the chapter by describing two key concepts useful for understanding the material in this and later chapters: *preference projections* and *component STPs*. Then we describe STPP_Greedy, its associated heuristics, and GAPS. We then describe a branch-and-bound algorithm and use it in evaluating STPP_Greedy and GAPS, showing their suitability to practical applications. We conclude our experimental evaluation with a comparison of GAPS to a recent SAT-based algorithm for solving STPPs and DTPPs.

3.1 Preference Projections and Component STPs

The idea of preference projections and component STPs are introduced in this chapter, but will be extended in Chapter IV for use in the DTPP algorithms as well. The concepts are essential to our algorithms and their understanding.

3.1.1 Preference Projections

Rather than operate directly on soft constraints, i.e., STPP constraints that include a preference function, algorithms in this chapter work with a construct we call *preference projections*. Preference projections are collections of hard constraints—normal STP constraints without preference functions—that are derived from an STPP constraint. In other words, a preference projection is a hard constraint representation of a preference function. By using preference projections, we can leverage standard, polynomial-time algorithms for solving STPs. While the specific definition is new, the general approach of projecting hard constraints from constraints with

preferences was introduced for an algorithm that found maximin-optimal solutions to STPPs [39] (see Section 2.5).

To obtain a preference projection for an STPP constraint, we first discretize its preference function range into a set of real values A, called a *preference value set* ($\{0, 1, 2, 3, 4\}$ in our Mars rover example). Then, we project the STPP constraint at each level $a \in A$ into a set of hard constraints. Finally, we organize each set of hard constraints into a tree which is termed the preference projection for that constraint.

The projection of soft (STPP) constraint C_k to preference level l is a list of hard (STP) constraints representing the intervals at which C_k is satisfied at preference level l or higher. Figure 3.1 shows the preference projection for each constraint in our Mars rover example. Each horizontal line segment in Figure 3.1(d-f) is an element of the preference projection. For example, the line segment at preference level 3 in Figure 3.1(f) denotes that if $0 \leq E - A \leq 6$, then the local preference value for constraint MC3 will be 3 or greater.

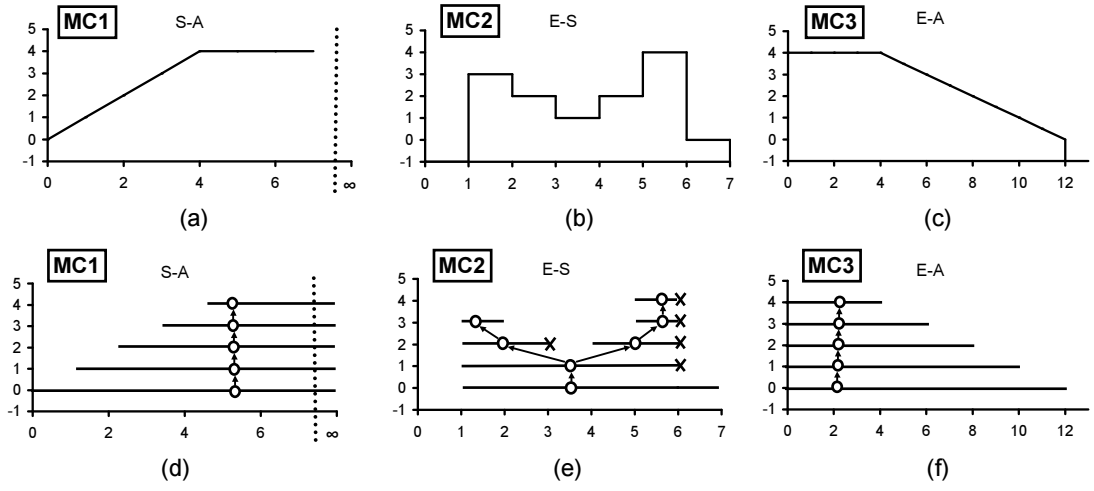


Figure 3.1: The preference projections for each constraint in the Mars rover example (see Section 1.6.3). Each line segment at level l represents temporal differences for which the value of the preference function is l or greater. "X" marks open intervals, and the circles and arrows represent a tree into which every interval is organized.

We identify each projected constraint with the 3-tuple $\langle k, l, p\rangle$, where k references the soft constraint from which it is derived, l is its preference level, and p is an index that distinguishes it from other constraints at the same level projected from the same constraint. The p element is needed for situations in which the projection breaks into more than one interval, as in levels 2 and 3 of MC2.

Definition 4 (Preference Projection)**.** *Given an STPP constraint* $C_k = \langle x-y \in [a_k, b_k], f_k\rangle$*, the projection at level* l *is* $\mathcal{P}_k[l] = \{C_{\langle k,l,1\rangle}, C_{\langle k,l,2\rangle}, \ldots, C_{\langle k,l,n\rangle}\}$*, where (1)* $C_{\langle k,l,p\rangle} = \langle x-y \in [a_p, b_p]\rangle$*, (2)* $b_p < a_{p+1}$ *for* $1 \leq p < n$ *and (3)* $\bigcup_{p=1}^{n} [a_p, b_p] = \{t | f_k(t) \geq l\}$*. We refer to the set of all hard constraints projected from a single STPP constraint* C_k *and preference value set* A *as the* **preference projection** *for* C_k*, denoted as* $\mathcal{P}_k = \bigcup_{l \in A} \mathcal{P}_k[l]$*.*

In Figure 3.1, circles and arrows show how each preference projection forms a tree. The relationships between the hard constraints in this tree are very important to the algorithms below, so a fair amount of notation is needed to make the explanations clear.

First, the root of the tree is always a single constraint at preference level 0. A constraint's child is always one preference level higher and represents an interval that is a subset of the parent constraint's interval. Given this subset relation, we say that a child is always as *tight* as or *tighter than* its parent. We denote the parent of constraint c as $par(c)$, the children of c as $ch(c)$ and the i^{th} child of c as $ch_i(c)$. The terms ancestor and descendant in this context have obvious meanings. The ancestors of constraint c are denoted by $anc(c)$ and the descendants as $des(c)$.

Definition 5 (Preference Projection Root)**.** *Given an STPP constraint* C_k *and its preference projection* $\mathcal{P}_k$*, the* **preference projection root** *is the single constraint*

in the set $\mathcal{P}_k[0]$, which is the projection of C_k at level 0.

The preference projection root is the ancestor of all hard constraints projected from a preference function. Given that every child is as tight as or tighter than its parent, the root is always the most relaxed constraint in the projection. It follows that, if some assignment to the constraint's two events does not satisfy the root, it will not satisfy any constraint in the projection. This property, which is exploited extensively in the algorithms below, can generalize to any member of the projection: if some assignment does not satisfy a hard constraint h, then no descendant of h will be satisfied. The converse of the property holds as well: if some assignment satisfies h, then it necessarily satisfies all ancestors of h.

3.1.2 Component STP

Given a set of preference projections, a hard STP called a *component STP* can be formed by selecting a single STP constraint from each STPP constraint's preference projection.

Definition 6 (component STP)**.** *A* **component STP** *is a set of m hard STP constraints, $\mathcal{C} = \{C_{\langle 1,l_1,p_1\rangle}, C_{\langle 2,l_2,p_2\rangle}, \ldots, C_{\langle m,l_m,p_m\rangle}\}$, where m is the number of STPP constraints. The utilitarian value of $\mathcal{C}$ is $u(\mathcal{C}) = \sum_{i=1}^{m} l_i$. The j^{th} constraint in $\mathcal{C}$, $C_{\langle j,l_j,p_j\rangle}$, is denoted by $\mathcal{C}_j$.*

Any component STP can be checked for consistency using one of several standard STP solvers mentioned in Section 2.3.1. If a component STP chosen from an STPP's preference projections is consistent (i.e., it has a solution), the solutions to the component STP are also solutions to the entire STPP. The value of each solution is guaranteed to be greater than or equal to the sum of the preference levels for

each constraint in the component STP. In other words, the utilitarian value of the component STP is a lower bound on utilitarian values of each schedule within it.

In each STPP, there exists a single distinguished component STP $ROOT = \{C_{\langle 1,0,1\rangle}, \ldots, C_{\langle m,0,1\rangle}\}$, which contains the root of each constraint's projection tree and is the most relaxed component STP. If $ROOT$ is consistent, then its solutions are guaranteed only to have value 0 or greater, since the preference level of each member of $ROOT$ is 0. If $ROOT$ is inconsistent, then the STPP has no solution. Conversely, the most restrictive and preferable component STPs contain constraints from the highest level of each tree; if consistent, no other consistent schedule for that STPP will have a higher value.

Given these extremes, we can cast our search for an optimal utilitarian schedule as a search for the highest-valued consistent component STP H, with value $u(H)$. Because the preference values are discretized, $u(H)$ may be up to $m \cdot \delta_A$ less than the value of the optimal schedule, where δ_A is the maximum distance between preference values in the preference value set A. When the preference functions are step functions and A contains a value for each step value, as in Figure 3.1(b), $\delta_A = 0$. However, for smooth functions, such as that in Figure 3.1(c), the search may produce slightly suboptimal answers due to the discretization.

Since we are searching for high-valued component STPs, the STPP search space is composed of all possible component STPs derived from the preference projections. Later, we discuss and use STPP *subspaces* which are defined as subsets (specifically subtrees) of each preference projection.

3.2 STPP_Greedy

Figure 3.2 presents `STPP_Greedy`, a simple algorithm for quickly finding a high-quality solution to an STPP. The idea is to search the space of component STPs starting with the lowest-valued one, *ROOT*, and repeatedly improving its value by replacing a single constraint with one of its children. The main function, called `replaceAConstraint` (line 4), picks a constraint, replaces it with one of its children, and returns the child's identifier (replacedChild) or NULL if no replacement is possible. If the replacement by a child leads to an inconsistent component STP (line 6), the child and its descendants are marked as "finished" and never selected again by `replaceAConstraint`.

```
STPP_Greedy(ROOT)
1.  IF ROOT not consistent RETURN ∅
2.  cSTP ← ROOT
3.  REPEAT
4.      replacedChild ← replaceAConstraint(cSTP)
5.      IF replacedChild is NULL, RETURN cSTP
6.      IF cSTP not consistent
7.          markAsFinished(replacedChild)
8.          replaceConstraintWithParent(replacedChild, cSTP)
9.      END IF
10. END REPEAT
```

Figure 3.2: The `STPP_Greedy` algorithm.

The condition in line 6 can be tested using any STP solver. In our implementation, we used the `AC-3cc` solver [12], which is designed for efficiently propagating small changes to an STP. See Section 2.3.1 for a description of this algorithm.

In the worst case, the algorithm calls propagate once for each projected constraint in each preference projection. In practice, of course, few propagations occur since (1) each time a child causes inconsistency, all its descendants are pruned and (2) each time a child successfully propagates, all of its siblings are pruned. This may

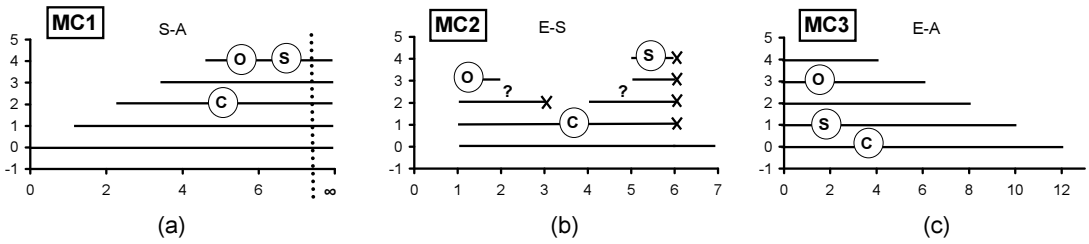


Figure 3.3: The constraints in the STPP example with multi-level preference functions. The horizontal lines represent the intervals at each preference level. 'C' denotes the current component STP before the heuristic choice is made. 'O' represents the optimal solution (value = 10), which is achievable if the constraint in MC2 is replaced with its left child. 'S' represents a suboptimal solution (value = 9), which is the best solution achievable if the heuristic replaces the constraint in MC2 with its right child.

seem unimportant given the low worst case complexity of `STPP_Greedy`, but it comes into play because this algorithm is called with great frequency in the `GAPS` algorithm described below.

The heuristic for the greedy search is located in the `replaceAConstraint` function on line 4. It chooses a constraint in the network that can possibly be replaced by its child one level higher. The value of `STPP_Greedy` depends greatly on the quality of the heuristics. To understand the effect of heuristics, we use consider a situation in which `STPP_Greedy` is part way through its search on the Mars rover STPP. Figure 3.3 illustrates this example.

Three circles marked 'C' in Figure 3.3 denote the "current" component STP maintained by `STPP_Greedy`, directly before the algorithm heuristically chooses which child should replace its parent. Any one of four constraints could be chosen to replace its parent in 'C': the lone child of the constraint in MC1; the two children of the constraint in MC2, or the lone child of the constraint in MC3. We show the result of choosing both of the children in MC2. If the left child is chosen, then the optimal solution, which is marked as 'O', can be reached. However, if the right child is

chosen, the optimal solution cannot be reached: the best component STP achievable is marked as 'S'. In this case, the value of 'S' (9) is only 1 less than the value of 'O' (10), so making the wrong choice has only a minor effect, assuming the subsequent choices are good.

3.2.1 Heuristics

In this section we describe a few heuristics for choosing which constraint to replace next, based on the state of the current component STP.

First, we offer a particular viewpoint concerning the replacement of constraints that gives insight into what makes a good heuristic. At any given iteration of the algorithm, the component STP `cSTP` represents the set of schedules that satisfy the STPP constraints to some degree, where the "degree" is quantified as the utilitarian value of a schedule. The utilitarian value of `cSTP` represents a lower bound for all schedules it contains; specifically, if `cSTP` has a value of v, then every schedule within `cSTP` has a value greater than or equal to v. The eventual goal is to find the schedule within `cSTP` that has the maximum value.

The process of replacing a constraint with its child tightens `cSTP`, removing 0 or more of the schedules from `cSTP`. The schedules removed could be the lowest valued schedules, one of the optimal schedules, or all schedules. A good heuristic is one that chooses children that do not remove optimal schedules. Obviously, a sequence of heuristic choices in which no replacement removes an optimal schedule is guaranteed to find the optimal solution. We say that a heuristic choice that does not remove all optimal schedules is a correct choice, while a choice that does remove them all is an incorrect one. The degree to which a choice is incorrect can be quantified by the difference between the value of the optimal schedule before the choice was made

and the value after. When designing heuristics, we focused on making choices that would, on average, make correct choices or ones that were incorrect to a small degree.

There is at least one case in which we can use information in `cSTP`'s minimal network to guarantee a choice will be correct. The minimal network encodes a tightened form of each constraint, called *inferred constraints*, and a window of possible assignments for each event, called *assignment windows*. If we compare a child constraint to its corresponding inferred constraint in the minimal network and find that the inferred constraint *subsumes* the child, we know that propagating a child through the network will not change the minimal network. In other words, no schedules will be removed so the optimal schedules will remain. This is a check that we can use to improve any heuristic.

When the special case does not apply, however, we can still use the minimal network's inferred constraints in a heuristic we call *constraint width difference* (CWD). The CWD heuristic chooses the constraint with the minimum difference between the inferred constraint width and the child constraint's width. The idea is that a child that differs only slightly from its inferred constraint will remove fewer schedules than one that greatly differs from its inferred constraint. When multiple children have the minimal difference, ties are broken randomly.

The next heuristic calculates the effect of propagating a child more directly than does CWD by imitating the first step of the `AC-3cc` algorithm. `AC-3cc` detects inconsistency in a network when the assignment window for an event shrinks to width 0. This fact is the basis for a heuristic called *one step assignment window reduction* (1AWR). 1AWR calculates how much the assignment windows of the constraint's two events will shrink after the first iteration of `AC-3cc`, given each child constraint. This calculation requires only four arithmetic operations per child, so finding the

child with the minimal 1AWR value takes very little time.

A third heuristic uses information from the dependency graphs maintained in `AC-3cc` (see Section 2.3.1). Recall that the upper and lower dependency graphs reveal which constraints determine the upper and lower bounds in the assignment windows. The constraints in the graphs are, in some sense, the "important constraints". Therefore, it is conceivable that incrementing constraints that do not participate in either graph will have less effect than those that do: a constraint on both the upper and lower dependency graphs is guaranteed to shrink the network, whereas a constraint on neither graph can possibly have no effect. We defined a heuristic called *dependency graph aversion* (DGA) which gave a value of 0 to constraints on neither graph, a value of 1 to constraints on a single graph, and 2 to constraints on both graphs. The heuristic chooses the constraint with the lowest value, breaking ties randomly.

The next heuristic we describe, called *upper bound reduction* (UBR), computes an upper bound for the utilitarian value and chooses the constraint that lowers the upper bound the least. The upper bound for a single constraint is computed by finding the highest-level descendant that overlaps with the inferred constraint. The upper bound for each constraint is then summed to produce a value for the entire network. To choose which constraint reduces the upper bound the least, each child is propagated using `AC-3cc`. In terms of computational complexity, UBR is by far the most expensive, almost doubling the runtime of the greedy algorithm using it. Despite the extremely low complexity of `STPP_Greedy`, this is still an issue for our `GAPS` algorithm, which may call `STPP_Greedy` hundreds or thousands of times.

Finally, the most basic heuristic we use is to simply choose the next constraint randomly using a uniform distribution over all possible children. This heuristic, orig-

inally tried for a baseline comparison, works surprisingly well on randomly generated STPPs.

3.2.2 Multiple iterations

As we show in the following section, no one heuristic completely dominates the others. Which heuristic works best depends not only on the type and shape of the preference functions, but also on luck, since several heuristics randomly break ties. Therefore, a reasonable way to produce better results is to run all heuristics and use the best solution found. Alternatively, each decision could be based on a weighted vote from all heuristics, effectively combining all heuristics. A final strategy would be to pick the best heuristic and run it multiple times.

The third strategy is appealing since we can prove that a greedy search using the random heuristic is guaranteed to find the optimal solution if run an infinite number of times. In practice, running it only a few times produces very good results. Heuristics other than random do not work as well using multiple iterations since they tend to make the same decisions. However, there is a simple way to jump-start subsequent iterations for all heuristics to ensure that different decisions are made.

We demonstrate this with an example, shown in Figure 3.4, which shows three STPP constraints and a solution, 'GS', found after the first iteration of greedy search. If any solution with greater value than 'GS' exists, then that solution will involve at least one interval on or above the intervals marked 'SP'. If the next iteration is seeded with an initial component STP containing any of the intervals marked 'SP', then the greedy search will search part of the space not encountered by the first iteration. Exactly which 'SP' is chosen for the starting point can be determined by a heuristic or by running at most m additional iterations, once for each possible starting

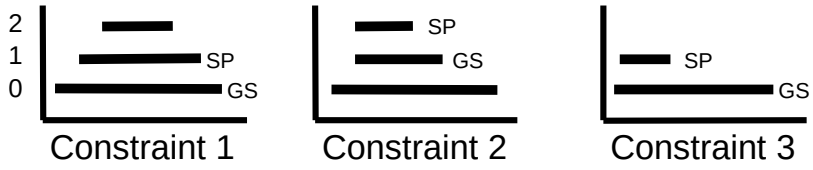


Figure 3.4: Three STPP constraints with horizontal lines representing the intervals at each preference level. 'GS' denotes the greedy solution found after the first iteration. If any solution with greater value exists, then the solution will involve at least one interval on or above the intervals marked 'SP'.

point. Given the superb solution quality obtained by running multiple iterations in this manner, this strategy would be acceptable for many practical applications. However, as we show in Section 3.3, a slightly more complicated algorithm (based on the same general idea) will give equivalent anytime performance and add a guarantee of completeness.

3.3 GAPS

We now show how to use `STPP_Greedy` to achieve an anytime, complete, and memory-boundable algorithm for finding the utilitarian-optimal solutions to unrestricted STPPs. Our algorithm, called the Greedy Anytime Partition algorithm for STPPs (`GAPS`), can be understood as combining pruning, greedy search, and a "divide and conquer" strategy. `GAPS` starts with a solution found by `STPP_Greedy` and uses it to partition the entire STPP search space into n subspaces, $n-1$ of which will be recursively solved by `GAPS`, and one of which will be discarded. While n varies for each case, it is approximately equal to the number of constraints in the STPP.

3.3.1 Partitioning subspaces

The heart of the algorithm is the `partition` method, which accepts a subspace SS and then partitions it with respect to a consistent component STP G within SS.

The method divides SS into n smaller subspaces and returns $n-1$ of them. The n^{th} subspace not returned is discarded (pruned) because it will only contain component STPs with utilitarian value less than or equal to G's value. If a better solution than G exists, it is guaranteed to reside in one of the other $n-1$ subspaces.

We illustrate the partition method on the subspace in Figure 3.5(a), which represents the entire search space for our example. We represent a subspace as a set of (partial) preference projections: the three trees in (a) correspond to the preference projections in Figure 3.1(d-f). We partition with respect to a component STP found by running `STPP_Greedy` on (a). In our example, the result is a component STP $G = \{C_{\langle 1,2,1\rangle}, C_{\langle 2,2,2\rangle}, C_{\langle 3,3,1\rangle}\}$ of value 7 (2+2+3) represented by the boxed projected constraints within (a). (Recall that a component STP is formed by selecting a single constraint from each projection).

The partition method performs a series of smaller partitions—one for each preference projection. The central idea for each smaller partition is to select a projected constraint $C_{\langle k,l,p\rangle}$ from the greedy solution, and then split the search space into two or more parts: one part will contain component STPs in which the k^{th} constraint has value less than or equal to l, while the others will represent component STPs in which it has value greater than l.

Thus, in the current example, (a) is initially partitioned with respect to constraint $C_{\langle 1,2,1\rangle}$, which is constraint MC1's representative in the greedy solution. To achieve the split, into subspaces (b) and (b'), we first copy the preference projections for constraints other than MC1 into both (b) and (b'). Then subspace (b') also inherits the hard constraints projected from MC1 with values less than or equal to 2 (the value of the constraint projected from MC1 in the greedy solution), while (b) inherits the ones with values greater than 2.

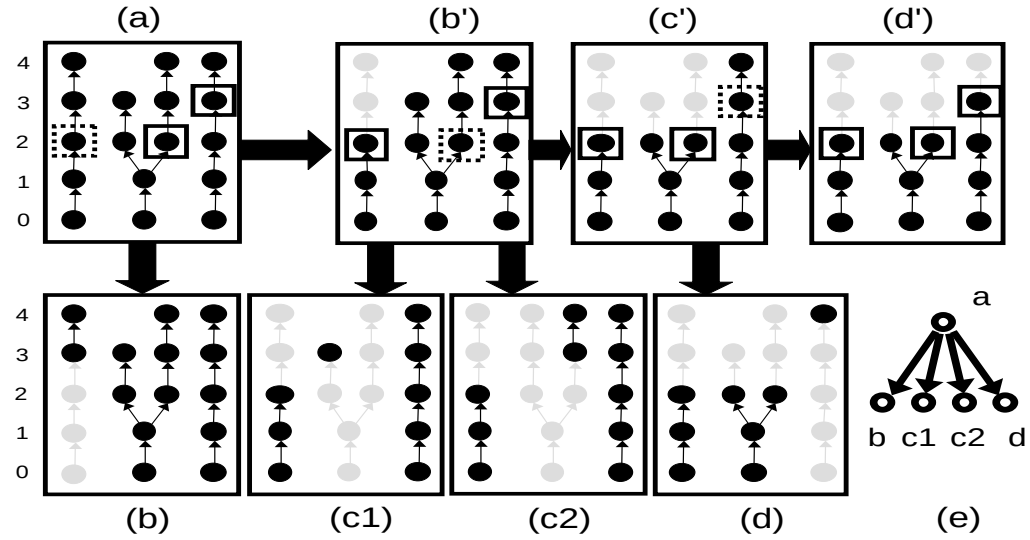


Figure 3.5: Subspace partitioning process used in GAPS.

Next, subspace (b') is split with respect to constraint MC2 into (c1), (c2), and (c'). Again, we first copy the preference projections for constraints other than MC2 into each new subspace. Then subspace (c') gets the subtree consisting of constraints at and below the greedy solution level, while the rest is split into two parts, (c1) and (c2), because the constraints from MC2 above the greedy solution split into two trees. Similarly, subspace (c') is split into (d) and (d') based on constraint MC3. In the end, the original problem (subspace (a)) is partitioned into the subspaces (b), (c1), (c2), (d), and (d').

Significantly, the search space represented by part (d') consists only of component STPs with value 7 or less, since at each stage of its formation, we kept only the projected constraints at or below the level of the initial greedy solution. Therefore, since no component STP in the space can exceed the value of the first greedy solution, the (d') partition can be pruned from the search. Notice that the size of subspace (d') (i.e., the number of component STPs in (d')) depends on the value of the greedy solution. Therefore, better greedy heuristics lead to more effective pruning.

We formalize this process with three definitions: first we define an *STPP subspace* in terms of preference projections; second, we define the *single constraint partition* of a subspace based on a constraint in a preference projection; and finally we define a *component STP partition*, which is recursively defined in terms of single constraint partitions.

Definition 7 (STPP subspace)**.** *An* **STPP subspace** *SS of an STPP represented as a set of m preference projections $S = \{\mathcal{P}_k : 0 \leq k < m\}$, is a set of preference projections $SS = \{\mathcal{P}'_k : 0 \leq k < m, \mathcal{P}'_k \subseteq \mathcal{P}_k\}$.*

Each box in Figure 3.5 represents a single STPP subspace. Notice that the constraints in each partial preference projection form a tree, which is important for our greedy search because it relies on the current component STP getting tighter after each replacement.

A single constraint partition splits a subspace into two or more parts, using a single projected constraint, $C_{\langle k,l,p\rangle}$.

Definition 8 (single constraint partition)**.** *A* **single constraint partition** *of a subspace $S = \{\mathcal{P}_k : 0 \leq k < m\}$ with respect to a single constraint $C_{\langle k,l,p\rangle}$ is a set of $n+1$ subspaces, $\Upsilon(S|C_{\langle k,l,p\rangle}) = \{S_0, S_1, \ldots, S_n\}$, where n is the size of $H = \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r = l + 1\}$, H_i is the i^{th} element of H and*

- $S_n = S \backslash \mathcal{P}_k \cup \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r \leq l\}$
- $\forall_{0 \leq i < n} S_i = S \backslash \mathcal{P}_k \cup H_i \cup des(H_i)$.

Theorem III.1. *A single constraint partition retains all component STPs in the original subspace.*

Proof. $\Upsilon(S|C_{\langle k,l,p\rangle})$ retains all component STPs in S if $S = \bigcup_i S_i \in \Upsilon(S|C_{\langle k,l,p\rangle})$. By definition 8, $\bigcup_i S_i \in \Upsilon(S|C_{\langle k,l,p\rangle}) = S\backslash\mathcal{P}_k \cup \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r \leq l\} \cup \bigcup_i H_i \cup des(H_i)$. Since $S = S\backslash\mathcal{P}_k \cup \mathcal{P}_k$, it suffices to show that $\{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r \leq l\} \cup \bigcup_i H_i \cup des(H_i) = \mathcal{P}_k$. Since $\mathcal{P}_k = \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r \leq l\} \cup \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r > l\}$, it suffices to show that $\bigcup_i H_i \cup des(H_i) = \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r > l\}$, which is equivalent to $H \cup \bigcup_i des(H_i)$ Since a child is always tighter than its parent (and its parent's parent), any member of the projection has an ancestor at each level below it. Therefore, since H contains all members at level $l+1$, and $\bigcup_i des(H_i)$ contains all members that descend from elements in H, all members of $\{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r > l\}$ must be in H or $\bigcup_i des(H_i)$. □

Each stage in Figure 3.5 represents a single constraint partition: (a) into (b) and (b'); (b') into (c1), (c2), and (c'); and (c') into (d) and (d'). The series of single constraint partitions compose a component STP partition:

Definition 9 (component STP partition)**.** *A* **component STP partition** *of a subspace* $S = \{\mathcal{P}_k : 0 \leq k < m\}$ *with respect to a component STP* $G_{[0,m]} = \{C_{\langle k,l,p\rangle} : 0 \leq k < m\}$ *is as set of subspaces* $\Psi(S|G_{[0,m]})$*, recursively defined as* $\Psi(S|G_{[i,m]}) = \Upsilon(S|G_i)\backslash S_n \cup \Psi(S_n|G_{[i+1,m]})$*, where the base case is* $\Psi(S_n|G_{[m,m]}) = S_n$*.*

Theorem III.2. *A component STP partition retains all component STPs in the original subspace.*

Proof. $\Psi(S|G_{[0,m]})$ retains all component STPs in S if for any $G_{[0,m]}$, $S = \bigcup_i S_i \in \Psi(S|G_{[0,m]})$. Since $\Upsilon(S|G_i)$ is a partition (Theorem III.1), it suffices to show that $\Psi(S_n|G_{[i+1,m]})$ retains all component STPs in S_n. The base case, $\Psi(S_n|G_{[m,m]}) = S_n$, trivially retains all component STPs in S_n. Therefore, $\Psi(S|G_{[0,m]})$ retains all component STPs in S. □

Thus, Figure 3.5 represents one component STP partition of subspace (a) by component STP $G_{[0,3]} = \{C_{\langle 1,2,1\rangle}, C_{\langle 2,2,2\rangle}, C_{\langle 3,3,1\rangle}\}$ that resulted from three single constraint partitions. $\Psi(S|G_{[2,3]})$ in this example corresponds to the partition of (c'): $\Psi(\text{c}|G_{[2,3]}) = \text{d} \cup \Psi(\text{d'}|G_{[3,3]})$. The base case is $\Psi(\text{d'}|G_{[3,3]})$, which adds (d') to the set of subspaces. That we prune (d') from our search is distinct from the definition.

We now prove that the final subspace that results from a component STP partition has no component STP with value greater than the component STP with which it was partitioned.

Theorem III.3. *For a component STP partition $\Psi(S|G)$ that returns subspaces $\{S_0, S_1, \ldots, S_n\}$, the optimal solution to subspace S_n is a component STP $H = \{C_{\langle k,l,p\rangle} : 0 \leq k < m\}$ such that $u(H) \leq u(G)$.*

Proof. Given the set of n subspaces $\Psi(S|G)$, the subspace S_n is the result of m single constraint partitions, each of which operates on the last subspace produced from the previous single constraint partition. For single constraint partition 0, the last subspace is $S_n^0 = S \backslash \mathcal{P}_0 \cup \{C_{\langle 0,r,p\rangle} \in \mathcal{P}_0 : r \leq l_0\}$, where l_0 is the preference level for constraint G_0. For single constraint partition i, the last subspaces is $S_n^i = S_n^{i-1} \backslash \mathcal{P}_i \cup \{C_{\langle i,r,p\rangle} \in \mathcal{P}_i : r \leq l_i\}$, where l_i is the level for constraint G_i. Filling out the recursion, $S_n = S_n^m = \bigcup_i \{C_{\langle i,r,p\rangle} \in \mathcal{P}_i : r \leq l_i\}$. The largest valued component STP H in S_n is one in which each component H_i has value l_i. Therefore, $u(H) = \sum_i l_i = u(G)$. □

3.3.2 Algorithm

The `partition` method just described is a key operation in `GAPS`, for which pseudo-code is given in Figure 3.6.[1] The first step of the algorithm calculates the

[1]For clarity, the algorithm presented hides many bookkeeping tricks necessary for space efficiency. Most importantly, not all the information required to represent subspaces is stored directly;

preference projections (line 1), which compose the first subspace to be processed (subspace (a) in our example). The iterative part of the algorithm begins by calculating the upper bound for the subspace denoted by *currentSS* (line 5). The upper bound is the sum of the highest possible level for each projection. In subspace (d), for example, the upper bound is 8 (2+2+4).[2] Next, if the upper bound is greater than the best solution found thus far, `STPP_Greedy` is called to find a consistent component STP within *currentSS* (line 6). If the value of the greedy solution exceeds the best found so far, the solution is stored (lines 8-10).

Next, *currentSS* is partitioned based on the greedy solution *gSol* and adds the returned subspaces to a priority queue (line 12). Finally, a new subspace is retrieved from the priority queue (line 14) and the next iteration begins. The iterations continue until the priority queue empties, a signal that the optimal solution has been found. Not shown are additional stopping conditions that allow the algorithm to quit after a time threshold or after a specified number of iterations.

3.3.3 Memory-boundable property

By default, priority is given to subspaces with higher upper bounds so that subspaces with the greatest potential are explored first. Unfortunately, this strategy allows the set of queued subspaces to grow to an unmanageable size. An alternative is to recursively explore each child of a subspace before moving on to the next (similar to a depth-first search). Since the maximum number of recursive partitions for a search space is $m \cdot (|A|-1)$ and each partition can produce at most $m + (n-1)$ children (where m is the number of constraints and n is the maximum p for any $C_{\langle k,l,p \rangle}$), the maximum number of subspaces queued is $m^2 \cdot (|A|-1)$ (n disappears

organizing the subspaces in a tree allows for efficient storage. Appendix A gives the full algorithm.

[2] A tighter upper bound is calculated within `STPP_Greedy`. The tighter bound is passed on to a subspace's children.

```
GAPS(STPP)
1.  currentSS ← project(STPP)
2.  ssQ ← priority queue of subspaces, initially empty
3.  bestValue ← -1; best ← NO_SOLUTION
4.  REPEAT
5.      IF bestValue < calcUB(currentSS)
6.          gSol ← STPP_Greedy(currentSS)
7.          value ← sumPreference(gSol)
8.          IF value > bestValue
9.              best ← gSol
10.             bestValue ← value
11.         END
12.         addToQueue(ssQ, partition(currentSS, gSol))
13.     END
14.     currentSS ← getNextSS(ssQ);
15. WHILE (ssQ not empty)
RETURN best
```

Figure 3.6: A high-level description of the STPP algorithm `GAPS`. A detailed version can be found in Appendix A.

because $n \ll m$).

Of course, the two strategies can be combined: given a bound of M nodes, the first $M-(m^2 \cdot (|A|-1))$ nodes can be inserted using the upper bound priority function. Any time the number of nodes in memory exceeds this threshold, new subspaces are inserted using the depth-first priority function. Any time the number of nodes drops below this threshold, upper bound priority is resumed. Therefore, the caller of the algorithm can effectively manage a space/time trade-off.

3.3.4 Completeness

The `GAPS` algorithm is complete with respect to preference projection of the original problem. In other words, a complete algorithm for this problem finds a component STP whose value is provably greater than or equal to the value of all component STPs in the projection. Therefore, proving `GAPS` is complete requires two smaller

proofs: that the partition in each iteration is a true partition; and the component STPs not searched (i.e., those in the discarded partition) have value less than or equal to the value of the best.

The first part has already been proved in Theorem III.1 and Theorem III.2. The second part is proved by Theorem III.3 because the highest valued component STP in the pruned partition has value equal to component STP with which the subspace was partitions.

Theorem III.4. *GAPS is complete with respect to preference projection of the original problem.*

Proof. Each iteration of GAPS converts the (sub)problem into $n+1$ subproblems and finds a solution within it of value g. The $n+1$ subproblems retain all component STPs in the original (sub)problem (Theorem III.2). The first n are recursively solved by GAPS. The last subproblem is guaranteed not to have a component STP with value greater than g (Theorem III.3). Therefore, either the optimal solution has value g or the optimal solution resides in one of the first n subproblems. □

3.4 Empirical Evaluation of STPP_Greedy

In this section, we evaluate STPP_Greedy and GAPS, showing which heuristics work well for STPP_Greedy and how GAPS compares to a branch-and-bound algorithm and a SAT-based algorithm that also find utilitarian solutions to unrestricted STPPs. Before describing the tests and results, we describe how we generated the random STPPs used in the tests.

3.4.1 Generating random STPPs

To aid in understanding the test results we present below, we describe our method of generating random STPPs, which is similar to the one used in [52]. While understanding the exact process is not necessary for assessing our results, it is important to understand the parameters that define the size of the STPP and the shape of its preference functions.

We begin the process by generating a random assignment for all events, and stochastically building an STP around it. Then, we expand each constraint in the STP to an STPP constraint, using the STP constraint as the lowest level preference. In this process, we use the following parameters:

numEvents	Number of events in the network
timeHorizonPerEv	Maximum value of any event is numEvents × timeHorizonPerEv
numConstraints	Number of dual bounded constraints (max: $\frac{1}{2}(\text{numEvents})^2$)
minDomainWidth	Minimum width of any STP interval
maxDomainWidth	Maximum width of any STP interval
numLevels	Maximum number of preference levels in a constraint
reductionFactorLB	Minimum fraction of level i STP interval width that will exist at level $i+1$
reductionFactorUB	Maximum fraction of level i STP interval width that will exist at level $i+1$
splitProb	Probability of an STP interval at level i splitting into two intervals at level $i+1$

The first two parameters listed define how the initial solution is generated. The assignment for the first event is 0, while all other events are randomly chosen from the interval [0, `numEvents` · `timeHorizonPerEv`]. We refer to this assignment as the *seed solution*.

The next three parameters define how the constraints for the initial STP are chosen. To create a constraint, a pair of unconstrained events is randomly chosen along with a constraint width from the interval [`minDomainWidth`, `maxDomainWidth`]. The start bound of the constraint is determined by the formula *solution difference* - *offset*, where the solution difference is the difference of the seed solution values for each event, and the offset is some random value less than the constraint width. The end bound is the start bound plus the constraint width.

For example, imagine the pair of randomly chosen events is t_5 and t_{12}, which are assigned the values 22 and 30 in the seed solution—a solution difference of 8. Imagine the constraint width is randomly chosen as 15, and the random offset chosen as 11; then the start bound is 8-11=-3, and the end bound is -3+15=12. Notice that with this strategy, a solution to the STP will always exist, since each constraint is guaranteed to allow the seed solution.

The last four parameters define how the preference function is created from each STP constraint. First, the lowest preference level is set to be the interval defined by the STP constraint. The number of intervals for the second level is determined by a random number between 0 and 1: if less than `splitProb` the number of intervals is 2; otherwise a single interval is created. (When `splitProb` is 0, all preference projection trees are chains.) We choose the total width of the interval(s) by multiplying the parent interval's width by a reduction factor, chosen from [`redLB`, `redUB`]. Then the interval(s) are split (if necessary) and placed randomly within the parent interval. This step iterates until the `maxNumLevels` level is hit or until the interval width reduces to 0.

Continuing our example using the STP interval [-3, 12], let `reductionFactorLB` = .5 and the `reductionFactorUB` = 7. If the randomly chosen reduction factor is 0.6, then the width of the new interval will be $0.6 \times 15 = 9$. The new interval's start bound could be placed anywhere in the interval [-3, 3]. We can choose a value in this interval by choosing from the interval [0, 15-9] and adding it to -3.

This process continues for each level until the maximum number of levels has been reached or the constraint width at some level rounds down to 0. The result is a semi-convex step function.

The last parameter, `splitProb`, defines how often a level "splits" when creating

a new level. If `splitProb` is 0, the intervals never split, ensuring the functions are semi-convex. If set to any value greater than 0 and less than or equal to 1, the parameter defines the probability that at any level, the interval at the lower level is split into two parts.

3.4.2 Comparing STPP_Greedy heuristics

To test the heuristics described in Section 3.2.1, we first explored STPPs with semi-convex functions with three different shapes defined by the reduction factor parameters: low-slope, steep slope, and highly-varied. The low slope preference functions used a reduction factor range from 0.01 to 0.2; the steep slope preference functions used a range from .8 to .99; and the highly varied slope preference functions used a range from .01 to .99.

Within these three shapes, we varied the number of preference levels and the constraint density. In this test, we define constraint density as the number of constraints in the problem divided by the maximum number of constraints possible, which is $\frac{1}{2}(\texttt{numEvents})^2$. A "low" constraint density is any fraction less than .3, while a "high" density is any greater fraction. The size of the networks in this test are small due to the difficulty of finding utilitarian optimal solutions to STPPs. Remember that the `numLevels` parameter defines the *maximum* number of levels, which is not always reached by each constraint, especially when the reduction factor is low.

Table 3.1 shows the quality of solutions found by 5 different heuristics on different types of problems. The quality is presented as a percentage of the utilitarian optimal score averaged over 100 runs. For each run, the maximum of all heuristics is recorded as the value of the "Max" heuristic. This corresponds to the first multiple iteration strategy mentioned in Section 3.2.2.

Low slope

Density	levels	Random	CWD	1AWR	DGA	UBR	Max
low(12,16)	2	.938	.924	[.959]	.953	.957	.989
	5	.922	.938	[.967]	.949	[.967]	.993
high(10,40)	2	.786	.831	.894	.823	[.904]	.953
	5	.822	.832	.882	.817	[.889]	.951

Steep slope

Density	levels	Random	CWD	1AWR	DGA	UBR	Max
low(15,20)	3	.999	.998	.999	.999	[1.0]	1.0
	5	.996	.992	[.997]	.995	[.997]	.999
	7	.976	.973	[.991]	.977	[.991]	.995
high(8,23)	3	.991	.994	.994	.993	[.995]	.999
	5	.964	.966	.980	.965	[.981]	.992
	7	.918	.918	.957	.941	[.965]	.975

Highly varied slope

Density	levels	Random	CWD	1AWR	DGA	UBR	Max
low(13,18)	3	.932	.924	.978	.950	[.980]	.988
	5	.867	.861	[.944]	.911	.943	.969
	7	.835	.823	.949	.883	[.952]	.967
high(8,25)	3	.860	.883	.935	.899	[.943]	.969
	5	.794	.813	.890	.831	[.895]	.935
	7	.794	.806	.891	.834	[.897]	.936

Table 3.1: Greedy search scores for different heuristics, reported as percentage of optimal score. Each line represents 100 trials. The values next to "low" and "high" indicate the number of events and constraints, respectively. The best heuristic for each test is bracketed.

The most striking result evident in Table 3.1 is how well the greedy solution works, regardless of the heuristic used. For problems that contain preference functions with steep slopes, the worst heuristics found solutions that averaged over 90% of the optimal value. For the low-slope and highly-varied slope cases, the worst heuristics averaged just under 80%. This data suggest that for randomly genereated STPPs, multiple iterations of greedy search using any heuristic is a fast, high-quality alternative to solving STPPs exactly.

As for individual heuristics, two dominated the others: 1 step assignment window reduction (1AWR) and upper bound reduction (UBR). In some cases, the two other

heuristics we designed, CWD and DGA performed worse than random, suggesting (wrongly) that these heuristics are of no value. A quick look at the data might lead to the false conclusion that since the UBR heuristic dominates in most cases, the other should be discarded in favor of UBR. However, Table 3.1 only reports the *average* percentage of optimal value and does not reflect that even heuristics with horrible average case performance may find the best solution for a given problem instance. The "Max" column in the table always exceeds the best heuristic, suggesting a strategy using multiple good heuristics may outperform a single great heuristic.

As a point of reference, the value of the optimal solution for the last test in the table (highly-varied, high density, 7 levels) was an average of 65% of the initial upper bound for each problem (standard deviation of about 7%). We state this as evidence that the problems in this category did not have solutions that were in the "easy" regions of the space, namely where optimal solutions are close to 0 or close to the original upper bound. This notion explains the extremely good performance of greedy search in the low-density, steep-sloped problems. In a function with steep slopes, the projected intervals at the top of the preference function are a sizeable portion of the lowest projected level; meaning if a solution satisfies the lowest level constraint, it is highly probable that it satisfies the highest level constraint as well. Notice that as the number of levels increases in this category, the performance drops.

The main drawback of this test is that the heuristics were tested on relatively small problems. To test the relative performance of the heuristics on much larger problems, we ran additional tests where we could not compute the optimal solution. We found that for the larger problems, UBR and 1AWR still dominated, with UBR barely emerging as the best heuristic in terms of solution quality. In these tests, we found that the running time for the heuristics varied substantially. Table 3.2 shows

Heuristic	25	50	75	100
RANDOM	1.25	0.79	1.89	3.12
CWD	1.71	2.80	6.26	7.99
1AWR	0.93	3.14	7.52	10.58
DGA	0.78	2.18	3.58	6.40
UBR	23.90	171.41	573.12	1280.34

Table 3.2: Average running times (ms) for greedy heuristics on STPPs with 25, 50, 75, and 100 constraints.

the average running times for a test involving problems with 25, 50, 75, and 100 constraints.

1AWR is much faster than UBR so the small quality advantage of UBR may not be worth the additional computation time for situations that require many iterations of the greedy algorithm.

3.4.3 Greedy search heuristics for STPs with unrestricted functions

When the semi-convexity restriction is removed, the preference functions project into full trees rather than simple chains. Each "split" in the tree (which corresponds to a hard constraint with more than one child) causes an exponential increase in the size of the search space (relative to the case where it does not split). However, the split causes only a constant increase in the run time of `STPP_Greedy`. It is reasonable to expect, therefore, that the solution quality of `STPP_Greedy`'s solutions would reduce significantly as the number of "splits" increases. Our tests on random networks revealed only a slight reduction.

We found that if the number of interval splits is small, the heuristics 1AWR and UBR still do very well. Table 3.3 shows the results for the highly-varied case when the semi-convexity restriction is removed.

The most important result in Table 3.3 lies in the difference between the semi-convex case (Split Prob = 0) and the case where intervals split the most often (Split

Highly varied slope

Density	levels	Split Prob	Random	1AWR	UBR	Max
low(13,18)	3	0	.951	.979	[.980]	.988
	3	.2	.911	.961	[.962]	.973
	3	.4	.882	[.956]	[.956]	.962
	3	.8	.819	.931	[.938]	.945
	5	0	.869	.944	[.947]	.957
	5	.2	.838	[.935]	[.935]	.948
	5	.4	.825	.928	[.933]	.944
	5	.8	.778	.922	[.929]	.937
	7	0	.851	[.951]	[.951]	.960
	7	.2	.831	.931	[.935]	.942
	7	.4	.812	.921	[.926]	.939
	7	.8	.796	.919	[.921]	.932
low(8,25)	3	0	.878	.938	[.941]	.957
	3	.2	.838	.934	[.937]	.951
	3	.4	.845	.917	[.918]	.940
	3	.8	.797	[.893]	.890	.916
	5	0	.816	.886	[.891]	.911
	5	.2	.796	.883	[.890]	.918
	5	.4	.776	.875	[.880]	.906
	5	.8	.773	.873	[.878]	.900
	7	0	.811	.885	[.898]	.915
	7	.2	.800	.884	[.893]	.916
	7	.4	.772	.868	[.872]	.897
	7	.8	.759	.867	[.868]	.890

Table 3.3: Greedy search scores for different heuristics on STPPs with unrestricted preference functions. Each line represents 100 trials. The values next to "low" and "high" indicate the number of events and constraints, respectively. The best heuristic for each test is bracketed.

Prob = 0.8) for each density/level combination. The difference is not dramatic, bodding well for practical applications in which the semi-convex restriction is a problem.

Allowing unrestricted preference functions does not substantially affect run time, as we will show later in Figure 3.9 and Figure 3.11.

3.5 Empirical Evaluation of GAPS

To test `GAPS`, we randomly generated a set of STPPs and compared the solution quality and running time for `GAPS` to a branch-and-bound (`BB`) algorithm that we

implemented and optimized. We are mainly interested in the anytime properties of GAPS, since for the Autominder application (and we expect many other real-world domains) getting the optimal solution is less important than finding high-quality solutions quickly. However, a danger with algorithms designed to have good anytime properties (e.g., most local search algorithms) is that they take excessive amounts of time to find the optimal solution, or at least to prove that the solution found is optimal. Thus, our comparison to BB is meant to show that GAPS compares favorably to an algorithm designed simply to find the optimal solution as quickly as possible.

3.5.1 Branch-and-bound algorithm

The branch-and-bound algorithm we describe is a variant known as the Russion Doll algorithm, which is a way to combine the branch-and-bound concept with dynamic programming [21]. The BB algorithm accepts as input an ordered set of m preference projections, each of which represents an STPP constraint. The algorithm systematically searches the space of component STPs by making incremental changes to a single component STP, called cSTP, and by using a lower bound and a vector of upper bounds to prune the search space. In addition to cSTP, the algorithm maintains the following elements: best, which stores the best component STP found thus far; ppi, which is a index into the ordered set of preference projections; and a flag for each constraint in the projections indicating whether the constraint has been "finished". We refer to constraint k in cSTP as cSTP[k] and to projection k as $\mathcal{P}_k$.

The algorithm initializes cSTP to $ROOT$ and ppi to m. If $ROOT$ is inconsistent, the algorithm immediately exists. Otherwise, the algorithm repeats the following two steps:

1. Move to the next search point:

(a) If $ppi = m$, a solution has been found; assign `cSTP` to `best`, decrement ppi, and repeat step 1.

(b) If $ppi = -1$, the search is complete; return `best`.

(c) If an unfinished constraint in $\mathcal{P}_{ppi}$ exists, assign the lowest valued one to `cSTP[ppi]`.

(d) Otherwise, assign the root of $\mathcal{P}_{ppi}$ to `cSTP[ppi]`, reset all flags in $\mathcal{P}_{ppi}$ to unfinished, decrement `ppi`, and repeat step 1.

2. Check consistency of `cSTP`:

(a) If `cSTP` is consistent, mark `cSTP[ppi]` as finished and increment `ppi`.

(b) If `cSTP` is inconsistent, mark `cSTP[ppi]` and all its descendants as finished.

To make the search efficient, the algorithm also maintains a vector of m upper bounds (one per constraint) and a single lower bound. The lower bound is simply the value of `best`. Upper bound i, which corresponds to $\mathcal{P}_i$, represents the highest possible total for the constraints from projections $i + 1$ and greater. The upper and lower bounds can be used to calculate the minimum value for the constraint in projection i needed to lead to a better solution. For example, if the value of `best` is 20, the total value of all constraints in projections 0 through $i - 1$ is 8, and the upper bound for all later constraints is 9, then the constraint from projection i must have a value of 4 (20-8-9+1) to reach a score of 21. Therefore, all constraints in $\mathcal{P}_i$ with values less than 4 can be marked as finished.

Each upper bound is set the first time its corresponding projection is reached during search. Since the algorithm works backward from the last constraint, it is guaranteed that when the search first reaches projection i, the entire search space

consisting only of projections $i+1$ and greater have been searched. Therefore, the best score thus far is an upper bound on the total preference values for projections $i+1$ and greater.

3.5.2 Setup

We compared the `BB` algorithm and four variations of the `GAPS` algorithm: GAPS-1, which runs only a single greedy iteration; GAPS-m, which runs m iterations (m is the number of constraints); GAPS-m^2, which runs m^2 iterations; and `GAPS`, which runs until completion or until a 10-minute time threshold is reached. The same threshold was applied to `BB`, which returned the best solution found when the threshold is met.

We ran four tests: the first tested the effect of constraint density on performance, where density is the ratio of constraints to events. There were two parts to this test: one which fixed the number of events to 10 and varied the constraint density from 1 to 3.5, and another that held the number of constraints fixed at 20 and varied the number of events from 8 to 20. The second test compared performance as the number of events increased. The constraint density was fixed at 2 and the number of constraints, which was limited by the efficiency of the complete algorithms, was varied from 10 to 24. To measure how the algorithms fared with larger problems, the third test varied the number of constraints from 20 to 180 but omitted `BB` because of its extremely poor performance.

The first three tests operated on networks restricted to semi-convex preference functions. Recall that a semi-convex function f is one in which the set $\{x : f(x) > l\}$ forms at most a single interval for any level l. This restriction ensures all preference projection trees are chains, as in constraints MC1 and MC3 in our example. We

make this restriction because there is particular interest in this restriction in previous STPP work [39, 40].

To show that `GAPS` performs well without this restriction, our fourth test repeats the second, but sets the `splitProb` parameter to 0.2, which produces networks that contain, on average, about 1 branch in the projection tree for every 5 intervals.

The shapes of the preference functions were again determined by the interval [*reductionFactorLB*, *reductionFactorUB*] described in Section 3.4.1. This factor represents the fraction of a projected constraint's interval that is covered by its child constraints. A factor close to 1 results in functions with steep slopes; one close to 0 results in shallow slopes and fewer levels. All tests used the interval [.5, .9] for the reduction factor during generation. We arrived at this value because `STPP_Greedy` performed *worst* with this interval in the tests presented in Section 3.4.3.

The maximum number of preference levels was set to 10. The third test averaged the results from 100 trials, while the others averaged 200 trials.

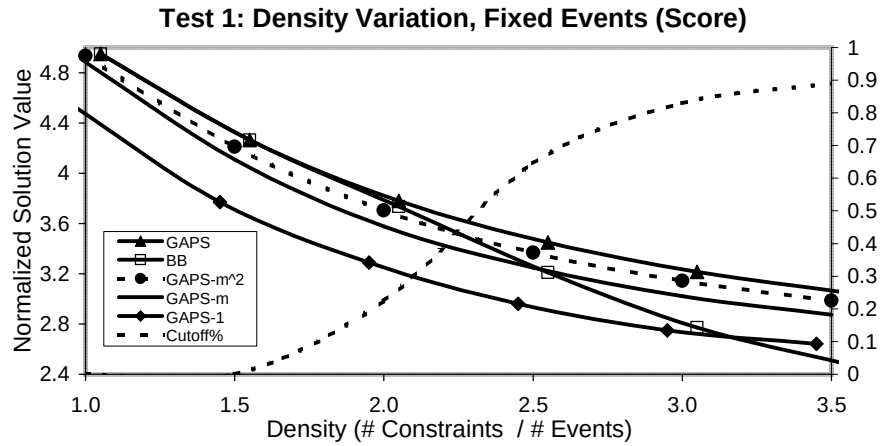
All tests and algorithms were written in Java and run on a Pentium 4 3Ghz WindowsXP machine. (Hyper-threading was enabled, so cpu usage was only 50%.) Timing was done using a library that counts only time spent in the main thread (i.e., it ingores garbage collection and system interruptions).

3.5.3 Results

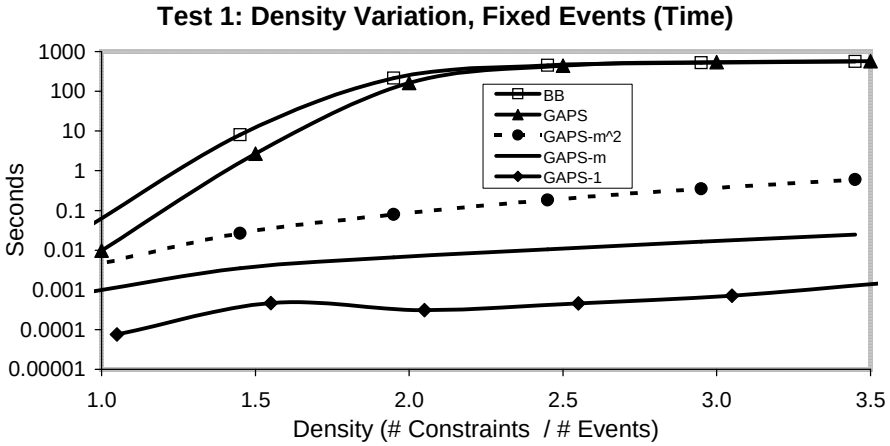
We now present the results for the first four tests.

Test 1: constraint density

Figures 3.7 and 3.8 show the results of the first test, which varied the constraint density. Figures 3.7(a) shows the average normalized utilitarian value for the solutions from each tested algorithm (the left axis) and the percentage of problems in



(a)



(b)

Figure 3.7: Solution quality and running times for the Test 1, which varied the constraint density while holding the number of events constant.

which at least one of the algorithms was cut off by the time restriction (the right axis). The normalized utilitarian value is computed by dividing the utilitarian value by the number of constraints. The value itself (shown on the left axis) has little significance; the relative values among algorithms interest us most. Plot (b) shows the running time in seconds for each of the algorithms.

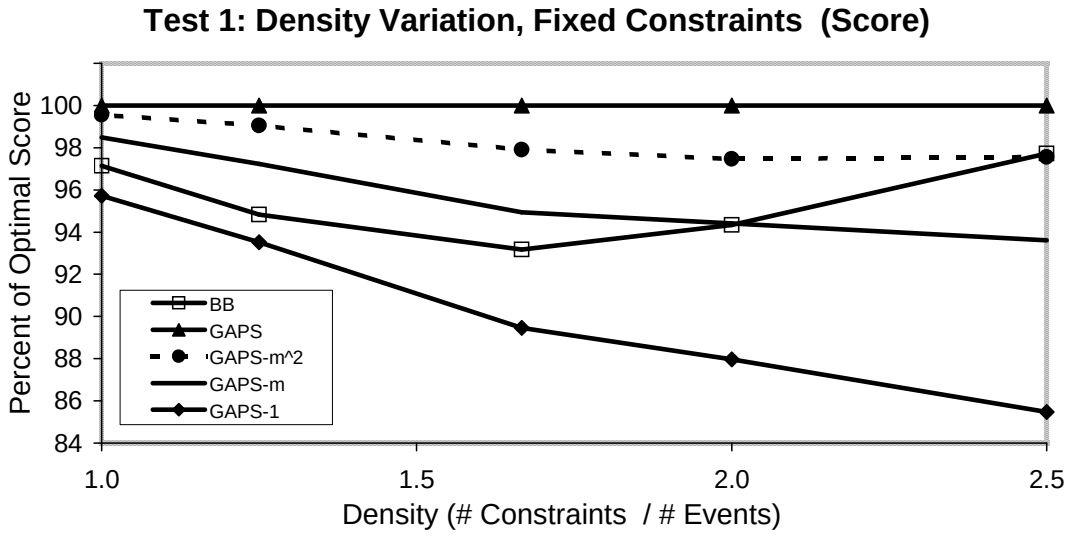
Figure 3.7(a) shows us that the relative performance of the `GAPS` variations changes little as density increases: GAPS-1 solution values ranged from 92-87%

of GAPS's solution values; GAPS-m ranged from 98-94%; and GAPS-m^2 ranged from 99.7-97.9%. As expected, the anytime abilities of BB are not good: as the cutoff percentage rises, the solution quality drops quickly relative to the GAPS variations. In fact, the final data point in the graph, where density=3.5, shows that even after 10 minutes BB produces lower scores than GAPS-1 (which takes only a few milliseconds). Unsurprisingly, the normalized solution quality trends downward, since increasing constraint density reduces the number of legal schedules and, consequently, the chance of having high-valued solutions.

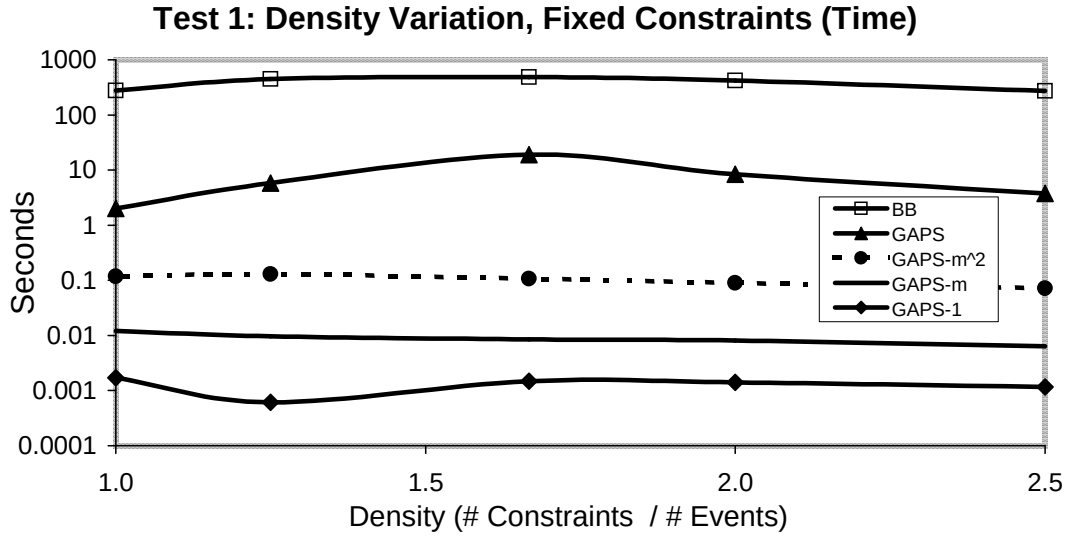
Figure 3.7(b) shows that for problems solvable in less than 10 minutes (densities 1.0 and 1.5) GAPS found the optimal solution much more quickly than BB (in 75% less time and 65% less time, respectively). Figure 3.7(b) also shows how quickly GAPS can find good solutions: for the first two datapoints, GAPS-m found solutions that were >96% of optimal in less than 5ms. For the last datapoint, it found solutions that averaged 94% of the GAPS score in an average of 24ms.

The problem with the test just described is that as we increase the density, we are increasing the number of constraints, which is the key parameter indicating problem size. As a result, the variation we see between events is dominated by the problem size, obscuring a meaningful comparison of constraint density. The second part of Test 1 addresses this issue by holding the number of constraints constant, allowing density effects to be the dominant parameter. Figure 3.8 shows the results from this test.

Notice that in Figure 3.8(a), the solution quality of all GAPS variations drops as density increases. This occurs because as the events become more connected, the effect of making a bad heuristic choice gets worse. BB improved as the problems became more connected. It tied with GAPS-m^2 on the highest density tested, but



(a)



(b)

Figure 3.8: Solution quality and running times for the Test 1, which varied the constraint density while holding the number of constraints constant.

required three orders of magnitude more time. Figure 3.8(b) shows that running time is only slightly affected by density.

Test 2: medium-sized semi-convex problems

Figure 3.9 shows the results for the second test, which varied the number of constraints from 10 to 24. The lessons from the second test are the same as those from the first. One difference is that because density is held constant, the normalized solution value remains relatively fixed.

Test 3: large semi-convex problems

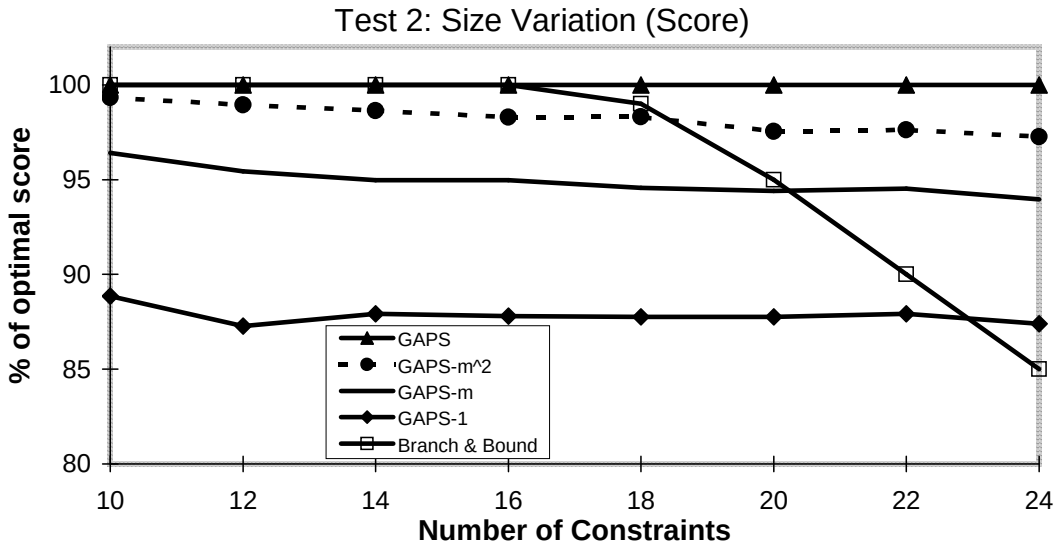
Figure 3.10 shows the results for the third test, which repeated the second test for much larger problems. Notice that as the problems get very large, running m^2 iterations starts requiring a significant amount of time. This is partly because m^2 is simply a large number (for 180 constraints, $m^2 = 32{,}400$), but it is also because maintaining the priority queue becomes an expensive task. Profiling one instance of the 90 event problem revealed that approximately 45% of the time was spent inserting subspaces in the priority queue. Had we bounded memory usage, we could have avoided this issue. However, since bounding the memory causes `GAPS` to completely search one subspace before attempting another, the solution quality may have been lower.

Test 4: medium-sized unrestricted problems

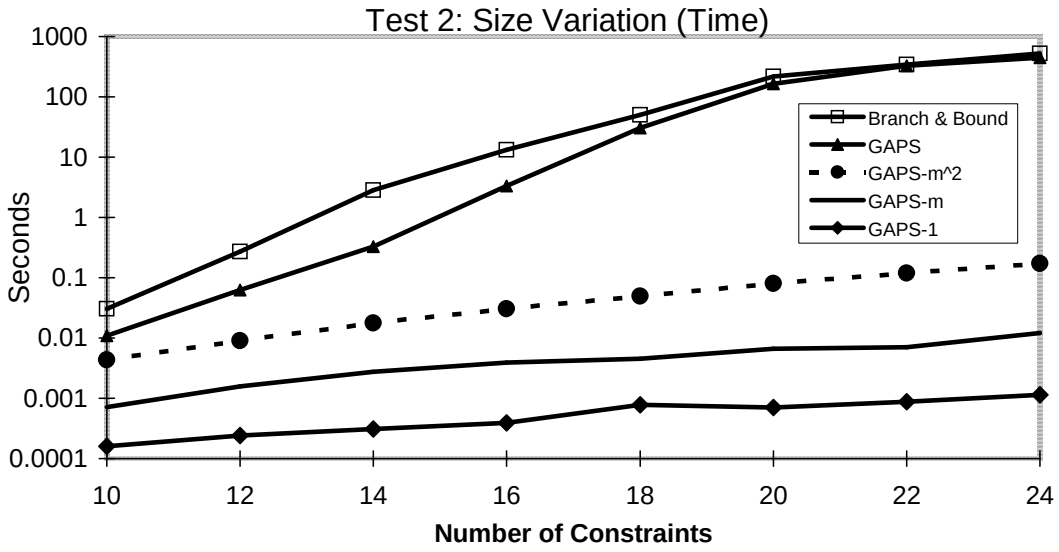
Figure 3.11 shows the results for the fourth test, which repeated the second test for unrestricted preference functions. Qualitatively, there is very little difference between the results from Test 2 and Test 4. The plots for Test 4 are not as "smooth"; meaning, their is slightly more variation between the trials.

In summary, the tests suggest the following:

- For smaller problems, `GAPS` finds the optimal solutions faster than `BB` (notice the log scale). As problem size increases, the time limit is reached more often



(a)

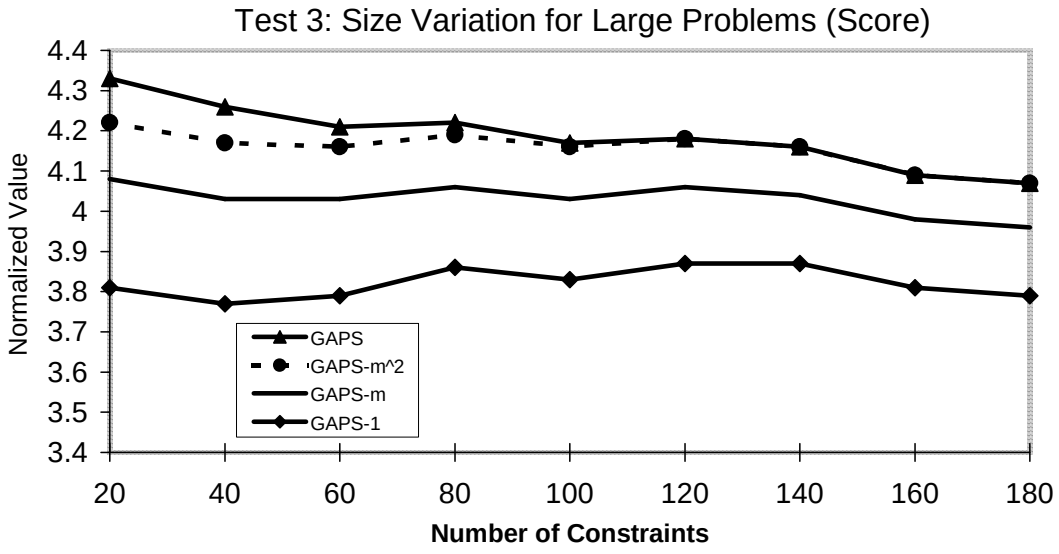


(b)

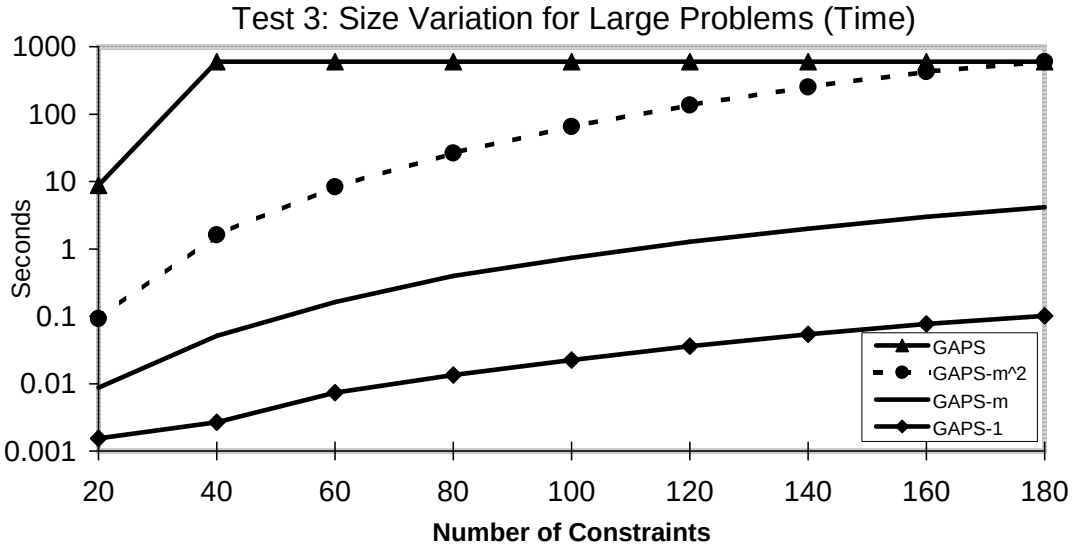
Figure 3.9: Solution quality and running times for the Test 2, which varied the number of constraints from 10 to 24.

and both GAPS and BB get suboptimal solutions. In these cases, solution quality for BB drops quickly relative to GAPS.

- GAPS has excellent anytime properties: GAPS-1 finds solutions averaging $>80\%$ of GAPS's value, while GAPS-m^2 averaged 96.5% and 99%.



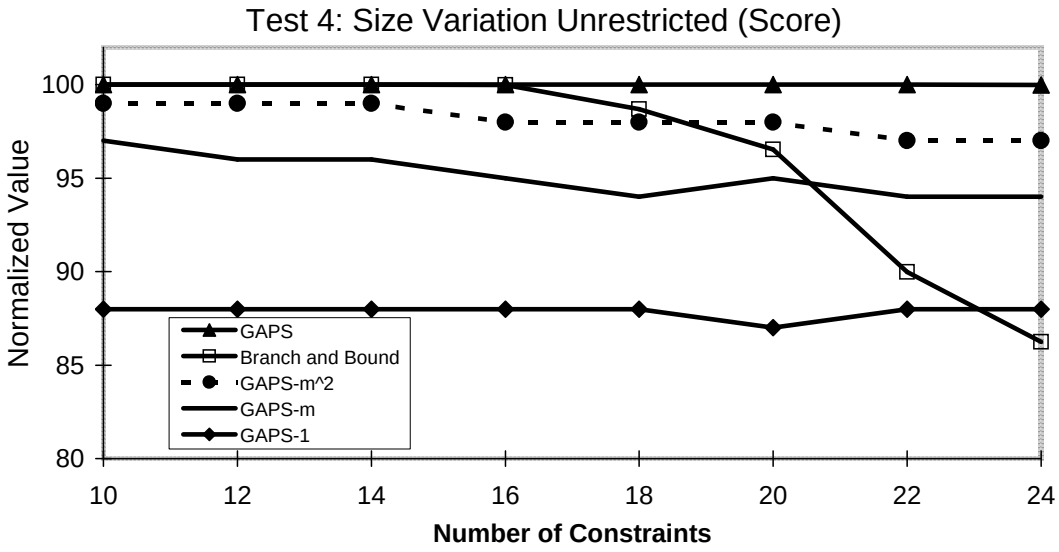
(a)



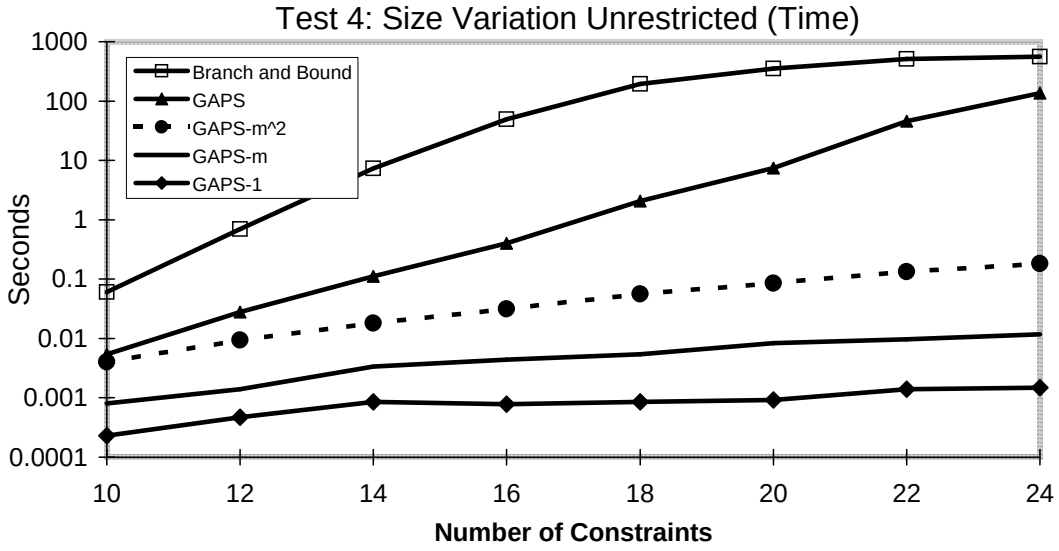
(b)

Figure 3.10: Solution quality and running times for the Test 3, which varied the number of constrains from 20 to 180.

- GAPS-1 solution quality equals `BB` for the largest problems tested. For reference, GAPS-1 averaged <2s in all cases where `BB` hit the 10 minute limit.
- The relative solution quality of GAPS-1, GAPS-m^2, and `GAPS` changes little as problem size increases further (Test 3).



(a)



(b)

Figure 3.11: Solution quality and running times for the Test 4, which repeated the second test for unrestricted preference functions.

The important result lies in the anytime performance of the `GAPS` algorithm. That solutions 80-99% of optimal can be found so quickly is important for planning and scheduling applications that use STPPs. Our comparison to `BB` is meant to show that `GAPS` compares well to an algorithm designed only to find the optimal

solution. Again, the danger is that when an algorithm is designed to have great anytime properties, other algorithms (without good anytime properties) will find the optimal solution significantly faster, but this did not prove to be a problem for GAPS.

3.5.4 Comparison to SAT-based strategy

Recent work has shown how SAT-based methods for solving CSPs and Temporal CSPs match or outperform standard CSP methods in some situations [3]. One recent SAT-based solver that has been modified to find utilitarian-optimal solutions to STPPs and DTPPs is the Ario SMT solver [77]. Ario solves constraint satisfaction problems that include two types of constraints: logical constraints over Boolean variables, and Unit-Two-Variable-Per-Inequality (UTVPI) constraints of the form $ax - by \leq d$, where $a, b \in \{-1, 0, 1\}$. Each hard constraint in the STPP preference projections can be represented as a pair of UTVPI constraints—one for each bound of the dual-bounded projected constraints. Logical constraints over Boolean variables that represent each UTVPI constraint encode the structure of the preference projections. Ario solves a STPP by incrementally building a component STP—very similar to the way in which the branch-and-bound algorithm presented in Section 3.5.1 solves STPPs. The difference is that Ario incorporates many powerful pruning techniques and variable ordering strategies that dramatically increase its efficiency.

The Ario algorithm was designed with the intent to find the optimal solution as fast as possible, which is in contrast to our goal of achieving excellent anytime performance. As we show in the tests below, both algorithms achieve their respective goals: for small problems, Ario find the optimal solution faster than GAPS; for large problems, the anytime performance of GAPS clearly surpasses Ario's.

To compare the two algorithms, we fed the problems from tests 2 and 3 above into

Ario and recorded the quality of the solution and the time required to find it. Given that Ario is compiled only in Linux, we ran the tests on a different computer. The Linux computer had two 2.1 GHz Pentium 4 processors with one of the processors allocated solely to the Ario and `GAPS`. We limited both `GAPS` and Ario to 10 minutes, but the Ario ran for an average of 12 minutes (the time check did not occur very often).

Figure 3.12 shows the results for the two tests.

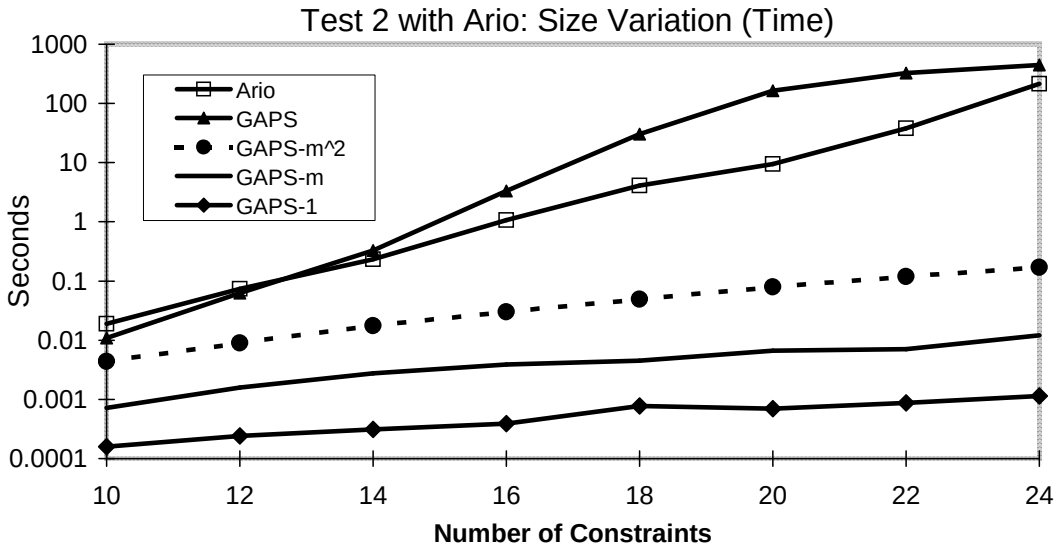
It is clear in Figure 3.12(a) that Ario finds the optimal solution faster than `GAPS`. We note that although the curve appears to be flattening for `GAPS`, `GAPS` continues its exponential increase. The curve tapers because of the 10 minute time cutoff imposed. Although `GAPS` quit early for a few problems containing 22 or 24 constraints, it did find the optimal solution for all instances.

Figure 3.12(b) shows the value of `GAPS` anytime performance. Once problem sizes exceed 60 constraints, a single iteration of `GAPS` (which requires $\sim$0.01 seconds for 60 constraints) produces better quality solutions than does Ario after 10 minutes. The performance of Ario depends heavily on the number of preference levels chosen for the problem. With fewer levels, Ario has much better performance. It is clear, however, that if anytime performance is important for a given application, `GAPS` is preferable.

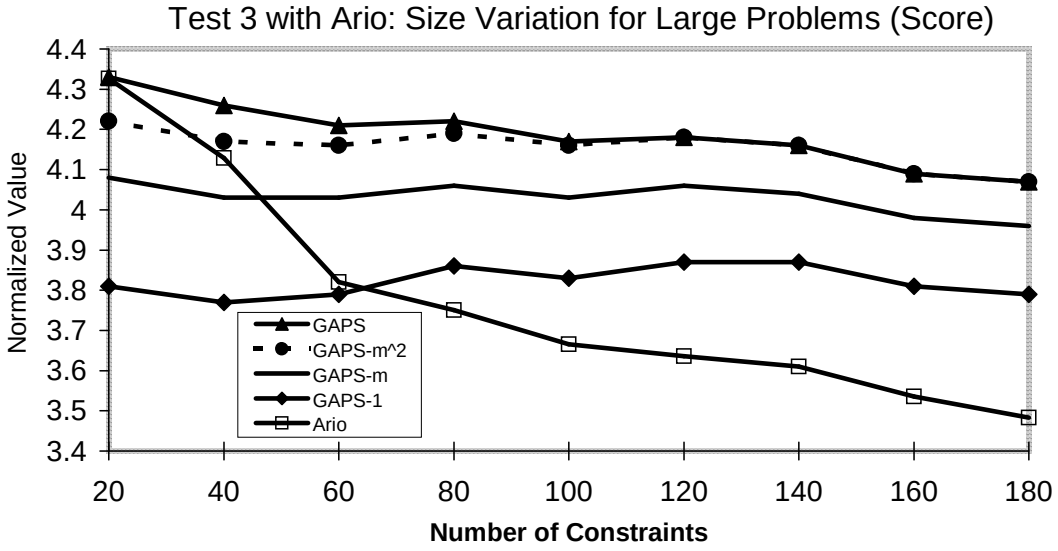
3.5.5 Outcome of Hypotheses

In this section, we evaluate the hypotheses originally presented in Section 1.4 that relate to `GAPS` and `STPP_Greedy`. The hypotheses were as follows:

1. For STPs with *semi-convex* preference functions and less than 10 preference levels per function, greedy search will find solutions above 80% of optimal for



(a)



(b)

Figure 3.12: (a) Results for rerunning Test 2 with Ario. Shows running time of Ario compared to running time of `GAPS` (a) Results for rerunning Test 3 with Ario. Shows normalized solution quality for Ario compared to normalized solution quality for `GAPS`.

95% of problem instances.

2. For STPs with *unrestricted* preference functions and less than 10 preference levels per function, greedy search will find solutions above 70% of optimal for

95% of problem instances. This property will hold in cases where the number of local minima in each preference function is less than 5.

Part 1 of this hypothesis turned out to be overly ambitious. For semi-convex problems, greedy search found solutions above 80% of optimal for an average of **84.7**% of problems with sizes up to 24 constraints—much less than the 95% predicted. Of course, decreasing the percent of optimality required significantly changes the result: greedy search found solutions above 70% of optimal for 97% of the same problems. There is no discernable trend for how this number changes for different problem sizes, so we cannot predict whether these hypotheses hold true for larger problems.

Part 2 of the hypothesis proved to be true. In unrestricted problems, greedy search found solutions with value above 70% of optimal for an average of 96% of problems. In this case, the trend indicates that `STPP_Greedy` more reliably finds solutions with values that exceed 70% of optimal as problem size increases.

Table 3.4 summarizes these results, which were extracted from the tests results in Section 3.5.3.

Number of Constraints	Semi-convex		Unrestricted
	% above 80% opt	% above 70% opt	% above 70% opt
10	84.0	97.0	96.0
12	76.5	96.5	98.0
14	83.3	95.8	97.5
16	87.1	95.4	97.5
18	84.0	97.4	98.0
20	86.1	95.9	99.5
22	88.7	99.5	98.5
24	88.0	98.5	100.0

Table 3.4: Results for the `STPP_Greedy` hypothesis. The second column shows the percentage of semi-convex problem instances in which `STPP_Greedy` found solutions with 80% of the optimal value. The third column shows the percentage in which `STPP_Greedy` found solutions with 70% of optimal value. The last column shows the percentage of unrestricted problem instances in which `STPP_Greedy` found solutions with 70% of optimal value.

The superior anytime performance of GAPS inspired the algorithm GAPD, which finds utilitarian solutions to DTPPs. In Chapter V, we describe GAPD and how its performance depends on these results.

CHAPTER IV

Finding Maximin-optimal Solutions to DTPPs

In this Chapter, we address the problem of finding maximin optimal solutions to DTPPs with unrestricted preference functions. We show that our algorithm, called `DTPP_Maximin` [61], has excellent worst-case complexity relative to hard DTP solvers and performs well empirically on randomly generated problems. While `DTPP_Maximin` makes no restriction on local preference function shape, we show that the worst-case analysis is better when the preference functions are semi-convex. In both cases, the analysis shows that preferences can be added to DTPs with low additional cost over solving the underlying DTP without preference functions. Thus, the algorithm achieves our overall practical requirement, which is that for applications in which the use of DTPs is feasible, the use of DTPPs must be feasible as well.

Our approach to finding maximin solutions to DTPPs (and bTCSPPs) combines the idea of projecting component STPs from DTPs with the idea of preference projections used in solving STPPs. Like the greedy algorithms presented in Chapter III, `DTPP_Maximin` does not operate directly on the preference functions; rather, it operates on *preference projections*, which allow efficient hard constraint algorithms to be leveraged. At a high level, `DTPP_Maximin` operates much like `GAPS` and `STPP_Greedy`: it searches the space of component STPs for the one that maximizes the objec-

tive function, which in this case is *min* rather than *sum*. Before presenting the algorithm, we review the concepts of preference projections and component STPs, adapting them to the DTPP case.

4.1 Preference Projections and Component STPs

Since a DTPP constraint is a disjunction of STPP constraints, the preference projection for a DTPP constraint is simply a set of preference projections—one for each of the STPP constraints that compose it. Recall that to obtain a preference projection for an STPP constraint, we first discretize the preference function range into a set of values A, called a *preference value set*. Then, we project the STPP constraint at each level $l \in A$. (See Section 3.1.1 for a more complete description.)

Each STPP preference projection is a tree of hard constraints. In Section 3.1.1, we identified each constraint in the tree by a 3-tuple $\langle k, l, p \rangle$, where k references the soft constraint from which it is derived, l is its preference level, and p is an index that distinguishes it from other constraints projected at the same level. To represent DTPPs, we need to update this notation to a 4-tuple $\langle k, d, l, p \rangle$, where the new element d indicates the disjunct in constraint k from which the hard constraint was projected. We now update the definition of a preference projection for the DTPP case:

Definition 10 (Preference Projection (Updated))**.** *Given a DTPP constraint* $C_k = C_{k,1} \vee C_{k,2} \vee \cdots \vee C_{k,u}$*, where* $C_{k,d} = \langle x_{k,d} - y_{k,d} \in [a, b], f_{k,d} \rangle$*, the projection at level* l *is* $\mathcal{P}_k[l] = \bigcup_d \mathcal{P}_{k,d}[l]$*, where*

- $\mathcal{P}_{k,d}[l] = \{C_{\langle k,d,l,1 \rangle}, C_{\langle k,d,l,2 \rangle}, \ldots, C_{\langle k,d,l,n \rangle}\}$,
- $C_{\langle k,d,l,p \rangle} = \langle x_{k,d} - y_{k,d} \in [a_p, b_p] \rangle$,

- $b_p < a_{p+1}$ *for* $0 \leq p < n$, *and*
- $\bigcup_{p=1}^{n}[a_p, b_p] = \{t | f_{k,d}(t) \geq l\}$.

We refer to the set of all hard constraints projected from a single DTPP constraint C_k *and preference value set* A *as the* **preference projection** *for* C_k, *denoted as* $\mathcal{P}_k = \{\mathcal{P}_k[l] : l \in A\}$. *Note that a disjunction of the STP constraints in* $\mathcal{P}_k[l]$ *form a DTP constraint.*

Although the definition is more involved, the concept of preference projections is unchanged. The updated definition is a generalization of the one presented in Section 3.1.1, which is the sub-case in which u in the above definition is 1.

The idea of a component STP is unchanged as well: a component STP is formed by selecting a single hard constraint from each DTPP preference projection. For example a component STP can be formed from a level 2 constraint from the projection of DTPP constraint 1, a level 4 constraint from DTPP constraint 2, and so on. As we will explain in Section 4.3, we have a particular interest in component STPs whose components were all projected from the same level.

Definition 11 (level component STP)**.** *A* **level component STP** *of a DTPP* $T = \langle V_T, C_T \rangle$ *is a set of STP constraints,* $S = \{C_{\langle 1,d_1,l,p_1 \rangle}, C_{\langle 2,d_2,l,p_2 \rangle} \ldots, C_{\langle m,d_m,l,p_m \rangle}\}$, *where* $m = |C_T|$, *and* $C_{\langle k,d_k,l,p_k \rangle}$ *is the* p_k^{th} *constraint in the projection* $\mathcal{P}_{k,d}[l]$. *Since each component of* S *has a projection level of* l, *we say* S *is a* **level** l **component STP** *of* T.

If, given a DTPP preference projection, we collect all constraints at level l, then the result is a hard DTP composed of level l component STPs.

Definition 12 (level l DTP)**.** *A* **level** l **DTP** *of a DTPP* $T = \langle V_T, C_T \rangle$ *is a set of*

DTP constraints, $\mathcal{P}[l] = \{\mathcal{P}_1[l], \mathcal{P}_2[l] \ldots, \mathcal{P}_m[l]\}$*, where* $m = |C_T|$*, and* $\mathcal{P}_k[l]$ *is the preference projection for constraint* $C_k \in C_T$ *at level* l*.*

Definitions 11 and 12 are important because of the nature of the maximin optimality criterion. Because maximin uses the *min* function to evaluate a schedule (and by extension, a component STP), the value of any schedule is the lowest local preference value for any constraint in the problem. Any constraint projected from a level higher than the minimum level does not contribute to the component STP's value. For example, if the preference functions for a four constraint problem output the local preference values $\{8, 9, 7, 2\}$ for a schedule S, then the value of schedule S is 2. Similarly, if a schedule S' produces the preference values $\{2, 2, 2, 2\}$, then the value of S' is 2 as well.

Thus, checking a component STP with non-uniform levels (e.g., $\{8, 9, 7, 2\}$) for consistency is unnecessary because there always exists some *level* component STP (e.g., $\{2, 2, 2, 2\}$) that is less restrictive and of the same value: if the level 2 component STP is consistent, no other component STP that contains a constraint with value 2 or less will have a higher value. If any component STP with maximin value v_{opt} has a solution, then some level v_{opt} STP has a solution as well. Thus, any search for the optimal component STP need only reason about level component STPs. Furthermore, the search for a consistent component STP at any level l will necessarily search the space of the level l DTP.

4.2 Upward Inconsistency Property

We can extend the parent/child relationships found in the preference projection trees to level component STPs and level DTPs. For any level l component STP, S_l, the component STP formed by the parents of all constraints in S_l will be a level $l-1$

component STP, S_{l-1}. The STP S_l is a the child of STP S_{l-1} and S_{l-1} is S_l's lone parent. Since every child constraint in a preference projection is as tight as or tighter than its parent, any child STP is as tight as or tighter than its parent. Similarly, the level l DTP is as tight as or tighter than the level $l-1$ DTP.

These definitions allow us to describe two properties of the preference projections that lead to `DTPP_Maximin`. First, note that since a level p component STP will be tighter than any ancestor level component STP at level $q < p$, any solution to the level p STP is also a solution to the level q STP. This is the basis of the Downward Consistency property:

Property 13 (Downward Consistency)**.** *If there is a consistent level p component STP in a DTP, then there is a consistent level q STP for all $q < p$. By extension, if a level p DTP has a solution, then each level q DTP for all $q < p$ has a solution.*

The converse property is Upward Inconsistency:

Property 14 (Upward Inconsistency)**.** *If a level q component STP is inconsistent, then any of its descendant component STPs at level $p > q$ will also be inconsistent. By extension, if a level q DTP contains no solution, then no DTP at level $p > q$ will contain a solution.*

This property holds because the higher-level STP is tighter than any of its lower-level ancestors. If no solution exists in an STP, no tighter version of that STP will contain a solution. As we will show below, the upward inconsistency property allows us to prune away a large part of the space searched by `DTPP_Maximin` each time an inconsistent STP or DTP is found.

4.3 Simple Strategies for Finding Maximin-optimal Solutions

As noted in Section 4.1, the search for maximin-optimal solutions can be confined to level component STPs. Furthermore, since all level component STPs reside in a level DTP, the entire space can be searched by searching the level DTPs—enabling the use of standard DTP-solving algorithms.

A simple strategy, therefore, is to just search each level DTP separately, starting at the lowest one and moving to the next each time a solution to the level DTP is found. When an inconsistent level DTP is found, the search can stop, because the upward inconsistency property ensures that no higher level DTP can be consistent. Given that there are $|A|$ preference levels, and therefore at most $|A|$ level DTPs, the worst-case time complexity of this simple algorithm is $O(|A| \cdot \mathit{complexity}(\texttt{solve-DTP}))$, where `solve-DTP` is any DTP algorithm developed to date.

The upward inconsistency and downward consistency properties for DTPs also enable a binary search through the level DTPs. To start, the middle level DTP is fully searched for a consistent STP: if one is found, then all level DTPs below the middle level can be pruned from the search (since they all have lower maximin values); if no consistent STP is found, upward inconsistency ensures that all level DTPs including and above the middle level can be pruned. Then the search continues on a new level DTP chosen by the standard binary search method. This strategy reduces the worst-case complexity to $O(log_2(|A|) \cdot \mathit{complexity}(\texttt{solve-DTP}))$ and is the DTPP analogue of the WLO algorithm [39], presented in Section 2.5.

4.4 DTPP_Maximin

The `DTPP_Maximin` algorithm closely resembles the first simple strategy listed above. The algorithm starts at the level 0 DTP and works its way upward, finding a solution for every level DTP it passes. The algorithm exits when it finds a solution for the highest level DTP or when it finds a level DTP without a solution. The difference between the two lies in the transition between level DTPs: when beginning search on a new level DTP, `DTPP_Maximin` leverages the upward inconsistency property to prune a portion of the search space at the new level.

In this section, we explain how to determine which parts to prune and describe the details of the `DTPP_Maximin` algorithm. In Section 4.5, we analyze the algorithm, showing how it improves on the worst-case complexity of the two simple strategies. We conclude with an experimental evaluation of `DTPP_Maximin`, comparing it to the two simple strategies.

`DTPP_Maximin` begins its search at the lowest level DTP and searches for a consistent component STP. If no solution is found, then the DTPP has no solution, since the lowest level DTP represents the most relaxed version of the problem. If a solution is found, the solution is stored and the search moves up to the next level DTP. However, the DTP at the new level is not fully searched: all component STPs that are children of the inconsistent STPs at the lower level are "skipped". The upward inconsistency property guarantees that the children of inconsistent STPs are also inconsistent, so there is no need to check them.

At the new level, the search starts with an STP that is a child of the consistent one just found at the level below. The search continues in the same manner: each time a consistent STP is found, the solution is stored, and search resumes on the next level.

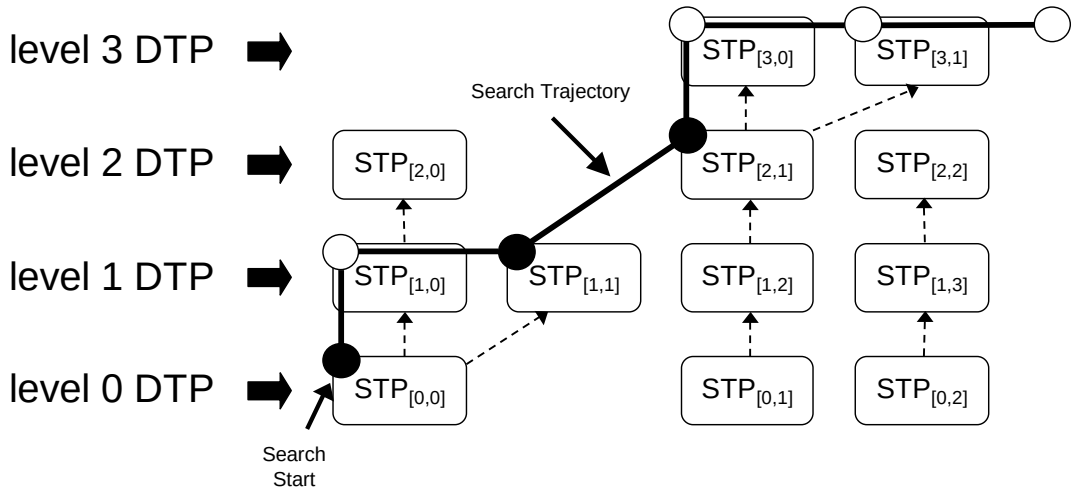


Figure 4.1: `DTPP_Maximin` example search trajectory: the search through the space of component STPs starts at the level 0 DTP and moves up each time a consistent STP is found. The dotted lines between STPs denote a parent/child relationship. The filled in circles represent consistent STPs and empty circles represent inconsistent STPs. The maximin optimal solution is $\text{STP}_{[2,1]}$.

The search ends when either all levels have been searched or when no solutions are found for a DTP at a given level. There is no need to search higher levels in this case, as the upward inconsistency property ensures that it is not possible for a solution to exist at one. Figure 4.1 shows an example path through the search space, with filled in circles representing consistent STPs and empty circles representing those that are not.

While Figure 4.1 gives a good intuitive picture of how the algorithm progresses, it is misleading when the intricacies of the algorithm are being learned. The difference lies in the distinction between checking entire component STPs for consistency (as suggested by Figure 4.1) and checking *partial* component STPs, as is done by leading hard DTP solvers and by `DTPP_Maximin`. See Section 2.3.2 for details on hard DTP solvers.

The `DTPP_Maximin` algorithm is shown in Figure 4.2. The line numbers correspond

```
DTPP_Maximin-Main(dtpp)
1. Produce preference projection for dtpp
2. cSTP = empty STP;
3. bestSTP = empty STP;
4. DTPP_Maximin(0, 0, true)

DTPP_Maximin(ci, p, moveToNext)
a.  IF p ≥ |A| RETURN bestSTP;
b.  IF moveToNext
1.      newDisjunct = getNext(ci,p) // Choose next unprocessed disjunct in constraint ci
2.      IF newDisjunct does not exist
3.          IF ci = 0 RETURN bestSTP // no solutions on this level, exit
4.          reset(ci, p) //Mark all disjuncts as unprocessed
5.          RETURN DTPP_Maximin(ci-1, p, true) // Try next disjunct in prev. constraint
6.      END
7.      cSTP = cSTP ∪ newDisjunct
c.  END
8.  IF cSTP is consistent
9a.     IF cSTP is a full component STP
9b.         bestSTP = Minimal Network of selected STP
9c.         p = p+1
9d.         [ci, cSTP] = getFirstChild(cSTP,p)
9e.         RETURN DTPP_Maximin(ci,p, false)
9f.     END
10.     RETURN DTPP_Maximin(ci+1,p,true) // Try first disjunct in next constraint
11. ELSE // Must backtrack
12.     cSTP = cSTP \ newDisjunct
13      RETURN DTPP_Maximin(ci,p,true) // Try next disjunct in same constraint
14. END
```

Figure 4.2: `DTPP_Maximin` algorithm for finding a maximin-optimal solution to a DTPPs. For comparison between `Solve-DTP` and `DTPP_Maximin`, the line numbers for steps in this figure correspond to the same steps in Figure 2.4. Added lines are labeled with letters.

to equivalent line numbers in the `Solve-DTP` algorithm in Section 2.3.2. Just as in `Solve-DTP`, the partial component STP that is repeatedly updated is called `cSTP`. The initial call to `DTPP_Maximin` starts the search at the first DTP constraint and at preference level 0. The third argument, `moveToNext`, is a boolean directive that will be explained shortly. The algorithm behaves exactly like `Solve-DTP` until it finds the first solution (line 9b). It then stores the solution, moves up one preference

level, and tries the solution STP's first child at the higher level (lines 9c-9e). This continues until a solution is found at the highest level (line a) or no more solutions are found (line 3).

To move up to the next level, the function `getFirstChild` finds the child of `cSTP` that occurs earliest in the search order and assigns `cSTP` to it. It then calls `DTPP_Maximin` with the `moveToNext` argument set to false, indicating that the next iteration should leave `cSTP` unchanged. The first child is retrieved by collecting the first child of each constraint in `cSTP`. The first child of each constraint is the one that would be retrieved first by the `getNext` function. Since not every STP constraint has a child, a child STP does not always exist at this step in the algorithm. In the case where the i^{th} constraint has no child, the children of the first $i-1$ constraints are added to `cSTP`, and the constraint pointer (ci) is set to i.

The key is to force the next call to `DTPP_Maximin` to resume the search at a point that "skips" all STPs at the new level that are descendants of STPs already searched at lower levels, to fully exploit the upward inconsistency property.

The only requirement for the `getNext` function is that it always returns the disjuncts of a constraint in the same order. The particular order does not affect the correctness and completeness of the algorithm. However, it does affect efficiency, both in terms of how fast an optimal solution is found and how long it takes for the algorithm to exit. A bad ordering can cause the algorithm to spend a lot time on the lower level DTPs searching parts of the space that do not have descendants at higher levels.

Consider Figure 4.3, which illustrates the effect of a bad ordering. In this network, consisting of only two constraints with a single disjunct each, the DTP at level 2 contains 9 component STPs, whereas the DTP at level 3 contains only 1. If a solution

exists at level 3, then a bad ordering of the disjuncts may require `DTPP_Maximin` to check up to 11 STPs for consistency.

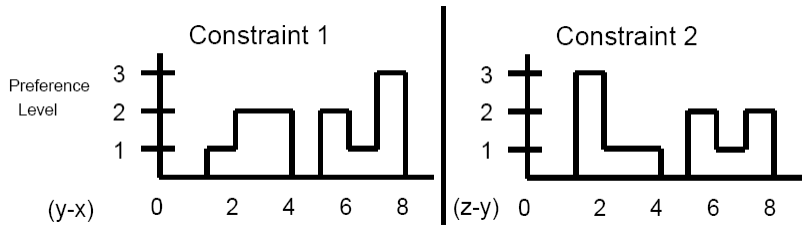


Figure 4.3: DTPP constraints whose lower level DTPs have much greater complexity than the upper level DTP.

The general issue is that the number of component STPs in the largest DTP may be much larger than the number in the solution-level DTP. The algorithm may search the largest DTP completely before reaching the solution level. This issue can be partially resolved by carefully ordering the disjuncts during the projection step of the algorithm. They should be ordered so that the STP constraints with descendants at high levels are ordered first, increasing the chances that if the optimal level is *opt*, STP with descendants in level *opt* is searched before STPs without descendants in *opt*. Figure 4.4 shows an algorithm `Index-DTPP-Projection` that properly chooses indexes for each interval in a preference projection. The 2D array returned by `Index-DTPP` is used by the `getNext` function to choose the next disjunct.

4.4.1 DTP pruning techniques

The `DTPP_Maximin` algorithm as presented in Figure 4.2 does not include any of the common pruning techniques used in meta-CSP DTP solvers. However, all pruning techniques found in the Epilitis solver [79] can be integrated into `DTPP_Maximin`, including *semantic branching*, *removal of subsumed variables*, *conflict directed backjumping*, and *no-good recording*. We implemented all techniques except no-good

```
Index-DTPP-Projection(P_k)
    highestIndex = 0 // Highest index assigned so far
    index[ ][ ] // array of indexes, first index denotes level, second denotes disjunct
    for each P_k[l] ∈ P_k starting with highest level and moving downward
        for each C_⟨k,l,p⟩ ∈ P_k[l]
            if constraint C_⟨k,l,p⟩ has a child C_⟨k,l+1,q⟩
                index[l][p] = index[l + 1][q])
            else
                index[l][p] = highestIndex
                increment highestIndex
            end
        end
    end
return index
```

Figure 4.4: Algorithm for indexing the disjuncts in the set of projected DTPs. Ensures high-level solutions are found quickly.

recording.

Implementing each technique increases the complexity of implementing the transition between level DTPs during search. Conceptually, however, the additions are straightforward and uninteresting. In general, the state required by each technique must be either reset or translated for use in the new DTP. The details of translating the state depend heavily on the particular implementation, so we do not discuss them here.

4.5 Analysis

As we suggested in Section 4.3, a simple algorithm for solving DTPPs is to run `Solve-DTP` for the DTP projected at the lowest preference level and then increment the level until a solution is found. This algorithm would have an O($|A|\cdot$ *complexity*(`Solve-DTP`)) worst-case run time, where *complexity*(`Solve-DTP`) is the complexity of solving the largest of the $|A|$ DTPs. A slight improvement can be made by performing a binary search through the preference levels, reducing the worst case

complexity to O($log(|A|)\cdot complexity$(`Solve-DTP`)). The complexity of `Solve-DTP` is based on the number of component STP consistency checks which can be as high as k^m, where k is the number of disjuncts per constraint and m is the number of constraints. Thus, $complexity$(`Solve-DTP`)) $= O(|X|^3 \cdot k^m)$, where X is the set of events in the DTP.

`DTPP_Maximin` improves on these algorithms through its use of the upward inconsistency and downward consistency properties. Any time `DTPP_Maximin` finds an inconsistent component STP, it will prune all of its descendants using the upward inconsistency property. Since the algorithm never drops down to a lower level, all of its ancestors are implicitly pruned using the downward consistency property. For any tree of component STPs, the maximum number of inconsistent STPs that would need to be searched is equal to the number of leaves in the tree. Therefore, to find the maximum number of inconsistent STPs searched in the entire DTPP, we need to count all component STPs that have no child. The number of consistent STPs processed is always at most $|A|$, since the algorithm moves to the next level DTP each time a consistent component STP is found. Obviously, the number of inconsistent component STPs dominates, so we find the complexity of `DTPP_Maximin` by calculating the number of childless component STPs.

A component STP is childless when at least one of its components (STP constraints) does not have a child. The calculation of the number of childless component STPs can therefore be made using properties of the preference projections. In the following analysis, we will show how to calculate the total number of childless component STPs, $\mathcal{Q}^-$, by summing the number of childless component STPs at each level l, $\mathcal{Q}_l^-$.

For the preference projection of constraint i, let the number of disjuncts at level

l be denoted as $k_{i,l}$, the number of disjuncts without a child at the next level as $k^-_{i,l}$, and the number with a child at the next level as $k^+_{i,l}$. Thus, $k_{i,l} = k^+_{i,l} + k^-_{i,l}$. Using this notation, the number of component STPs in the level l DTP is

$$\mathcal{Q}_l = \prod_{i=1}^{m} k_{i,l},$$

and the number of component STPs in a problem with $|A|$ preference levels is

$$\mathcal{Q} = \sum_{l=0}^{|A|-1} \mathcal{Q}_l.$$

Note that the k values can differ greatly between constraints at the same level and between levels of a single constraint. Consequently, the size of different level DTPs can differ greatly: if the projected constraints in one level have few children, then the size of the DTP at the next level will be must smaller; conversely, if the projected constraints in one level have many children, then the size of the DTP at the next level can be exponentially larger.

Now we calculate the number of childless component STPs at a given level l. Since we know that the first constraint in the level l DTP has $k^-_{1,l}$ disjuncts without a child, we know that at least $\frac{k^-_{1,l}}{k_{1,l}}$ of the component STPs are without children:

$$\frac{k^-_{1,l}}{k_{1,l}} \cdot \prod_{i=1}^{m} k_{i,l} \leq \mathcal{Q}^-_l,$$

where, again, $\mathcal{Q}^-_l$ denotes the number of childless component STPs at level l. This simplifies to

$$k^-_{1,l} \cdot \prod_{i=2}^{m} k_{i,l} \leq \mathcal{Q}^-_l.$$

Since the second constraint has $k^-_{2,l}$ disjuncts without a child, we know that of the remaining component STPs, at least $\frac{k^-_{2,l}}{k_{2,l}}$ of them have no children:

$$\frac{k^-_{2,l}}{k_{2,l}} k^+_{1,l} \cdot \prod_{i=2}^{m} k_{i,l} =\leq \mathcal{Q}^-_l.$$

This relation simplifies to

$$k^+_{1,l} \cdot k^-_{2,l} \cdot \prod_{i=3}^{m} k_{i,l} \leq \mathcal{Q}^-_l .$$

Continuing this recursion, we find that the j^{th} element of this series is the following:

$$\prod_{i=1}^{j-1} k^+_{i,l} \cdot k^-_{j,l} \cdot \prod_{i=j+1}^{m} k_{i,l}.$$

Summing every element of this series, we find the total number of childless component STPs at a given level l is

$$\mathcal{Q}^-_l = \sum_{j=1}^{m} \left(\prod_{i=1}^{j-1} k^+_{i,l} \cdot k^-_{j,l} \cdot \prod_{i=j+1}^{m} k_{i,l} \right).$$

Finally, for a given DTPP with $|A|$ preference levels, the total number of childless component STPs is

$$\mathcal{Q}^- = \sum_{l=0}^{|A|-1} \mathcal{Q}^-_l .$$

Thus, in the worst case, `DTPP_Maximin` must search $|A| + \mathcal{Q}^-$ component STPs. The value of $\mathcal{Q}^-$ depends on the structure of the preference functions. In the worst case, $\mathcal{Q}^- = \mathcal{Q}$, which means that it is possible `DTPP_Maximin` will not prune a single component STP. To achieve this worst case, the only consistent component STP in each level must be the last one searched and all component STPs in each level must be children of the lone consistent component STP in the level below. This is obviously a pathological case that would rarely occur. The ordering algorithm in Figure 4.4 prevents this particular case from occurring, but for other extreme cases, $\mathcal{Q}^-$ approaches $\mathcal{Q}$ even with the ordering.

4.5.1 Analysis of special cases

There are a few common special cases in which we can guarantee that the number of inconsistent component STPs is much less than $\mathcal{Q}^-$.

Semi-convex functions

The first case is the one in which all functions are semi-convex. Many real-world preference functions are naturally convex, and others can be approximated as such. Therefore, we (and other authors [39, 40, 52]) consider this an important class of problems.

When a function is semi-convex, the constraints in the corresponding preference projection never "split" into two or more constraints at the next level. In the notation above, this means $k_{j,p} \geq k_{j,q}$ for all $q > p$. Put another way, any level 0 constraint has at most one childless constraint in the levels above it. Thus, the maximum number of childless component STPs is equal to the number of component STPs in the lowest level: $\mathcal{Q}^- = \mathcal{Q}_0^-$.

This means that the worst-case complexity of searching the entire DTPP is equal to the complexity of searching only the bottom level DTP plus the complexity of solving $|A|$ consistent STPs: $O((|A| + \mathcal{Q}_0^-) \cdot |X|^3)$.

Top-level peaks

A second special case is when the only childless component STPs are in the top level DTP. This corresponds to functions in which every local maximum (peak) reaches the top preference level. In this case,

$$\mathcal{Q}^- = \mathcal{Q}_{|A|-1}^- = \mathcal{Q}_{|A|-1},$$

since $\mathcal{Q}_l^- = 0$ for all $l < |A| - 1$. Since the algorithm stops one level after finding the solution, we can say the complexity of `DTPP_Maximin` in this case is equal to the complexity of solving the level DTP above the solution level plus $|A|$ consistency checks.

Search-order partition

A final special case would likely occur less often in practice, but we mention it for two reasons: its conditions could be enforced when specifying preference functions; and even if its conditions are not fully met, the degree to which the conditions are violated gives an indication of how good or bad the worst case will be. We will first state the case and its complexity and then prove its complexity.

The special case holds when the constraints in each level DTP can be partitioned into two sets: those for which $k^- = 0$ and those for which $k^+ = 1$. Intuitively, the first set corresponds to the case in which all disjuncts have children, and the second set corresponds to the case in which exactly one disjunct has children. If all constraints do not satisfy at least one of those conditions, then the special case does not hold. Using a particular value and variable ordering, the complexity of `DTPP_Maximin` is equal to the complexity of solving the level DTP above the solution level plus $|A|$ consistency checks.

The value ordering required is that provided by the algorithm in Figure 4.4. The variable ordering is performed by a similar algorithm, only one that operates on entire constraints instead of disjuncts. The algorithm produces an ordering in which, for each level DTP, the constraints in which $k^+ = 1$ occur first, and those in which $k^- = 0$ come last. When searching a level DTP ordered in this way, all component STPs with children at the next level will be searched before those without children. Consequently, the optimal solution will be reached before any childless component STP has been searched (unless the optimal solution is childless). To prove the solution is optimal, the level DTP above the solution level will have to be fully searched.

Even when the conditions are not fully met, the property behind the savings in

this case still works to reduce the worst case complexity. Imagine that only a few constraints do not fall into one of the partitions. These constraints can be placed in the middle of the ordering, ensuring that *most* of the component STPs with children at the next level will be searched before those without. The greater the number of constraints that do not fit into a partition, the closer the complexity gets to $\mathcal{Q}^-$.

Semi-convex functions and search-order partition

For a problem that is both semi-convex and satisfies the search-order partition, the analysis gets slightly better: the complexity of `DTPP_Maximin` is equal to the complexity of solving the level DTP *at* the solution level plus $|A|$ consistency checks. This holds because with semi-convex DTPs, each level DTP is of equal or lesser size than the level DTP above it.

When functions are semi-convex, it is often possible to make the search-order partition. In fact, if each constraint is limited to two disjuncts (almost always the case in practice), the property is guaranteed to hold. With only two disjuncts, there are only three possibilities for childless children: both disjuncts have a single child ($k^- = 0$), only one disjunct has a child ($k^+ = 1$), or no disjunct has a child. If the last case occurs, then no component STPs exist at the next level, so only the first two cases are meaningful.

Anytime performance

Finally, note that because it starts its search at the lowest preference level, `DTPP_Maximin` has an anytime quality in all cases. In practice, since DTPs at lower levels are less tight, it will be easier to find a solution at a lower level. The algorithm can be interrupted any time after the first solution is found if available computation time is exhausted. Given that general DTPs are NP-Hard, this is an important

property.

4.6 Random Restarts

Random restarts have been shown to be effective in performing backtracking search [32, 37], similar to the one performed in `Solve-DTP`. A random restart is used to help the solver out of a "rut" caused by a bad constraint ordering. Typically, a restart is enacted by stopping the search, randomly reordering the constraints, and starting the search again. The question of when to restart depends on the type of problem and problem size. Often, restarts are enacted after the passage of a specified amount of time. Using this strategy, the algorithm may never complete, because the restart time may be less than the time required to solve the problem using the optimal ordering. To account for this possibility, the restart time is typically increased after each restart, ensuring that the restart time will eventually exceed the amount of time necessary to solve the problem.

To our knowledge, there does not exist a published study on the utility of using random restarts for solving DTPs. However, there is no reason to believe there is any property of DTPs that except them from favorable results in other problems. Thus, we can be reasonably confident that the use of random restarts will at least improve the ability of `DTPP_Maximin` to find the first solution at the lowest level.

The question remains as to whether allowing random restarts at higher levels will improve or diminish the overall performance. A priori, we can reason that random restarts would diminish performance because a restart throws away all information gained by searching the lower levels. Specifically, reordering the constraints destroys the upward inconsistency property, or at least destroys our ability to efficiently check for when the property holds. We can also legitimately reason that random restarts

would improve performance: a higher-level DTP may be much different than the lower-level DTP on which the variable ordering heuristics are based—the benefits of a better variable ordering (found by restarting) may outweigh the benefits gained by relying on the upward inconsistency property. In the next section, we answer this question empirically.

4.7 Experimental Results

The analysis above states only the worst-case performance of `DTPP_Maximin` relative to the complexity of the `Solve-DTP` algorithm. Experimental analysis is needed to determine an average case comparison, which is important in DTPs since such powerful pruning techniques exist.

4.7.1 Generating random DTPPs

To generate random DTPPs, we used a similar strategy to the one used to generate STPPs in Section 3.4.1: create a hard DTP, and then generate preference functions for each disjunct in each constraint.

The generator accepts the following parameter set: $\langle E, C, D_-, D_+, L, R_-, R_+, S\rangle$, where E is the number of events, C is the number of constraints, D_- and D_+ are the minimum and maximum bounds on any constraint, and the remaining elements define the preference functions.

We first generate a set of events $\{x_1, x_2, \ldots, x_E\}$. Then, we generate a set of C 2-disjunct DTP constraints (without preference functions) by creating one disjunct at a time. For each disjunct of each constraint, we randomly choose a pair of events, and then randomly choose an upper and lower bound for the temporal difference between the events from the interval $[D_-, D_+]$.

Once the hard DTP constraints are formed, we create a preference function for

each disjunct. Instead of creating the functions directly, we create the constraints of its function's preference projection. The lowest level of the preference projection is formed using the bounds that were chosen for the underlying STP constraint. To form a new constraint at the next level, we first calculate the width of the new constraint's interval by multiplying the width of the previous constraint's interval by a reduction factor, chosen from the interval $[R_-, R_+] \subset [0, 1]$. An interval of the newly calculated width is created and placed randomly within the original interval. With probability S, the new interval is split into two parts, and each interval is randomly placed. When S is 0, the interval never splits, and the result is a preference function that is semi-convex. We continue creating new preference levels until the calculated interval width for a new level is 0 (occurs often because we round to the nearest integer) or we hit the maximum number of preference levels defined by L.

4.7.2 Effect of random restarts

We start by determining whether random restarts are effective in improving the performance of `DTPP_Maximin`. We answer this question first to ensure that later experiments use the best version of our solver. We generated a set of 100 DTPPs of four different sizes: 15 events / 30 constraints, and 20 events / 40 constraints, 25/50, and 30/60. For each size, we tested the semi-convex case and the case where the split probability was 0.05. For each of these problems, we tested three variations of `DTPP_Maximin`: with no restarts (NO_RR), with restarts starting after 1 second (RR_1), and with restarts starting after 5 seconds (RR_5). The maximum number of preference levels for each function was set to 10. On average, that resulted in approximately 6 levels for each DTPP.

Table 4.1 shows the average and median run-times for the 100 instances of each

problem type. We also include columns that report the average amount of time spent finding the optimal solution.

	Average Runtime			Median Runtime			Time to Opt		
	NO_RR	RR_1	RR_5	NO_RR	RR_1	RR_5	NO_RR	RR_1	RR_5
15/30 Semi	0.038	0.034	0.035	0.016	0.016	0.016	0.018	0.015	0.015
15/30 0.05	0.041	0.042	0.039	0.016	0.016	0.016	0.019	0.022	0.018
20/40 Semi	0.443	0.331	0.393	0.110	0.110	0.109	0.249	0.154	0.200
20/40 0.05	0.876	1.117	0.966	0.171	0.188	0.187	0.288	0.188	0.265
25/50 Semi	14.800	2.510	4.899	0.547	0.531	0.531	1.254	0.622	0.849
25/50 0.05	20.012	9.249	9.791	1.172	1.094	1.204	6.997	2.319	2.105
30/60 Semi	40.768	28.101	17.595	3.516	1.641	3.500	16.066	5.124	5.780
30/60 0.05	163.414	102.200	104.521	26.485	6.391	11.766	41.317	21.083	21.623

Table 4.1: Average and median running times showing effect of random restarts for different problem sizes

We first note that random restarts improve average case performance in larger problems. For the largest problems, average savings were between 30-80%. For the 30 constraint problems, restarts had no effect because the problems were solved before the first restart occurred. For the 40 constraint non-convex problems, restarts hurt performance, which means the restarts were premature (RR5 did better than RR1 because it was premature less often).

Perhaps the more interesting fact is that a significant percentage of computation time was spent proving the optimal solution was found, rather than in finding the optimal solution itself. In the largest problems, approximately 80% of computation was spent proving optimality. This has two implications, one positive and one negative. First, it bodes well for anytime performance: the optimal solution is found early on, so the stopping the algorithm early may often produce no penalty. Second, it suggests that the level DTP directly above the optimal may be the hardest level DTP to solve. It is well-known that a CSP problem with only a few solutions or one that is "almost" consistent is the hardest type to solve. If many solutions exist, finding one is relatively easy; if the problem is massively over-constrained, then finding a conflict that proves it is unsatisfyable is relatively easy. The optimal level

DTP and the level DTP above it are two similar DTPs on either side of the line of consistency. Therefore, we can intuitively expect these to be the hardest to solve.

4.7.3 Comparison to binary search

We now test `DTPP_Maximin` against the two alternative algorithms mentioned in Section 4.5: to search each level DTP starting at the bottom level and moving up (B_UP); and performing a binary search through the level DTPs to find the consistent STP at the highest-level level (BIN). The last alternative corresponds to the `WLO` algorithm [39] developed for the STPP case (see Section 2.5).

At first thought, the comparison to B_UP is uninteresting, since `DTPP_Maximin` is essentially the same algorithm with extra pruning techniques. The comparison could be seen as a way to measure the pruning power of the `DTPP_Maximin`'s upward inconsistency property. However, the two algorithms do behave differently because of the variable-ordering heuristics used by `DTPP_Maximin`. The B_UP algorithm chooses a different ordering for each level DTP, whereas `DTPP_Maximin` is "stuck" with the variable ordering chosen for the bottom level. The exception to this is when a random restart occurs. Therefore, it is not certain at the outset that `DTPP_Maximin` will have better average case performance than B_UP.

How well `DTPP_Maximin` performs compared to the binary search algorithm depends partly on how "close" the preference functions are to being semi-convex: when the functions are semi-convex, the higher-level DTPs are typically either the same size or smaller than the lower-level DTPs; when the functions contain many peaks and valleys, the higher-level DTPs are typically *much* larger than lower level ones. This occurs because the lower level intervals in the projection tend to split into several intervals as the preference level rises. Since a binary search may jump to levels

much higher than the solution level, it pays a large price when the higher levels are much larger.

Playing counter to this reasoning is the empirical evidence reported in the previous section. We suggested that in practice, the average time requirement for solving the optimal level DTP and the level DTP above it dominate the other levels. If this is the case, `DTPP_Maximin` and BIN should have similar performance, since they both have to search the optimal and optimal+1 levels before exiting.

For this test, we used the same problems as in the previous test, and tested all algorithms using an initial random restart time of 1 second. Table 4.2 shows the average and median runtime for each test, as well as the amount of time needed to find the optimal solution.

	Average Runtime			Median Runtime			Avg Time to Opt		
	B_UP	DMax	BIN	B_UP	DMax	BIN	B_UP	DMax	BIN
15/30 Semi	0.038	0.033	0.031	0.031	0.016	0.016	0.018	0.014	0.020
15/30 0.05	0.057	0.041	0.055	0.016	0.016	0.016	0.022	0.017	0.025
20/40 Semi	0.297	0.293	0.251	0.125	0.110	0.109	0.127	0.136	0.170
20/40 0.05	0.852	0.853	0.633	0.125	0.187	0.156	0.223	0.208	0.407
25/50 Semi	3.082	3.139	2.816	0.407	0.531	0.391	0.642	0.781	1.815
25/50 0.05	12.908	12.070	15.884	1.110	1.047	1.187	2.560	2.254	7.633
30/60 Semi	28.164	27.100	32.492	3.234	3.156	2.078	5.342	5.007	12.746
30/60 0.05	96.763	99.739	213.889	5.906	3.984	4.578	19.087	18.364	78.598

Table 4.2: Average and median running times for bottom up search, `DTPP_Maximin`, and the binary search strategy.

Table 4.2 shows that for small problems, virtually no difference exists between the three algorithms. The only difference is that BIN requires more time to find the optimal solution, a difference that became more pronounced as problem size increased. It is clear that for non-convex functions `DTPP_Maximin` is superior to BIN in both time to optimal solution and the average and median total times. For semi-convex functions, the two tie in terms of median and average runtime, but `DTPP_Maximin` finds the optimal solution faster.

In general, the results show that although `DTPP_Maximin` is clearly superior to the alternatives in worst-case complexity, it shows little advantage in the average case. This follows from two related facts: that the DTP search depends heavily on variable ordering, and because random restarts destroy the pruning power of `DTPP_Maximin`. The main contribution of `DTPP_Maximin` is that its worst-case complexity is barely greater than the worst-case complexity of the optimal level DTP and the one above it. For real-time situations in which the upper bound of computation time must be known in advance, this may be of some value. However, for most practical situations (in which the worst-case time far exceeds available computation time), the pruning power of `DTPP_Maximin` has little effect; it effectively ties with the B_UP algorithm. Unexpectedly, the B_UP algorithm is superior to the BIN algorithm in both total time and anytime performance; with unrestricted preference functions, the upper level DTPs are often much larger than the lower ones, which can severely and negatively affect the BIN strategy.

CHAPTER V

Finding Utilitarian-optimal Solutions to DTPPs

The problem of finding utilitarian-optimal solutions to DTPs with Preferences is exceptionally difficult; in fact, two orthogonal special cases are known to be NP-Hard. First, if you consider the case in which none of the constraints have preference functions, the problem reduces to a hard DTP. DTPs are extensions of binary Temporal Constraint Satisfaction Problems (bTCSPs), which are NP-hard [23]. Second, when all constraints in a DTPP have only a single disjunct, the problem reduces to an STPP, which is also NP-Hard [39]. Consequently, there is little hope for an efficient solution, and we focus on developing an algorithm with good anytime properties.

In this chapter, we present the Greedy Anytime Partition algorithm for DTPPs (`GAPD`), which builds on the `GAPS` algorithm presented in Section 3.3 in three ways: first, the entire `GAPS` algorithm is used as an inner call; second, it borrows the ideas of partitioning the problem into smaller ones and focusing on anytime performance; and third, it uses the empirical results from `GAPS` to tune the algorithm's parameters. `GAPD` satisfies our general practical requirement, that for applications in which the use of DTPs is feasible, the use of DTPPs must be feasible as well.

We compare `GAPD` to a recent SAT-based solver that has been adapted to the DTPP problem and show that while the SAT-based solver finds the optimal solution

faster for small problems, the anytime performance of GAPD clearly surpasses it for large problems. We also compare to an integer programming solution to the problem and show the effectiveness of some efficiency techniques we present.

5.1 Example

To motivate this problem, we first refer you to the Autominder example presented in Section 1.6.2, in which an elderly woman has the option of exercising before or after a friend visits. The option, which is represented as a disjunctive constraint, is very typical of the *non-overlapping constraints* that occur frequently in Autominder domains and other planning domains. The result of non-overlapping constraints are problems in which events can be executed in many different orderings while respecting all constraints. In this example, the single disjunctive constraint results in only two possible orderings, but in general, the number of possible orderings increases exponentially with the number of disjunctive constraints.

We can better see this exponential increase in orderings by extending our Mars rover example in Section 1.6.3 to a DTPP. In this example, the task was to optimally schedule the start and end of a single experiment. Imagine that the number of experiments to schedule increases from one to three: the one originally described (which uses the same instrument as the previous experiment), and two others that use a second instrument. The three experiments can be executed in any order, but they cannot overlap. Therefore, three non-overlapping constraints are needed, one for each pair of experiments. The non-overlapping constraints are needed to express sentiments such as "Experiment 1 can occur before or after Experiment 2" and to designate which option is more preferable.

Whereas the STPP case had only a single ordering possible for the events to be

scheduled, the new situation allows six different orderings for the experiments. If we were scheduling four experiments, the six disjunctive constraints needed would allow 24 different orderings. This exponential increase in possible orderings is intuitive evidence of why finding utilitarian optimal solutions to DTPPs is so much more difficult in practice than the STPP case. As we showed in Section 2.6, DTPPs with mutually-exclusive disjuncts have very similar expressive power to the STPPs. However, problems more effectively described as DTPPs are often much more difficult to solve than those more naturally described by STPPs.

5.2 Connections to Solving DTPs

Our `GAPD` algorithm borrows heavily from concepts used to solve hard DTP problems. It is therefore useful to present the many ways in which aspects of the DTPP problem parallel aspects of the DTP problem. Our approach uses the fact that DTPP constraints are disjunctions of STPP constraints, just as DTP constraints are disjunctions of STP constraints. Given this parallel, we define a construct analogous to a component STP used in solving DTPs: the *component STPP* (hereafter cSTPP), which is an STPP formed using single a disjunct from each constraint.

Definition 15 (component STPP)**.** *A* **component STPP** *of DTPP* $T = \langle V_T, C_T \rangle$ *is a set of STPP constraints,* $S = \{C_{\langle 1,d_1 \rangle}, C_{\langle 2,d_2 \rangle} \ldots, C_{\langle m,d_m \rangle}\}$*, where* $m = |C_T|$*, and* $C_{\langle k,d_k \rangle}$ *is the* d_k^{th} *disjunct of the* k^{th} *constraint in* C_T*.*

Just as a DTP can be viewed as a collection of component STPs, a DTPP can be viewed as a collection of cSTPPs. The `Solve-DTP` algorithm presented in Section 2.3.2 effectively checks every component STP for a solution; if a solution exists in a component STP, that solution is also a solution to the DTP. Conversely, if a DTP has a solution, then the solution will also satisfy at least one of its component STPs.

A brute-force algorithm for solving a DTPP would simply run `GAPS` on each of its component STPPs and retain the highest-valued solution. A solution of value h for a cSTPP is also a solution of value h for the DTPP. Conversely, if the optimal solution for a DTPP has value h_{opt}, then the same solution will have value h_{opt} in at least one of its cSTPPs.

Table 5.1 summarizes the concepts that apply to solving both DTPs and DTPPs. The first six entries in the table present the concepts just described, while the last two will be described shortly. Because the `GAPD` algorithm borrows heavily from the general concepts used in solving hard DTPs, Table 5.1 provides a good reference for understanding the motivation behind the algorithm.

DTPs	**DTPPs**
Can be viewed as collection of component STPs	Can be viewed as collection of component STPPs
If all disjunctive constraints are non-overlapping constraints, cSTPs are distinct orderings	If all disjunctive constraints are non-overlapping constraints, cSTPPs are distinct orderings
S is solution to a cSTP $\Leftrightarrow$ S is solution to DTP	S is solution of value h to a cSTPP $\Leftrightarrow$ S is solution of value h to DTPP
DTP is inconsistent $\Leftrightarrow$ all cSTPs are inconsistent	DTPP is inconsistent $\Leftrightarrow$ all cSTPPs are inconsistent
DTP is consistent $\Leftrightarrow$ there exists a cSTP that is consistent	DTPP has optimal solution of value h_O $\Leftrightarrow$ there exists a cSTPP that has optimal solution of value h_O
Can solve DTP by searching all cSTPs	Can optimally solve DTPP by searching all cSTPPs
Searching a cSTP for a solution is fast, but there are exponentially many of them	Finding high-quality solutions to cSTPPs is fast, but there are exponentially many of them.
`Solve-DTP` finds all consistent cSTPs in a DTP	`Solve-DTP` finds all consistent cSTPPs in a DTPP by searching its underlying DTP

Table 5.1: Connections between solving DTPs and DTPPs.

5.3 GAPD

The GAPD algorithm progresses using the standard anytime strategy: find *a* solution as fast as possible, and then spend available time on computations that are most likely to improve on that solution in the shortest amount of time. At the highest level of abstraction, GAPD executes the brute force algorithm mentioned in the last section: it systematically searches all consistent cSTPPs within a DTPP. Before describing the full algorithm, we first explain how GAPD quickly finds the first solution, and the types of computations available for improving the solution.

We can efficiently find a first solution to the DTPP by using a standard hard DTP algorithm like Solve-DTP, described in Section 2.3.2. We first create a hard DTP that is guaranteed to have the same solutions as the original DTPP by projecting all DTPP constraints at the lowest level. We call this DTP the *underlying DTP*.

Definition 16 (underlying DTP)**.** *The* **underlying DTP** *of DTPP* $T = \langle V_T, C_T \rangle$ *is a DTP* $U = \langle V_U, C_U \rangle$*, where* $V_U = V_T$ *and* $C_U = \{\mathcal{P}_k[0] : C_k \in C_T\}$.

Recall from Definition 10 that $\mathcal{P}_k[p]$ is the projection of constraint C_k at level p. Intuitively, this projection simply removes the preference functions from each constraint, but the projection is necessary because we do not restrict preference functions to be continuous at the lowest preference level, i.e., the bottom level can "split" into multiple disjuncts[1].

Solutions to the underlying DTP are solutions to the DTPP and vice versa. This fact can be easily proved using the definition of the projection operation. Therefore, if we begin solving a DTPP by searching its underlying DTP for a solution, then the first solution to the DTPP will be found in the same amount of time as finding

[1]This occurs often with repulsive preference functions, which are described in Section 2.4.2.

a solution to a DTP, thus satisfying our general practical goal: any application that uses DTP solvers can upgrade to preferences with no penalty. Moreover, when computation time remains, the solution can be improved using the second phase of the `GAPD` algorithm, which will now be described.

Each disjunct of each constraint in the underlying DTP is a hard constraint that corresponds to the *root* of each disjunct's projection tree. In other words, if we create a preference projection for each disjunct, just as we did in Chapter III, the root of the projection tree is the same as the hard constraint produced during projection of the underlying DTP. Therefore, each solution to the underlying DTP, which is a component STP, has a corresponding cSTPP in the DTPP; the cSTPP is formed by collecting the preference projections rooted by each element of the component STP.

Definition 17 (corresponding cSTPP)**.** *Given a DTPP* $T = \langle V_T, C_T \rangle$*, its underlying DTP* $U = \langle V_U, C_U \rangle$*, and a solution to* U*,* $S = \{c_k : c_k \in C_k \in C_U\}$*, the* **corresponding cSTPP** *for* S *is a set of preference projections* $\overline{S} = \{\mathcal{P}_k : \mathcal{P}_k[0] = \{c_k\}, c_k \in S\}$*.*

Figure 5.1(a) shows the preference projection of a simple 2-constraint DTPP, and identifies its underlying DTP. Part (b) shows one possible solution to the underlying DTP—a component STP. Part (c) shows the component STPP that corresponds to the DTP solution in (b).

After finding a solution to the underlying DTP, we can quickly improve it by running one or more iterations of `GAPS` on its corresponding cSTPP.

At the point at which a single cSTPP has been found and partially solved using `GAPS`, there are two choices for the next computation: the cSTPP can be further searched using more iterations of `GAPS`, or the underlying DTP can be searched again for another solution. To handle the second case, we can modify the `Solve-DTP` algorithm to make it restartable; after it finds a solution, it saves its state, so that

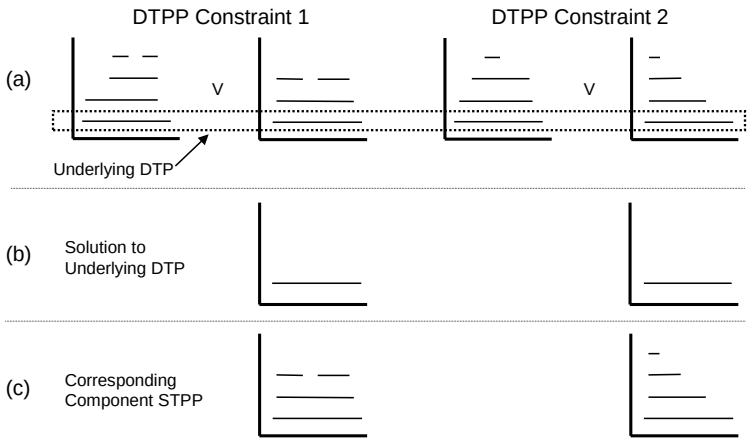


Figure 5.1: (a) The preference projection and underlying DTP for a simple 2-constraint DTPP. (b) A solution to the underlying DTP. (c) The corresponding component STPP for the solution in (b).

a subsequent call will find the next solution. Once found, the corresponding cSTPP for the new solution can be partially solved using `GAPS`. Recall that `GAPS` can be run for a few iterations, stopped, and then restarted again. The overhead of stopping and restarting is negligible, allowing an algorithm that must search a set of cSTPPs to jump between different cSTPPs with almost no penalty.

In general, after a set of cSTPPs have been identified, the algorithm must choose between finding additional consistent cSTPPs, or working on one of the cSTPPs already found. The anytime performance of the `GAPD` algorithm depends on how well this choice is made. Regardless of which choices are made however, the algorithm will eventually prove it has found the optimal solution. Optimality is proved when two conditions are met: the modified `Solve-DTP` algorithm completes its search of the underlying DTP, and all found cSTPPs have been fully searched with `GAPS`.

The basic choice between finding more cSTPPs and working on those already found is not the only factor that determines anytime performance for this basic algorithm. Before describing how we make the basic choice, we will discuss three

other factors which must be considered:

- An evaluation function is needed to determine which of the already found cSTPPs has the greatest potential for improving the current DTPP solution.
- The order in which the modified `Solve-DTP` algorithm searches its space determines how quickly cSTPPs containing high-valued solutions are found.
- Information gained from working on one cSTPP can be used to reduce search in others.

5.3.1 Evaluating cSTPPs

We now discuss the problem of evaluating the potential of a cSTPP, which will enable us to determine which cSTPP is most deserving of available computation time. Given that our goal is to develop an algorithm with great anytime properties, the determination of which of the already found cSTPPs is the "most promising" is critical to our success. To make this assessment, an evaluation function for cSTPPs is necessary. The following information can be useful as input to the function:

- An upper bound for the value of solutions within the cSTPP.
- The value of the best solution found thus far (if the cSTPP has been partially solved).
- The amount of time already spent solving the cSTPP.

The evaluation function can use this information to encode one of several high-level strategies. For example, it could place great value on cSTPPs that contain the best solution found so far, since only a slight improvement would increase the solution value for the entire DTPP. Alternatively, it could more heavily value cSTPPs with the highest upper bound, since, in theory, they have the greatest potential.

While both of these strategies have merit, the empirical results for GAPS in Chapter III suggest a third option: work on each consistent STPP for T iterations, and then move on to the next. This strategy is motivated by the GAPS results that show most gains are made in the initial iterations. Intuitively, we can expect a GAPS instance that has run just a few iterations to eventually produce a sizable increase in solution value, while an instance that has run for 1000s of iterations likely has found solutions close to the optimal value.

A drawback of this simple strategy is that it ignores how close the value of a cSTPP's best solution is to the "target" value. For a case in which GAPD has found a solution of value 200, the target value for any subsequent computation is a value of 201 or greater. If a cSTPP for which a single iteration of GAPS produced a value of only 50, it is very unlikely that the optimal solution to the cSTPP is 201 or greater. Yet, evaluating a cSTPP using the simple "lowest number of iterations" metric just described would attach great importance to the cSTPP.

To refine this simple strategy, we can create a model of how close a GAPS instance is to finding its optimal solution based on the number of iterations already performed. Such a model would allow the prediction of the cSTPP's optimal value, as well as a prediction of the amount of time needed to achieve the optimal value. Using the model, the next GAPS instance can be chosen using the expected number of iterations necessary to achieve the target value. In the example just mentioned, the model would tell us that the cSTPP with best value 50 has little chance of reaching the target.

The model we have chosen is again motivated by the GAPS experiments. Notice that for the results presented in Section 3.5.3, the first iteration achieves upwards of 80% of the optimal value, that m iterations produce 90% of the optimal value,

and that m^2 iterations achieves about 98% (m is the number of constraints in the problem). In general, the expected error (percent distance from optimal) decays exponentially with the base m log of the number of iterations. We can model the relationship between the expected error, E, and number of iterations performed so far, q, using the following formula:

$$E = \alpha e^{-\beta \log_m q},$$

where m is the number of constraints, α indicates the error after the first iteration ($q = 1$), and β is a constant that determines the rate of decay. The parameter values $\alpha = .25$ and $\beta = .9$ approximately fit the model to the results of the `GAPS` experiments.

While the above form of the model matches our intuition, it can be simplified to the following:

$$E = \alpha q^{-\frac{\beta}{\ln m}}.$$

The error E can be used in tandem with the value of the best solution, *best*, to predict the optimal value, *opt*, for the cSTPP:

$$\begin{aligned} E &= 1 - \frac{best}{opt} \\ opt &= \frac{best}{1 - E} \end{aligned} \tag{5.1}$$

We now illustrate how to use this model to quantify the potential of cSTPPs. Imagine a DTPP with 30 constraints and 3 cSTPPs eligible for the next computation:

	Best	q	E	E_T	$q_A = q_T - q$
cSTPP 1	200	1000	3.3%	2.8%	720
cSTPP 2	197	50	7.9%	6.0%	75
cSTPP 3	150	1	25%	-.5%	∞

The first two columns show the best value found so far for each cSTPP and the number of iterations (q) that have been performed. The third column shows the

expected percentage error between its best value and optimal value, E. Because 200 is the best solution found thus far for the DTPP, our target (given our anytime aspirations) is to find a solution with value 201. The fourth column, E_T, represents the error for the target value and can be understood with the following relationship:

$$E_T = 1 - \frac{target}{opt} = 1 - \frac{best + 1}{opt} \tag{5.2}$$

Combining equations 5.1 and 5.2, we can calculate E_T without knowing *opt*:

$$E_T = 1 - (1 - E)\frac{target}{best}.$$

Inverting the model, the target error is used to solve for the expected number of iterations necessary to reach the target value:

$$q_T = \left(\frac{E_T}{\alpha}\right)^{-\frac{\ln m}{\beta}}.$$

Therefore, the expected number of additional iterations necessary to reach the target is $q_A = q_T - q$. The cSTPP with the greatest potential, according to the model, is the one with the lowest q_A value. The last column of the table shows that we can expect cSTPP 1 will require 720 additional iterations to reach 201, while cSTPP 2 will require only 75. cSTPP 3 is not expected to reach 201, according to the model.

5.3.2 Controlling the Solve-DTP search

The series of solutions found by the modified `Solve-DTP` algorithm has a substantial influence on the anytime performance of `GAPD`. For large under-constrained problems, often thousands of solutions exist. Moreover, solutions that are adjacent in the DTP search space often have many or most components in common, meaning that the optimal solutions for their corresponding component STPs are likely

to have similar utilitarian values. If the subsequences of solutions that contain the high-valued cSTPPs are at the end of the search space, then anytime performance suffers.

Ideally, we would like to order the search space of the underlying DTP in such a way that guides the search to the solutions whose corresponding cSTPP have high values. However, considerable effort has been made to optimize the order in which DTP solvers navigate the search space. Specifically, variable ordering heuristics have been developed to reduce the amount of time required to find a solution or prove none exists. Changing these variable ordering strategies would invalidate a key contributer to their efficiency, compromising our overall practical goal.

Luckily, some effective variable ordering strategies define only a partial ordering of variables. Essentially, they define that a set of variables should be first, another set second, and so forth. This leaves open the possibility of reordering each set to suit our purposes without adverse effects.

Therefore, we can introduce the idea of random restarts into the backtracking search [37]. After finding a given number of solutions R, we restart the search, randomly reordering the variables before the search is restarted[2]. Although the variable ordering heuristics reorganize the variables back into their proper sets, the orderings within the set are changed, likely leading the solver to new solutions.

Restarting the search in this way destroys the completeness of the `GAPD` algorithm, since the DTP solver will possibly never finish its search. If completeness is important in the application, R can be increased after each restart, ensuring that eventually R will be greater then the number of solutions to the underlying DTP. Duplicate cSTPPs can be easily detected using the tree-based method for storing cSTPPs,

[2]See Section 4.6 for a more detailed explanation of random restarts.

which will be described below.

5.3.3 Sharing information between cSTPPs

The individual GAPS instances can benefit from information produced by other instances. For example, the best value found so far by GAPD can be used as a lower bound during the GAPS search. Any STPP subproblem produced during a GAPS search can be pruned if its upper bound is less than or equal to the value of GAPD's best solution. Using this additional pruning, a GAPS instance can finish much sooner than otherwise possible.

Many cSTPPs share common components; in fact, many pairs of cSTPPs differ by only 2 or 3 constraints. Information about their common components can be shared. For example, if a *partial* cSTPP (i.e., an STPP consisting of a subset of the cSTPPs constraints) belonging to several cSTPPs is solved exactly, the optimal value is an upper bound for the shared part of each full cSTPP. Therefore, if a partial cSTPP belongs to a large set of cSTPPs, a single computation can generate information relevant to the entire set. In this case, the tighter upper bound would better guide the GAPS search for all affected cSTPPs.

For large DTPPs, the previous two examples are of little use because there may be opportunity for only a few iterations of GAPS on each of the thousands of solutions. Thus, the pruning benefits enabled by having better upper and lower bounds are never realized. In these cases, a majority of the time is spent on the first iteration of each GAPS instance, and anytime performance is determined mostly by whether the DTP solver finds good solutions early on.

In situations where it makes sense to run only only a few iterations of GAPS on each cSTPP, it is possible to achieve savings if the algorithm avoids performing

the first iteration on unpromising cSTPPs. Unfortunately, our evaluation function presented in Section 5.3.1 depends on the value of the current best solution for the `GAPS` instance. To circumvent this requirement, we can estimate the value of the first iteration using the results of all other `GAPS` instances. Specifically, after the first iteration of `GAPS` for each cSTPP, we record the ratio of the solution value to the upper bound reported by `GAPS` before the iteration. By maintaining a running average of this ratio, we have a means to estimate the solution value for the first iteration of `GAPS` using the upper bound. Of course, this only makes sense if the variation in the ratio is fairly small. In a small test, which involved 12 problems of 3 different sizes, the variation was less than 7 percent in the smallest problems and less than 4 percent in the largest.

When using this strategy, which we call the *0 iteration technique*, the first iteration can be avoided in some cases, specifically those in which the predicted value for the first iteration is much less than the target value.

5.3.4 Choosing between GAPS and Solve-DTP

We now describe how to make the basic choice in the `GAPD` algorithm: between finding more cSTPPs and working on current cSTPPs. To make the choice, we must define the importance of finding more cSTPPs relative to the most promising cSTPP already found. We do so using a single threshold parameter, T, which denotes the expected number of iterations that any new cSTPP will require to meet the target. Consider the best of the three example cSTPPs in Section 5.3.1, where $q_A = 75$: if T is greater than 75, then the algorithm would choose to work on the best cSTPP; if less than 75, the algorithm would instead look for more STPPs.

In theory, the likelihood that a better cSTPP is yet to be found drops as the

GAPD(dtpp, α, β, T, R)

1. `udtp` $\leftarrow$ `project`(dtpp, 0)
2. $G = \varnothing$ // Initialize a set to hold `GAPS` instances for found cSTPPs.
3. REPEAT:
   (a) Using α and β, find `GAPS` instance $I = \arg\min_{I \in G} q_A(I)$.
   (b) IF $q_A(I) < T$
      i. run $q_A(I)$ `GAPS` iterations on I.
   (c) ELSE
      i. Search for a new solution in `udtp` using `Solve-DTP`.
         A. If solution is found, create a `GAPS` instance for it, and add the instance to G.
         B. Otherwise, set T to infinity.
      ii. IF R solutions since last restart, restart(`udtp`).

Figure 5.2: High-level `GAPD` algorithm.

number of solutions found increases. However, we can find no principled method to determine what percentage of solutions have been found. An obvious approach would be to measure the percentage of the search space already searched by the DTP solver. However, this erroneously assumes solutions are evenly spread through the space. Therefore, we leave T to be a constant parameter and empirically determine an acceptable value.

5.3.5 Algorithm

We now have all elements required to sketch the `GAPD` algorithm, which takes as input the DTPP and the parameters α, β, T, and R, where R is the initial number of STPPs to find before the DTP solver is restarted. Figure 5.2 presents a high-level description of the algorithm.

The algorithm exits when T is set to infinity (i.e., `Solve-DTP` has completely searched `udtp`) and when all `GAPS` instances in G have been fully searched.

5.3.6 Implementation Details

While Figure 5.2 displays all conceptual elements of `GAPD`, it masks significant details in how the algorithm manages its memory usage and stores the possibly thousands of consistent cSTPPs. This section can be safely skipped by those who have no interest in the dynamic version of the algorithm, which is discussed in Chapter VI.

We first discuss our method for storing consistent cSTPPs. In Figure 5.2, we simply stored them as the set G, leaving the method for managing the set unspecified. A straight set implementation is clearly unreasonable, given Step 3a requires all cSTPPs to be checked. Instead we store each cSTPP in a tree, which we call the *STPP tree*. The STPP tree for a DTPP has a dummy root plus one level for each constraint, giving it a depth of $m + 1$. Each node in the tree represents one STPP constraint from a DTPP constraint, and each full path of the tree (from root to leaf) represents a full cSTPP. An STPP tree can efficiently store many cSTPPs because cSTPPs commonly share multiple STPP constraints with other cSTPPs. Therefore, a single sub-path of the tree can encode parts of many cSTPPs.

Consider Table 5.2, which shows a set of four cSTPPs of a DTPP with four constraints: DC1, DC2, DC3, and DC4. Each element of each cSTPP is an STPP constraint d_Y^X, where X refers to the DTPP constraint and Y is the disjunct index within that constraint.

Notice that although each cSTPP is distinct, many share common STPP constraints. For example, cSTPP_2 and cSTPP_3 share the same disjuncts for DC1, DC2, and DC3. We can insert these cSTPPs into a tree sequentially and produce the tree shown in Figure 5.3. Each leaf in the tree corresponds to exactly one cSTPP.

While the tree in Figure 5.3 is much more compact than the tabular representation

	DC1	DC2	DC3	DC4
cSTPP_1	d_2^1	d_1^2	d_1^3	d_2^4
cSTPP_2	d_2^1	d_2^2	d_1^3	d_3^4
cSTPP_3	d_2^1	d_2^2	d_1^3	d_2^4
cSTPP_4	d_3^1	d_2^2	d_1^3	d_1^4

Table 5.2: A set of 4 STPPs in an STPP tree with four constraints. Each element of the STPP is a disjunct d_Y^X, where X refers to is the DTPP constraint and Y is the disjunct index within that constraint.

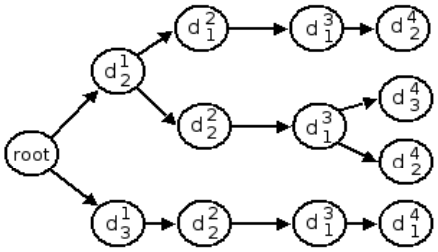


Figure 5.3: A set of 4 cSTPPs with four constraints inserted into an STPP tree.

in Table 5.2, redundancy still exists. For example, notice that for DTPP constraint DC3, only one of its disjuncts (d_1^3) participates in solutions, yet there are three nodes for this disjunct. A reorganization of the tree can collapse these nodes into one, as shown in Figure 5.4.

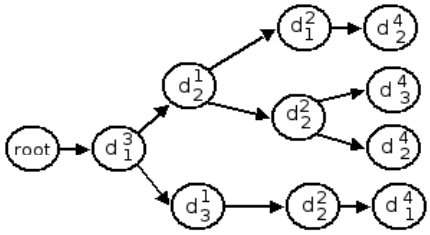


Figure 5.4: The tree in Figure 5.3 reorganized to reduce its size.

The new organization reduced the trees size from 13 to 11 nodes. Minimizing the size of the tree is not critical for the `GAPD` algorithm, but its size does effect the speed of some operations used in the dynamic version of `GAPD` discussed in Chapter

VI. For small problems, for which the upper bound reducing technique described in Section 5.3.3 is useful, a minimized tree allows fast discovery of partial STPPs that affect large numbers of cSTPPs.

The tree also acts as a priority queue, allowing efficient determination of the most promising cSTPP. Each node maintains the best solution value and highest upper bound for any of its descendant `GAPS` instances, as well as the priority value, q_A, for its most promising descendant. Finding the most promising cSTPP therefore requires time linear in the depth of the tree, as does all other operations that maintain these values.

After the Solve_DTP part of `GAPD` reports that all solutions to the underlying DTP are found, the upper bound in the root node represents the upper bound for the optimal solution value for the DTPP. This allows the algorithm to report a worst-case value for how far the current best is from the optimal solution.

Running many iterations of `GAPS` requires a significant amount of memory. Storing hundreds or thousands of partially solved `GAPS` instances becomes infeasible for any reasonably sized problem. Therefore, before executing steps 2(a) and 2(b) in Figure 5.2, `GAPD` first checks memory usage and frees enough memory before performing the operation. To free memory, unpromising `GAPS` instances are partially destroyed, retaining only the value of the best solution, its upper bound, and its priority value as determined by the evaluation function described in Section 5.3.1. If the instance ever becomes the most promising cSTPP (e.g., if all more valuable cSTPPs are completely solved), the `GAPS` instance can be recreated. Thus, the algorithm retains completeness even if some `GAPS` instances are destroyed once or multiple times before they are fully searched. To preclude the situation in which instances are rapidly destroyed and recreated, the priority value q_A of a cSTPP is artificially

increased slightly when its GAPS instance is destroyed. This encodes the fact that there is extra cost in reconstituting a GAPS instance.

5.4 Experimental Results

The first goal of our experiments is to show which parameters and strategies maximize GAPD's anytime performance. The second goal is to show how GAPD performs relative to a SAT-based solution and an integer programming solution, both of which are designed to find the optimal solution quickly, rather than to achieve anytime performance.

5.4.1 Optimal value of T

To test how our model-based strategy behaves with respect to the T value, and to show how it compares to the "lowest number of iterations" strategy presented above, we generated a set of 50 random DTPPs of the same size, and solved them using different strategies. For each strategy, we ran GAPD on each problem for 90 seconds, recording the value of the best solution every 5 seconds. We tested the following two strategies:

1. Work first on each cSTPP for T iterations, and then search for more solutions to the underlying DTP (LOW_ITER)

2. Work first on cSTPP with lowest expected number of iterations needed to achieve the target value (MODEL)

The effectiveness of the two strategies depend on the value of T, the number of iterations that determines how often new instances are sought. We tried several values, including 2 and multiples of 5 from 5 to 50. We denote the strategies for each value T by LOW_ITER(T) and MODEL(T) in the graphs.

We tested three different problem sizes: (1) 15 events and 30 constraints; (2) 20 events and 60 constraints; and (3) 75 events and 120 constraints. All preference functions were semi-convex with a maximum of 10 preference levels. The test includes the random restart technique, with R initially set to 30. The test does not include the 0 iteration technique, whose effect we will explore separately in a later section.

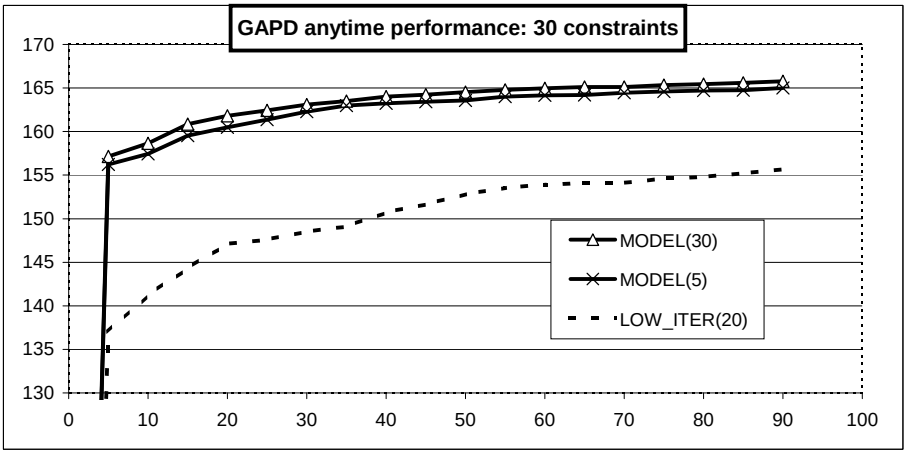


Figure 5.5: Comparison of anytime performance for different T values and strategies for problems with 30 constraints. We show the best and worst T values for the MODEL strategy and the best T value for the LOW_ITER strategy.

Figure 5.5 shows the results for the 30 constraint problems. We do not show every value of T tested, instead we show only the curve for the best and worst T values for the MODEL strategy and the best T value for the LOW_ITER strategy. As you can see, there is very little difference between the best and worst T value in the 30 constraint case. This small difference exists because of the small problem size: smaller problems usually have fewer cSTPPs, which means the algorithm has time to find a large percentage of all solutions (unless it is an extremely under-constrained problem). Therefore, the decision of whether to work on the current cSTPPs or find new ones is less important. If the value of T was increased to a more extreme value,

say 1000, the performance of the MODEL strategy would begin to suffer because it would dramatically increase the time it takes to find a significant number of cSTPPs.

Notice that the LOW_ITER strategy performs much worse because it allocates equal time to all cSTPPs. Since the choice between finding more cSTPPs and working on those already found is less important, the anytime performance depends almost completely on the choice of which cSTPP to work on next and for how long.

Figure 5.6 shows the results for the 60 constraint case.

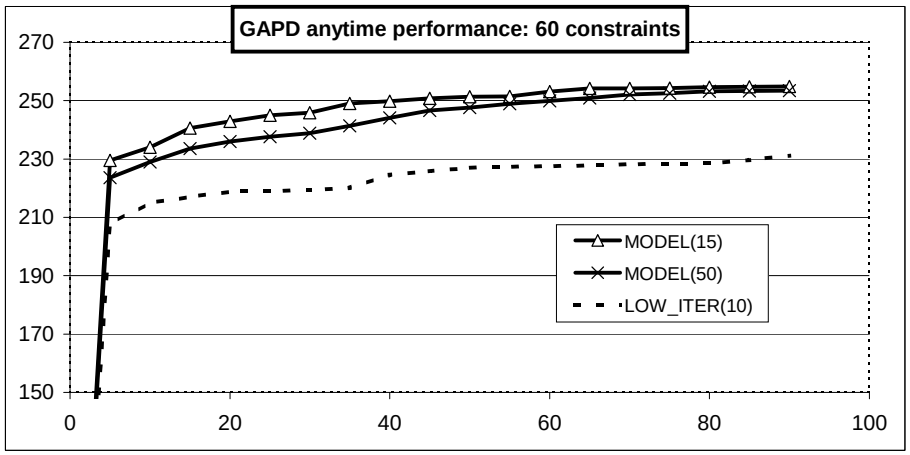


Figure 5.6: Comparison of anytime performance for different T values and strategies for problems with 60 constraints.

Doubling the number of constraints dramatically increases the number of possible cSTPPs, so the basic choice between finding more cSTPPs and working on current ones becomes much more important, meaning the T value is more important. As a result, the MODEL strategy does not beat the LOW_ITER strategy by as much (LOW_ITER achieves 95% of the MODEL value at the 20 second mark, as opposed to 90% in Figure 5.5). In addition, the difference between the best and the worst MODEL strategy grows as well.

For the 60 constraint case, $T = 15$ produced the best performance, which means that if a cSTPP is expected to hit the target value with 15 or less additional `GAPS`

iterations, those iterations will be performed before looking for new cSTPPs. The plot suggests that if T is much less than 15, promising cSTPPs are left unprocessed and time is instead spent looking for more solutions; if T is much greater than 15 (e.g., 50), too much time is wasted on searching cSTPPs and many high valued cSTPPs are never found.

Figure 5.7 shows the results for the 120 constraint case.

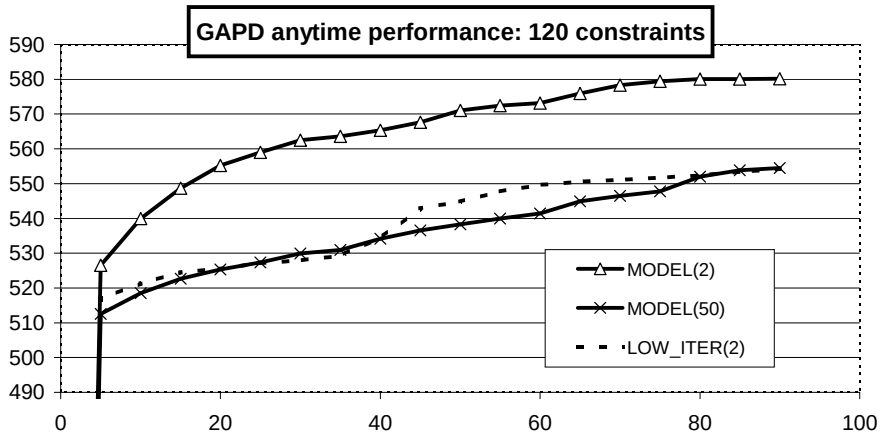


Figure 5.7: Comparison of anytime performance for different T values and strategies for problems with 120 constraints.

When problems get as large as 120 constraints, the number of consistent cSTPPs can be extremely large; moreover, it can take much longer to find them. With large problems, the anytime performance is mostly determined by the DTP solver's ability to find high-valued component cSTPPs. Executing more than 1 or 2 iterations of `GAPS` is less valuable than searching for a higher-valued cSTPP. Therefore, very small T values work well for both the MODEL and LOW_ITER strategies.

The best MODEL strategy does much better than the worst MODEL strategy, which is even outperformed by the LOW_ITER strategy. Both the worst MODEL strategy and the LOW_ITER strategy are spending too much time working on already found cSTPPs. It is interesting to note that when $T = 2$, there is very little difference

between the decisions made using MODEL and LOW_ITER: MODEL(2) performs 1-2 iterations on each found cSTPP, while LOW_ITER performs exactly 2 iterations on each. The extra iteration performed by the LOW_ITER strategy significantly affects the anytime performance, which is surprising given their similarity. Given that the 1 iteration savings of the MODEL strategy is significant, we can predict that the potential effect of the 0 iteration strategy is significant as well.

We emphasize that the T parameter does not have a single optimal value for all problem sizes. As the number of consistent cSTPPs increase, the optimal value for T will decrease, since the probability of having found the optimal cSTPP is lower. In fact, it may be possible to set T automatically based on the number of constraints in the problem.

5.4.2 Effect of random restarts

We now show the effect of the random restart technique on anytime performance for different problem sizes. Figure 5.8 shows how the best strategy in the previous test fared when the random restart technique is turned off.

As you can see, the effect of random restart is significantly positive for large problems. In the 30 constraint case, the effect is small, since, as we described in the previous section, `GAPD` has time to explore a significant portion of the consistent cSTPPs. In fact, toward the end of the run, there is no difference between the two strategies, because the set of found cSTPPs in each case is approximately equal. In the 120 constraint case, random restart is much more important. The restarts help ensure that the set of found cSTPPs is a more representative sample of all consistent cSTPPs. We conclude that the negative effects of starting the DTP search over are outweighed by the benefit of having diversity in the found cSTPPs.

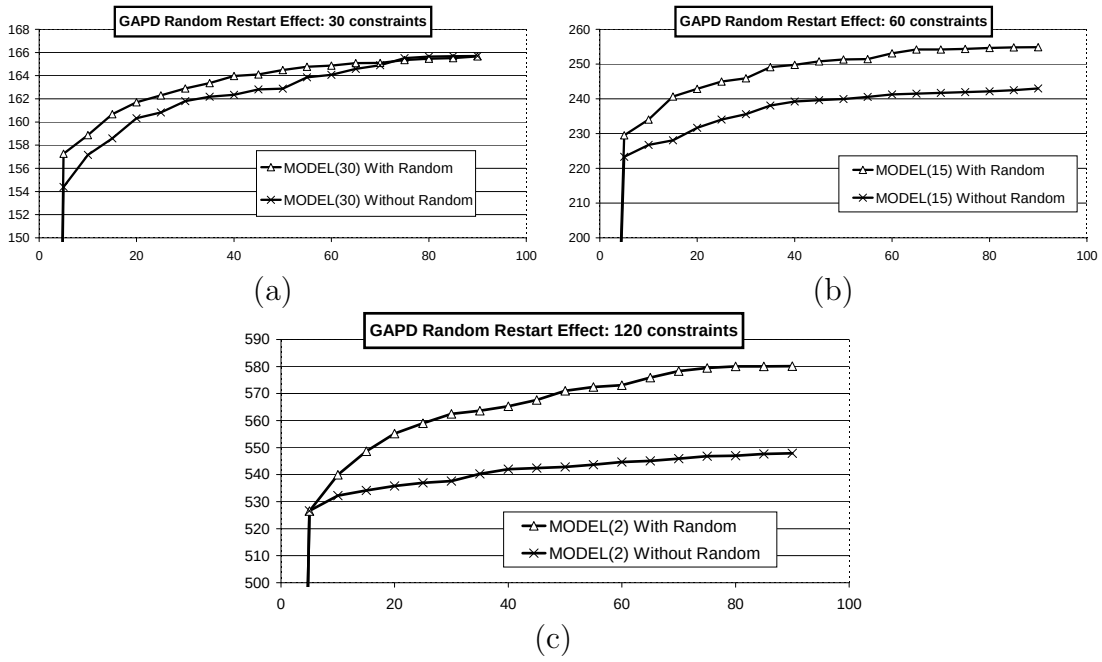


Figure 5.8: Anytime performance with and without the random restart technique.

5.4.3 Effect of 0 iteration technique

In Section 5.3.3, we described a technique for estimating the value of having performed a single iteration on a cSTPP. The technique allows `GAPD` to skip the first iteration on many cSTPPs, leaving time for finding new cSTPPs. We did not include this technique in the tests above because it can have a dramatic positive or negative effect on anytime performance and the optimal value of T. The technique can perform its intended function, which is to reduce time spent on unpromising cSTPPs, but it can also prune cSTPPs that contain high-valued solutions. Therefore, we examine this technique separately for the 120 constraint case, where we expect it to be most valuable.

We hypothesized that enabling the 0 iteration technique changes the optimal value of T. Since less time is wasted on unpromising cSTPPs, more time is available to either find more solutions or perform additional iterations on promising cSTPPs.

Given that the optimal T is so low for the 120 constraint case, we would expect that an increase in T might improve performance.

In Figure 5.9, we compare the MODEL(2) strategy that was optimal without this technique to different values of T using the technique. We tried T values of 2, 3, and 4. The figure shows that the optimal value of T did not change for this case. It also shows a significant improvement in the early anytime performance.

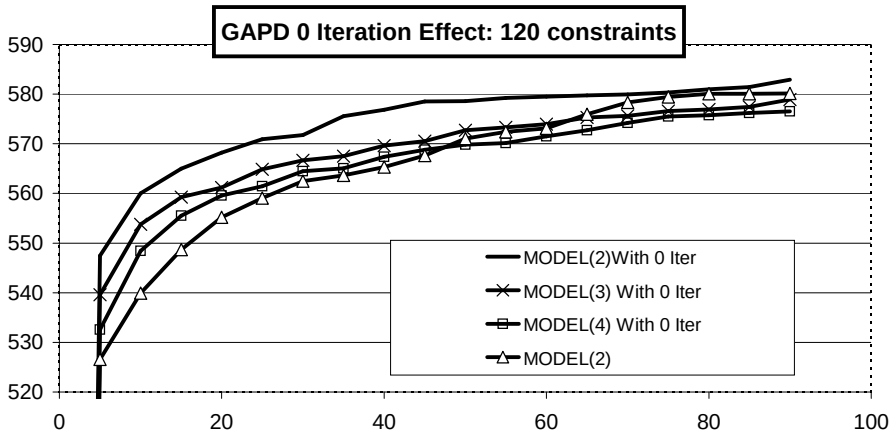


Figure 5.9: Anytime performance with and without the 0 iteration technique.

The increase in anytime performance was due to the increased number of cSTPP found during search. Without the 0 iteration technique, the average number of solutions found was 2570, compared to 8402 when the technique was on. For 93.5% of the cSTPPs found, 0 iterations were performed, showing that the technique substantially changes where `GAPD` spends most of its time. When not using the technique, 91% of the computation time was spent on performing `GAPS` iterations; when using it, only 30% of the time was spent inside `GAPS`. Based on these numbers, we expect that many promising cSTPPs were overlooked. In fact, when we compared on a problem by problem basis, we notice that for a few instances, using the 0 iteration technique caused a slight reduction in solution values. On average, though, the technique is

effective in guiding the search to better cSTPPs.

The technique can be made more conservative by adding a constant to the running average of the best to upper bound ratio. We attempted this using constants that reflected the variation in ratio (as discussed in Section 5.3.3), but this did not improve performance.

5.4.4 Comparison to Integer Programming formulation

Like many optimization problems, the problem of finding utilitarian-optimal solutions to DTPPs can be encoded as an integer program. For the encoding, we extended the linear programming formulation for convex STPPs described in [52], adding binary variables representing each disjunct in each DTPP constraint and constraining only one disjunct for each constraint to be active. To handle non-convex preference functions, we divided each non-convex function into a set of convex regions, and treated each region as a separate disjunct. Appendix B describes the details of this formulation.

To compare `GAPD` with the Integer Programming method, we randomly generated 50 DTPPs for two different problem sizes: 10 events/15 constraints, and 15 events/30 constraints. For each instance, we ran `GAPD` and CPLEX for 60 seconds, recording the time after each improvement to the solution.

Figure 5.10 shows the average anytime performance for the 10 events/15 constraints problem size. CPLEX did not find any solutions to the 15/30 problem size instances.

Figure 5.10 clearly shows that, for at least this IP formulation, CPLEX does not compete well with `GAPD` in terms of anytime performance. Although better formulations may exist, it is hard to imagine a formulation that uses fewer integer

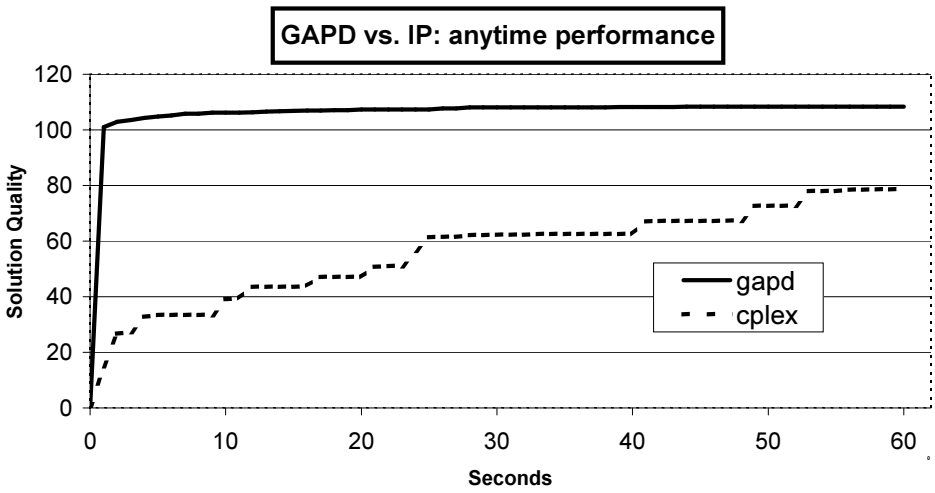


Figure 5.10: Solution quality and running times for the GAPD vs. CPLEX test.

variables, which, as a rule of thumb, should be minimized to improve speed.

5.4.5 Comparison to SAT-based strategy

Recent work has shown how SAT-based methods for solving CSPs and Temporal CSPs match or outperform standard CSP methods in some situations [3]. One recent SAT-based solver that has been modified to find utilitarian-optimal solutions to DTPPs is the Ario SMT solver [77]. Ario solves constraint satisfaction problems that include two types of constraints: logical constraints over Boolean variables, and Unit-Two-Variable-Per-Inequality (UTVPI) constraints of the form $ax - by \leq d$, where $a, b \in \{-1, 0, 1\}$. Each hard constraint in the DTPP preference projections can be represented as a pair of UTVPI constraints. Logical constraints over Boolean variables that represent each UTVPI constraint encode the structure of the preference projections. Ario solves the DTPP by incrementally building a component STP—very similar to the way in which the branch and bound algorithm presented in Section 3.5.1 solves STPPs. The difference is that Ario incorporates many powerful pruning techniques and variable ordering strategies that dramatically increase its efficiency.

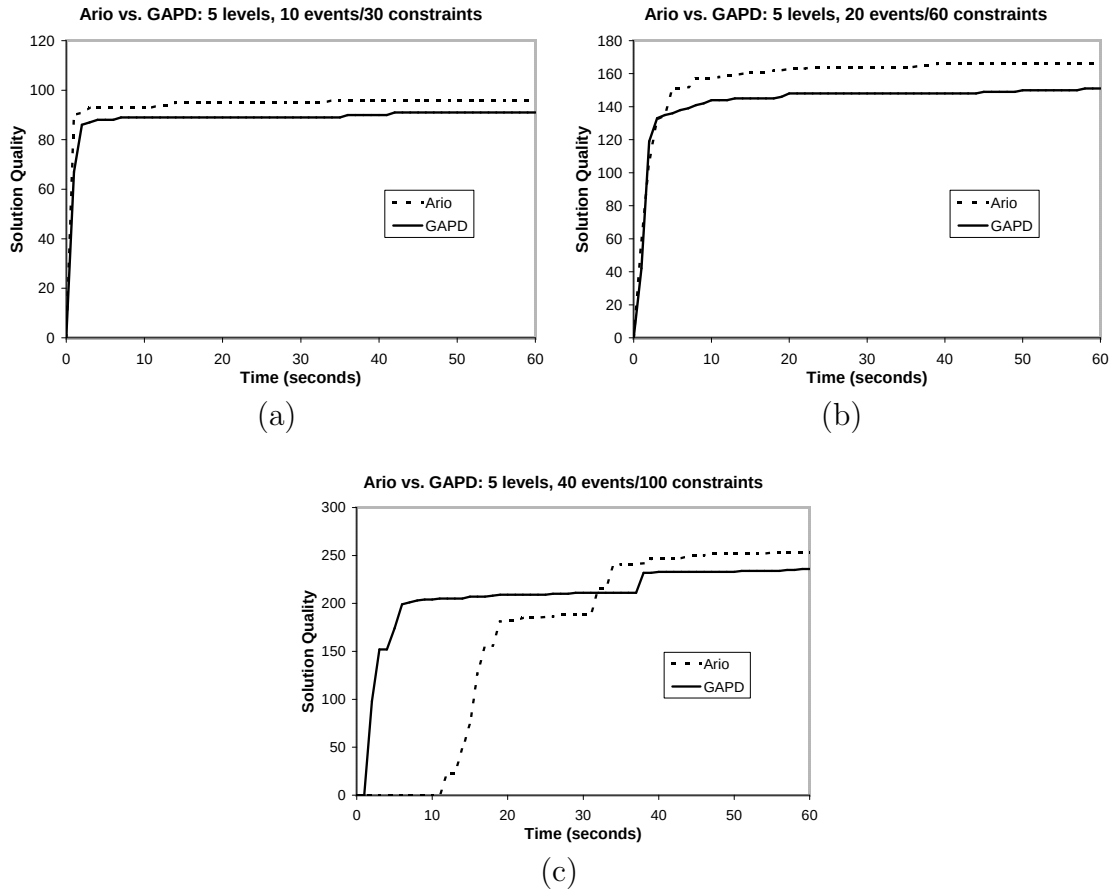


Figure 5.11: Anytime comparison of GAPD and Ario for problems with 5 preference levels.

To compare the two solvers, we randomly generated 50 DTPPs for nine different problem sizes chosen by varying two dimensions: the number of preference levels (5, 10, and 15) and the size of the underlying DTP (10 events/30 constraints, 20 events/60 constraints, 40 events/100 constraints). For each instance, we ran both solvers for 60 seconds, recording the time after each improvement to the solution.

Figures 5.11, 5.12, and 5.13 show the average anytime performance for the problem sizes with 5, 10 and 15 preference levels, respectively.

The results show that for small problems (in Figures 5.11(a), 5.11(b), 5.12(a), and 5.12(b)), Ario outperforms GAPD. For slightly larger problems (Figures 5.11(c) and 5.12(c)), GAPD has better anytime performance for the short term, but Ario

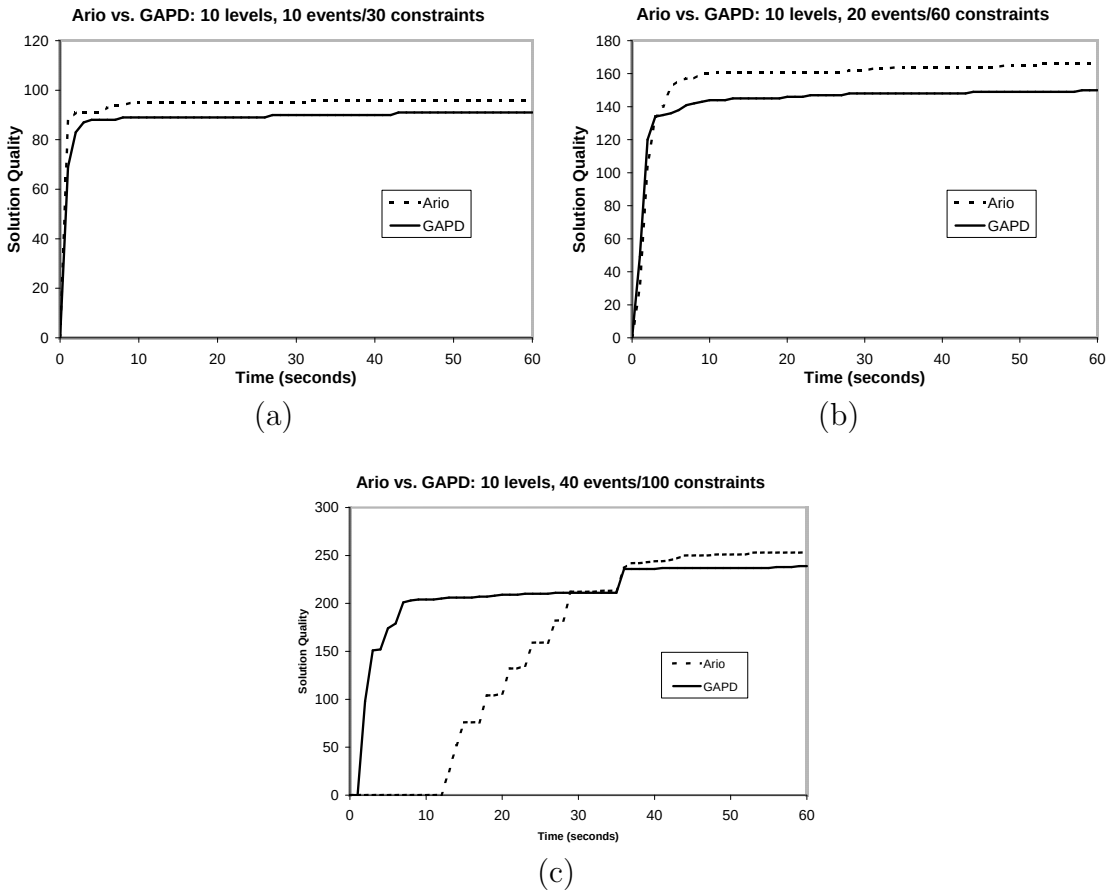


Figure 5.12: Anytime comparison of GAPD and Ario for problems with 10 preference levels.

eventually "catches up". For large problems (all of Figure 5.13), GAPD has superior anytime performance throughout the time tested. We expect that, given sufficient time, Ario will surpass the solution quality of GAPD on any problem size. The best algorithm for the application depends on the anytime requirements and the problem size. For large problems with stringent anytime requirements, GAPD performs best.

5.5 Future Work

An interesting avenue of future work is to explore the differences between Ario and GAPD, which approach the problem in very different ways. GAPD, with its anytime focus, employs a divide and conquer approach, while Ario essentially uses branch

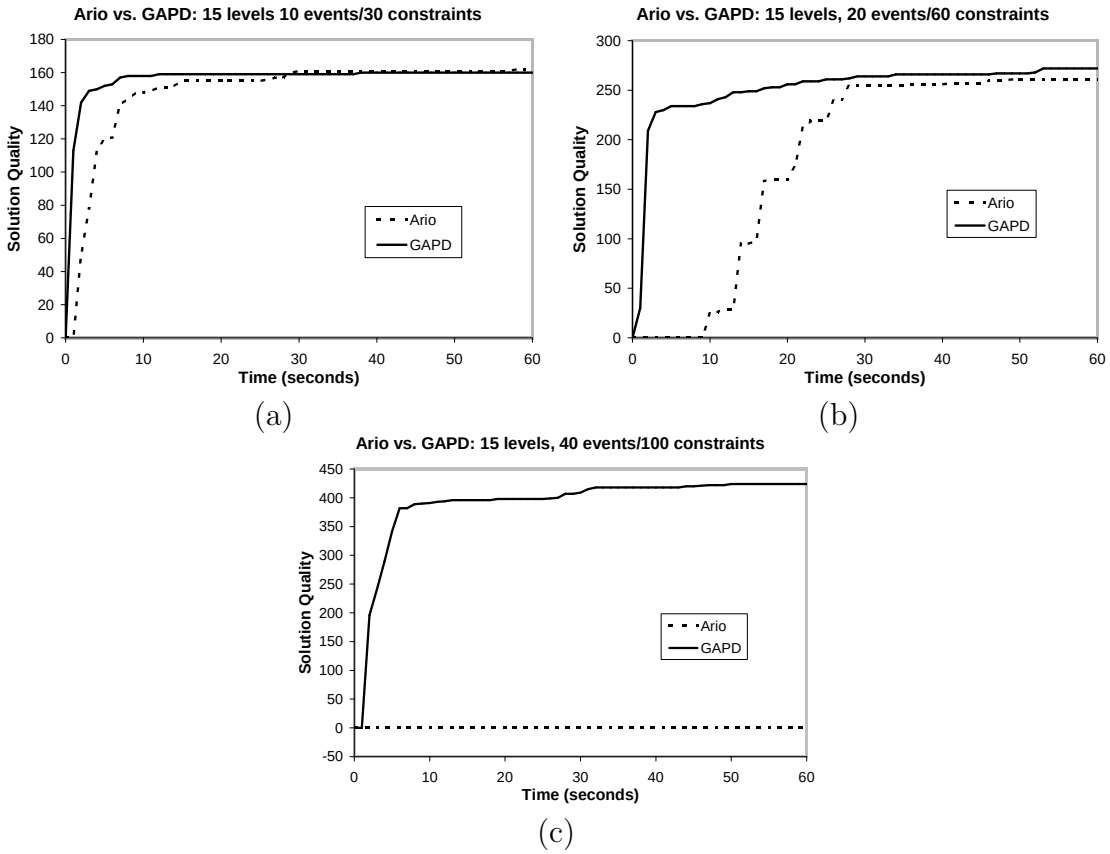


Figure 5.13: Anytime comparison of GAPD and Ario for problems with 15 preference levels.

and bound with very powerful heuristics. Some of Ario's heuristics are inherited from the long history of SAT research, while others were developed specifically for this problem. It would be interesting to see if the strong elements of one can be incorporated into the other. For example, when Ario finds a solution, can it use STPP_Greedy-style heuristics to improve the solution? Or, can GAPD incorporate conflict information uncovered by Ario to choose or eliminate cSTPPs?

The model of GAPS performance used by GAPD is static, elicited from empirical GAPS data over a wide set of STPPs. While informal experiments showed that GAPD's performance was not very sensitive to variations in the α and β parameters, learning the parameters while solving each problem may make a small difference.

CHAPTER VI

DTPPs in Autominder

In this chapter, we present results that show the efficacy of `DTPP_Maximin` (Chapter IV) and `GAPD` (Chapter V) when integrated into the Autominder system. We first describe Autominder's main components and its use of DTPs in managing the daily plans of cognitively-impaired individuals. We then discuss the issues involved in integrating the DTPP algorithms into Autominder. The chief issue is the additional requirement made on the DTPP solver by typical Autominder operations: the solver must quickly revise its solution in response to small changes in the DTPP. To meet this requirement, we modify the `DTPP_Maximin` and `GAPD` algorithms to handle *dynamic DTPPs* (Section 6.2). In Section 6.5, we show experimentally that both algorithms can be practically used in Autominder and that the dynamic versions of each algorithm perform better than the non-dynamic versions.

6.1 Autominder

Autominder [64] is a cognitive orthotic[1] for people with mild memory impairment, such as elderly people or those who have suffered a traumatic brain injury. People with mild memory impairment are often highly functional, but have problems plan-

[1] An orthotic is a device that augments some existing function in a person. Autominder augments the memory and planning functions of its user.

ning their day or remembering to perform important activities of daily living (ADLs). Thus, institutionalization is often necessary to ensure the safety of the individual. The goal of Autominder is to coexist in the person's living environment and provide reminders for important activities, hopefully enabling the person to continue living independently. By enabling independent living, Autominder, or a system like it, can improve quality of life while lowering health care costs.

Autominder manages the daily plan of its user by reasoning about the best times to perform important activities, by monitoring the progress of the user in executing those activities, and by issuing reminders to the user when appropriate. These three functions correspond to the three main components of Autominder:

Plan Manager The plan manager performs all functions related to forming and updating the user's plan. The plan is represented as a set of steps to execute, a set of conditions that must hold before executing a step, a set of conditions that hold after each step is executed, and a set of DTP(P)-style constraints that restrict when each step can occur relative to each other. The Plan Manager encodes all of these elements as a DTP(P) and uses a DTP(P) solver to calculate a flexible, legal schedule for each step in the plan.

Client Modeler Autominder is meant to function in environments that can sense the movements and actions of its user, using either fixed, distributed sensors or mobile, robotic-based sensors. In either case, the client modeler uses a stream of sensor data and the current state of the user's plan to infer which activities have been performed or not performed. The results of this inference is used to update the user plan and to guide the third component of Autominder, the Intelligent Reminder Generator.

Intelligent Reminder Generator The Intelligent Reminder Generator uses the plan, information from the Client Modeler, and a record of past user performance to reason about when or if to issue a reminder for each step in the plan. It must issue reminders in a way that does not annoy the user, encourage excess dependence on the system, or cause the user to violate the constraints in the plan.

The three components are tied together using a module that passes messages between the components and allocates processing time to each.

Of course, this thesis is concerned almost wholly with the Plan Manager component and extending its ability to reason about temporal constraints. Previously, the Plan Manger could handle only DTP constraints, and used the Epilitis solver [79] to find solutions to the DTPs. The goal of this work was to replace Epilitis with a solver that can reason about preferences without destroying the efficiency gains enabled by previous work, which developed heuristics and pruning strategies for solving DTPs. Both the `GAPD` and `DTPP_Maximin` algorithms retain the efficiency techniques used by Epilitis. In fact, the basic algorithm `Solve-DTP` mentioned often in previous chapters is a skeleton of the Epilitis algorithm. `DTPP_Maximin` builds directly on Epilitis, using it for solving the level DTPs (see Section 4.4). `GAPD` uses Epilitis to find the component STPPs that it later searches with `GAPS` (see Section 5.3). Therefore, for both algorithms, the first solution is found in the same amount of time Epilitis would require to find its solution. Once the first solution is found, the solution quality is quickly improved. Thus, preferences are added with no practical loss or trade-off.

6.1.1 Autominder operation

There are two phases in the operation of Autominder: the setup phase and the active phase. In the setup phase, a caregiver inputs a set of plan fragments that encode the set of activities the user is to perform. After each fragment is entered, Autominder reports whether the plan fragment "fits" into the plan, i.e., whether there is a way to execute the plan satisfying all constraints. The determination of whether a new fragment fits is made by encoding the combined plan as a DTP (or DTPP) and using the solver to determine whether it is consistent. We call the process of entering a plan fragment a *Merge*. The DTP(P) solver must determine the consistency of a plan in time that is acceptable to the caregiver. Thus, for the DTPP algorithms, finding the optimal solution after the insertion of each plan fragment is not always practical.

Once a set of plan fragments are entered, the active phase begins[2]. The Client Modeler begins processing sensor information and the Intelligent Reminder Generator begins determining when to issue reminders. As time passes, one of a few events cause the Plan Manager to "update" its plan. First, if the Client Modeler reports that a step has started or ended, the Plan Manager must update the constraints that involve the modified step. For example, if the current plan allows the start of an Exercise step to begin anytime between 10:20 and 10:50, and the Client Modeler reports that the user began exercising at 10:20, the plan must be updated to encode the fact that Exercise begins exactly at 10:20. Reducing the range from [10:20, 10:50] to [10:20, 10:20] may affect the allowable times for many other steps, once propagated.

[2]In Autominder, there is no requirement that the setup phase complete before the active phase begins. In fact, a new plan fragment can be entered at any time. We make the separation because the separation often occurs in practice and because our tests in Section 6.5 use it.

A second trigger for a plan update is the passage of an important time. In the previous example, if the Plan Manager learns that the current time is 10:51, and it has received no report from the Client Model indicating the Exercise step has started, then one or more plan constraints have been violated: the DTP(P) solution (a component STP) is no longer consistent. The DTP must be searched again for another solution. If no solution exists, one or more constraints must be relaxed to recover from the failure. At least four important times exist for each step of the plan, because each step has a start event and an end event, and each event has a range of allowable times with a start and end point.

We call each of these bound-changing updates a *Tightening*, because each change is tightening a constraint. It is not uncommon for Autominder to tighten a DTP at the rate of once per minute for short periods of time. Therefore, the DTP solver must be able to process the tightening in less than a minute. In fact, much less than a minute is available because the other two components of Autominder need computation time as well. Therefore, we adopt the requirement that our DTPP solver cannot take more than 15 seconds for each update.

6.2 Dynamic DTPPs

That both `DTPP_Maximin` and `GAPD` are designed for anytime performance bodes well, given that only about 15 seconds is available for solving the DTPPs after each update. For problems in which 15 seconds is not enough time to find an optimal solution (or for applications in which less time is available), we can take advantage of the fact that an update often modifies only a small portion of the DTPP—we can use the results of finding the solution to one DTPP to help find a better solution after the DTPP is modified. In many cases, the original solution is still valid, so at the

very least, the original solution should be checked before starting over completely. In this section, we explore the idea of solving *dynamic DTPPs* by augmenting the `DTPP_Maximin` and `GAPD` algorithms.

The need for solving dynamic constraint satisfaction problems is not specific to Autominder. In fact, the notion of dynamic CSPs has been explored in several other contexts [22, 48, 26, 73, 85, 75]. Our definition of dynamic DTPPs is most similar to that in [75], which defines a dynamic CSP using an initial static CSP and a sequence of changes to that CSP.

Definition 18 (Dynamic DTPP)**.** *A* **dynamic DTPP** *is defined as* $\langle T, \Delta \rangle$*, where* T *is a DTPP and* Δ *is a sequence of changes to* T*. Each change in* Δ *is defined as* $\delta = \langle \mathcal{N}, \mathcal{R}, \mathcal{M}, \mathcal{X} \rangle$*, where* $\mathcal{N}$ *is a set of new constraints,* $\mathcal{R}$ *is a set of removed constraints,* $\mathcal{M}$ *is a set of tightened constraints, and* $\mathcal{X}$ *is a set of relaxed constraints.*

Thus, the definition defines a sequence of $|\Delta| + 1$ DTPPs, where the i^{th} DTPP is formed by sequentially applying the first i changes in $|\Delta|$ to T. When talking about dynamic DTPP algorithms, we usually refer to the individual DTPP relative to an individual change, called the *current change*. The DTPP that existed before the current change is the *previous* DTPP and the one that results after applying the change is the *modified DTPP*.

A successful dynamic DTPP algorithm is one in which solving the sequence of DTPPs requires less time that individually solving each DTPP in the sequence. A solution to a dynamic DTPP is a sequence of DTPP solutions—one for the initial DTPP and one for the DTPP produced after each change is applied.

In general, it is difficult to make claims about algorithms that solve dynamic DTPPs (or other dynamic CSPs) unless restrictions are made on the size or type of each change in the sequence. For example, if we allow changes of any size and any

type, the dynamic DTPP could include a change that removes all current constraints and adds an equal number of new constraints. In such a case, we cannot expect a dynamic DTPP solver to do any better than a static one.

However, if we restrict the number of changes to one, or restrict the type of change to only allow additions or tightenings, then techniques can be defined that will retain and leverage information found solving the previous DTPPs. In Autominder, we often change several constraints at once, but the change is usually of a single type. During the setup phase, when plan fragments are being added, the changes typically contain only new constraints. During the active phase, the modifications are typically only tightenings of one or more constraints. Thus, we focus on these two cases in what follows. We show how to modify the `DTPP_Maximin` and `GAPD` algorithms to efficiently handle constraint additions and tightenings.

6.3 Dynamic DTPP_Maximin

For both constraint additions and constraint tightenings, the DTPP becomes *more constrained* after the change. Thus, when solving a modified DTPP after a change, all inconsistent assignments found for the previous DTPP remain inconsistent. Moreover, if the optimal solution was found in the previous DTPP before the current change is enacted, the maximin optimal value of the previous DTPP serves as an upper bound for the maximin optimal value of the modified DTPP[3]. This follows from the simple fact that constraining the problem more will not add to the set of possible solutions; rather, it can only reduce the set of possible solutions. Therefore, either the old solution is still valid, another solution exists at the optimal level, or

[3]This holds only for idempotent aggregation functions such as *min*. Since *sum* is not idempotent, the same is not true for utilitarian-optimal solutions: adding a constraint can increase the total value.

the optimal solution value will be reduced.

6.3.1 Adding new constraints

In work outside this thesis, we addressed the problem of solving over-constrained DTPPs, i.e., DTPPs that contain too many events and associated constraints in a given time-window (this corresponds to an Autominder scenario in which a set of plan fragments are entered all at once, letting Autominder decide which plans to keep or discard) [60]. In our solution to this problem, we introduced pruning techniques that allowed the `DTPP_Maximin` solver to quickly find the optimal solution to a DTPP after the addition of a set of constraints. We describe one of those techniques here, which we incorporated into our `dynamic DTPP_Maximin` algorithm without modification[4].

Since each modified DTPP in the sequence is tighter than previous ones, there exists a property that is analogous to the upward inconsistency property: the *persistent inconsistency* property. This property says that any component STP found to be inconsistent will remain inconsistent throughout all later changes. Therefore, when searching a level DTP of a modified DTPP, we can skip (partial) component STPs that have already been checked.

Figure 6.1 illustrates this property by showing the search space of a level DTP before and after a constraint is added. The level DTP starts with only 3 constraints, each with 2 disjuncts. The root of the search tree corresponds to a partial component STP with 0 constraints added. Moving down the left side of the tree represents adding the first disjunct of each constraint to the partial component STP. When the partial STP becomes inconsistent (gray nodes), backtracking occurs, effectively pruning all nodes that descend from the node that caused the inconsistency (hashed nodes). When a full consistent component STP is found (black node), the search exits.

[4]The ideas in this section were developed with Peter Schwartz and Michael Moffitt.

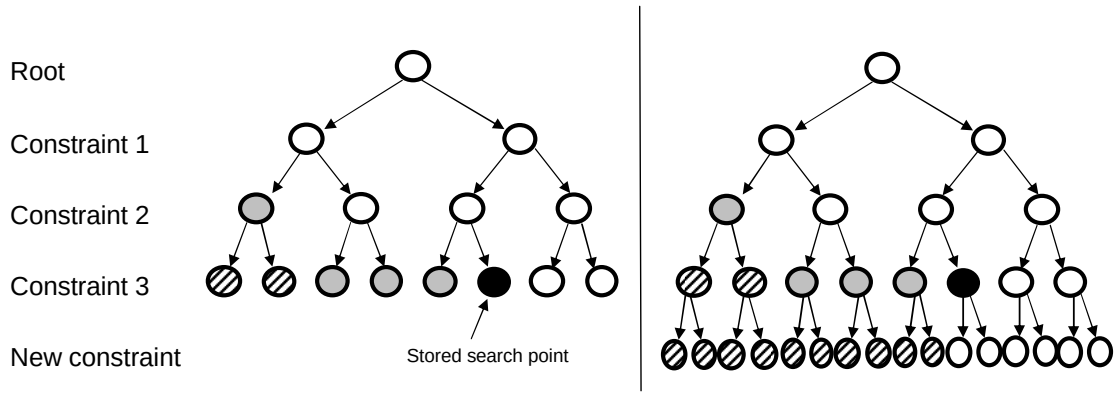


Figure 6.1: The search space of a level DTP before and after a constraint addition. The level DTP starts with 3 constraints, each with 2 disjuncts. The tree on the left represents the search space before the constraint is added: gray nodes represent partial component STPs that have been found inconsistent; hashed nodes represent unreached partial component STPs that are known to be inconsistent; and the black node represents the first consistent component STP found. The right tree shows the new search space after the new constraint has been added.

When a constraint is added to the level DTP, we get the search tree on the right side of Figure 6.1, assuming that we simply append the new constraint onto the bottom of the search tree. In this case, the search trajectory remains unchanged up to the point of the original solution (the black node). Notice that all nodes added on the bottom layer are pruned if they descend from previously visited or pruned nodes. The persistent inconsistency property ensures this.

Thus, if we store the search point for the original solution of a level DTP, we can start from the same point when searching the modified level DTP. The only requirement is that the original constraints are left in the same order and the new constraints are tacked onto the end. If we store the search points that correspond to the last solution found in every level DTP of a DTPP, we can considerably improve the worst-case performance of `DTPP_Maximin` with respect to constraint additions in dynamic DTPPs.

Therefore, the `dynamic DTPP_Maximin` algorithm for handling constraint additions proceeds as follows:

1. Start searching the level 0 DTP at the stored search point for level 0
2. When (if) a solution is found, move to the next level, starting at the stored search point.
3. Stop after no solution is found on a given level or after a solution is found on the upper bound level.

The upper bound level is the level of the optimal solution for the previous DTPP, if it was found. Otherwise, in the event that the optimal solution was not found before the cut-off time, the upper bound is the level of the highest level DTP.

The result of the pruning achieved by the upward inconsistency and persistent inconsistency properties is that no inconsistent component STP is ever checked twice. The only component STPs that are checked twice are those found to be consistent during the processing of any change. The number of consistent component STPs per change is bounded by the number of preference levels ($|A|$), since the algorithm moves up one level each time a consistent component STP is found [61]. Therefore, if all changes in the sequence are constraint additions, we can bound the complexity of the total search for all calls to `DTPP_Maximin` to be the sum of (1) the complexity of searching all level DTPs of the largest DTPP tried and (2) $|A| \cdot |\Delta|$ extra STP consistency checks. This analysis is good news in terms of worst case complexity: finding the optimal solution for a sequence of plan fragment additions to Autominder barely exceeds the worst case of finding the optimal solution to the DTPP that combines all plan fragments. This theoretical analysis is of little use in the Autominder system, where the time limit is imposed on individual changes, not the dynamic DTPP as a

whole. Other systems may be able to take advantage of it.

We do not expect the average case complexity for the two situations to be quite so close. The "skip" mechanism that enables the appealing analysis constrains the variable ordering, working against the variable ordering heuristics that contribute substantially to the efficiency of the `DTPP_Maximin` algorithm. In terms of practical efficiency, the biggest savings come from using the upper bound obtained from the previous DTPP. As we learned in Chapter IV, the most difficult level DTPs are at the solution level and the level above it. If the modified DTPP has the same optimal level as the previous DTPP, then search can stop as soon as the optimal solution is found. Without the upper bound, the algorithm would have to spend time searching the higher level to prove the optimal solution was indeed an optimal one.

6.3.2 Tightening current constraints

The persistent inconsistency property also holds when tightening constraints. Therefore, the practice of storing the past search points can be useful in handling tightened constraints as well. When searching the modified DTPP for a new optimal solution, we again start at the lowest level and attempt to "fast-forward" to the stored search point, i.e., the state of the search when the consistent component STP was found in the original DTPP. Unlike the previous case, in which we had a guarantee that the stored point would represent a consistent component STP, the search point in this case may represent an inconsistent component STP, because some of its constraints may have been tightened. In fact, the stored search point may not even be reachable in the search, since the inconsistency may have been caused before the final constraint is reached.

Therefore, when we fast-forward to the stored search-point, we must build the

component STP one constraint at a time. When an inconsistency occurs, search resumes at the point of inconsistency in a normal `DTPP_Maximin` fashion. This modification is both simple to implement and simple to analyze. We can easily show that no inconsistent component STP is ever checked twice in this situation. If all changes in a sequence Δ are tightenings, each tightening will cause at most $|A|$ consistent component STPs to be rechecked. Therefore, the worst-case complexity of the dynamic DTPP only exceeds the worst-case for the original DTPP by $|A| \cdot |\Delta|$ STP consistency checks.

Since the method for handling constraint additions and constraint tightenings use the same mechanism, they can be easily combined to handle both simultaneously. This is a small advantage for Autominder, since the setup and active phases are sometimes interleaved: a plan fragment may be merged at the same time other constraints are tightened.

6.4 Dynamic GAPD

Compared to `DTPP_Maximin`, the size of the state managed by the `GAPD` algorithm is enormous. The state of the algorithm includes the state of the `Solve-DTP` algorithm, all information in the STPP_Tree, and all information contained in each partially solved `GAPS` instance. Therefore, in making the algorithm dynamic, we must be mindful of the risk that updating the state in response to a constraint change or addition may require significant time, possibly absorbing much of the potential savings earned by making the algorithm dynamic.

We illustrate the state maintained by `GAPD` in Figure 6.2.

When adding or tightening constraints in the DTPP, each element of Figure 6.2 must be updated, including the internal nodes of `Solve-DTP`, the internal nodes of

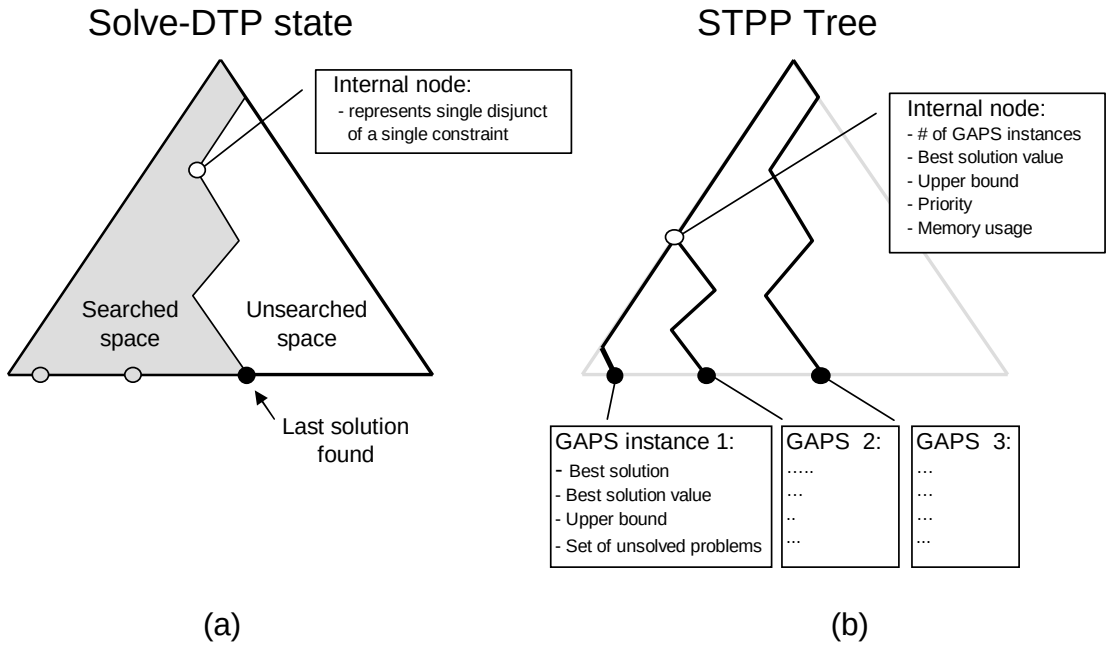


Figure 6.2: The state maintained by `GAPD`. (a) shows the state of `Solve-DTP` after having found three solutions. (b) shows the STPP_Tree containing those solutions, as well as the `GAPS` instances that correspond to the DTP solutions.

the STPP_Tree, and the state within the `GAPS` instances.

6.4.1 Adding new constraints

Adding a new constraint to `GAPD` is difficult because adding a constraint to `GAPS` is difficult. `GAPS` organizes its search by creating and pruning STPP subproblems. If a new constraint is added, it must be added to all subproblems, including those that have been pruned. Because `GAPS` does not maintain a record of pruned instances, and because the pruned instances may contain the new best value, the search must be restarted each time a new constraint is added. In addition, all upper bounds and best values are invalidated because the sum function is not idempotent[5]. In short,

[5]For example, assume that a problem has 2 constraints, 5 preference levels, a best solution of value 6, and an upper bound of 9. If we add a third constraint with five levels, the new optimal solution could be anywhere from 0 to 14; subproblems with an upper bound of 5 would have been pruned in the original problem (5<6), but may now participate in the optimal solution.

GAPS is not amenable to constraint additions. Therefore, any strategy for efficiently handling constraint additions should recreate all GAPS instances.

Savings are possible for the part of the algorithm that searches for cSTPPs. Just as we did with DTPP_Maximin, we can add the new constraints to the end of the constraint list and continue search from each leaf in the STPP_Tree. The time spent searching for the leaves of the original STPP_Tree could be saved.

This is not likely to make much of a difference, however, given our use of random restarts and the lesson we learned in Section 4.7.3 about the effects of a good variable ordering. Given these difficulties, we do not attempt to achieve any savings in this area. Each time new constraints are added, we will restart the GAPD algorithm.

6.4.2 Tightening constraints

When constraints are only tightened between two DTPPs in a sequence, often much of what was learned in solving the previous DTPs can be used in solving the next. Incorporating a tightened constraint into the copious amounts of state maintained by GAPD is not conceptually difficult, but it requires substantial changes to the GAPD algorithm. Since GAPD maintains a set of partially solved GAPS instances, we start by discussing how to make GAPS dynamic with respect to constraint tightenings.

Dynamic GAPS

GAPS solves an STPP by running STPP_Greedy on the problem and then partitioning the STPP based on the result of STPP_Greedy. During search, GAPS maintains the following four pieces of state: (1) a set of STP constraints that represent the best solution found so far, (2) the value of that best solution, (3) an upper bound for the optimal solution, and (4) a set of unsolved subproblems. Throughout its processing, many subproblems are partially or fully solved and many others are pruned before

they are solved. As a result, GAPS throws away the results of most subproblems, keeping only the best solution found and enough information to describe the remaining problems to be solved.

After a constraint is tightened, one of two possibilities can occur: either the current best solution remains consistent or the tightened constraints render it inconsistent. For the first case, the entire GAPS state can be preserved. Proving the former best solution is still valid proves that the first two elements of state remain valid. The third element, the upper bound, remains valid because tightening constraints can never lead to an increase in the value of the optimal solution. The fourth element, the set of unsolved subproblems, is partially invalidated since some subproblems would have been pruned if the tightened constraints were part of the original problem. However, such problems will be quickly pruned after a single iteration of GAPS[6].

For the second case, in which the former best solution becomes inconsistent, very little can be saved. Obviously, the best solution must be discarded and the best value is invalidated. The upper bound still holds, but the set of subproblems no longer represents the space left to be searched. Given that the best value is no longer valid, the optimal solution may be less than the former best, which means that the optimal solution may have been pruned in the original search. Therefore, the entire GAPS search should be restarted, saving only the upper bound.

Thus, a dynamic version GAPS is useful, but not very interesting—the search is either left unchanged or restarted, depending on the result of a simple check of the former best solution.

[6]This is only true for the GAPS algorithm as presented. In the implementation detailed in Appendix A, such problems are pruned without doing the GAPS iteration.

Updating the remaining state

Recall that GAPD maintains three pieces of state during its search: (1) the state of Solve-DTP, (2) the nodes of the STPP Tree, and (3) the GAPS instances stored at each leaf of the STPP tree. We have just discussed how to update the GAPS instances, so we now turn to the other two pieces.

To update the Solve-DTP state in response to tightened constraints, we can use the same principles discussed in Section 6.3.2: remembering the search point of the last found solution is all that needs to be done. Previously pruned component STPs are guaranteed to remain inconsistent, so search can just resume at the last search point. Of course, all constraints of the level 0 DTP it searches must be tightened appropriately. This creates challenges in the implementation, but is of no conceptual interest.

The third element of state, the STPP_tree, serves three functions: a priority queue for the GAPS instances, a mechanism for quickly finding the highest-valued solution, and a mechanism for tracking memory usage. To achieve these ends, the three values *priority*, *best*, and *memory* are maintained for each node of the tree. When constraints are tightened, the three values need to be updated to reflect the differing states of the GAPS instances at the leaves of the tree.

Algorithm for updating all state

Now that we have presented exactly what state must be updated after constraints are tightened and before restarting GAPD, we can describe a simple three-step algorithm for making the update.

1. **Update** Solve-DTP After determining exactly which disjuncts were tightened in each constraint, the preference projection for the level 0 DTP is updated in the

`Solve-DTP` state. This corresponds to tightening the disjuncts in the internal nodes of Figure 6.2(a).

2. **Find affected `GAPS` Instances** Then, we perform a pre-order traversal of the STPP_Tree depicted in Figure 6.2(b), keeping track of whether the current sub-path contains a tightened disjunct. Once a leaf is reached for a sub-path that contains a tightened disjunct, the tightened disjuncts are propagated to the `GAPS` instance for that leaf. The *ROOT* component STP of the instance is checked for consistency (i.e., the original solution found by `Solve-DTP`). If inconsistent, the `GAPS` instance is pruned. Otherwise,the best solution of the `GAPS` instance is tested for consistency: if consistent, no change is made; if inconsistent, the `GAPS` search tree is destroyed, leaving it ready to execute the first greedy iteration. The new *best* and *memory* values are propagated back through the STPP_Tree and the traversal continues.

3. **Recalculate Priorities** After the first traversal, the *best* value for the entire tree may have been lowered, since the `GAPS` instance that produced the original best may have been reset. If the best value is lowered, a second traversal is needed to recalculate all priority values.

The problem with this simple algorithm is that checking the `GAPS` instances for consistency in step 2 can be expensive—almost as expensive as a single `GAPS` iteration. Using the standard Floyd-Warshall-like algorithms for checking consistency requires time that is cubic in the number of events. When multiplied by the possibly thousands of `GAPS` instances after each update, the required time may absorb a significant amount of the savings promised by making `GAPD` dynamic.

To address this problem, we leverage the exceptional average case performance of the AC-3cc algorithm (described in Section 2.3.1) to check consistency of the GAPS instances *as we traverse the STPP tree*. Each node in the STPP tree is associated with a single disjunct of a single DTPPP constraint. When entering a node, we propagate the node into the AC-3cc algorithm. When backtracking through a node, we *unpropagate* the constraint using the procedure defined in [12][7]. If propagation fails at any node in the STPP tree, all instances descending from that node are pruned. If a leaf node is reached, and the propagation succeeds at that node, then the GAPS instance is consistent.

The state of the AC-3cc solver can be incorporated into the GAPS instance, since it also uses AC-3cc during its search. For instance, when checking the consistency of the best solution for a GAPS instance, the newly computed AC-3cc state can serve to reduce the necessary work: of all constraints composing the best solution, only those at levels 1 and greater need to be propagated.

Once all state has been updated, the GAPD search can be continued in its standard way. Assuming that the time required for the update is not significant, the algorithm should be better positioned to find good solutions faster than the case in which the algorithm had been completely restarted. However, there are two factors that admit the possibility that the restarted algorithm could find the optimal solution faster than the dynamic algorithm. First, the tightened constraints could lead to new variable orderings in the Solve-DTP portion of GAPD, which could result in finding the component STPPs faster. Second, since the heuristics of STPP_Greedy depend on constraint tightness, the GAPS instances in the restarted version may find the optimal solutions faster. In the experiments below, the dynamic version performed as well

[7]The unpropagate function only works when the last propagation yielded a consistent solution. For other cases, the search must be restarted.

or better than the restarted version on average.

6.5 Experimental Evaluation

We set out to answer two questions in these tests: (1) whether `DTPP_Maximin` and `GAPD` can be practically applied to a real-world application and (2) whether the dynamic elements of `GAPD` and `DTPP_Maximin` achieve any real savings. To answer both questions, we integrated the algorithms into the Autominder system, and measured the performance of each algorithm both with and without the dynamic elements enabled.

We measure performance in the Autominder system using two metrics: the time required for each algorithm to finish after each update, and the value of the objective function (the aggregated preference value) after the solver exits. Usually, only one of these metrics is interesting at any given time: for cases in which the solver can find the optimal solution in less than the alloted 15 seconds, only the time to finish is interesting, since the objective value will be the same with and without the dynamic elements. In cases where finding the optimal solution is difficult, both versions of the solvers will often use the full 15 seconds, so the objective value is the more interesting metric.

For the plans we feed into Autominder for this test, `DTPP_Maximin` almost always finds the optimal solution is less than 15 seconds, so we evaluate the performance of its dynamic version using only time-to-finish. `GAPD` found the optimal solution around half of the time, so we evaluate its dynamic version using both the objective value and running time.

6.5.1 Test plans

Ideally, we would test these algorithms using real plans used in Autominder test trials with real people. Unfortunately, the plans used in Autominder clinical trials were both extremely simple and small in number. Practical constraints outside of Autominder required us to restrict the complexity of plans and to linearize them, effectively taking the DTP manager out of the test.

Therefore, we analyzed the plans created by caregivers and medical professionals in the Autominder tests, and identified three plan templates that enveloped most of the given plans.

PrepDo The simplest plans consisted of a single core activity, preceded by some preparation action required to prepare for the activity. A typical example is the activity "Make Lunch", which must be preceded by the action "Go to the kitchen". The core activity has a range of allowable times with an ideal time somewhere in that range. The preceding action had a smaller range of times and was typically preferred to be as close to the core activity as possible.

Repeat Often, a single activity must be repeated throughout the day at semi-regular intervals. For example, an elderly person may need to be reminded to use the toilet once every 2-3 hours.

Process More complicated plans can be represented as a process in which all preparation activities need to be executed before some final activity. The preparation activities are partially ordered, and have simple temporal constraints that enforce that ordering. Often, the interesting temporal constraint is the one that requires the entire process to end by a certain time. An example is the plan "Do Laundry", which requires each load to be washed and dried before a final

"Put clothes away" activity.

Each type of plan fragment contains a special step called *dayStart*, which corresponds to the "Wakeup" step that exists in most Autominder sub-plans. When merging two plans containing a *dayStart* step, the resulting plan will contain only one copy of *dayStart*.

The three templates can easily be parameterized to allow instances of each template to be randomly generated. In general, different instances vary in the number of activities (Repeat and Process templates), the range of allowable start and end times of each activity, the duration of each activity, the number of temporal constraints between activities, and the shape of the preference functions on each temporal constraint.

We define preference functions to be a two-segment, piece-wise linear function, which can be parameterized using three values: center point (C), slope 1 (S1) and slope 2 (S2). Figure 6.3 shows the types of functions that, when combined with a limit on preference levels, can be represented by the three parameters[8].

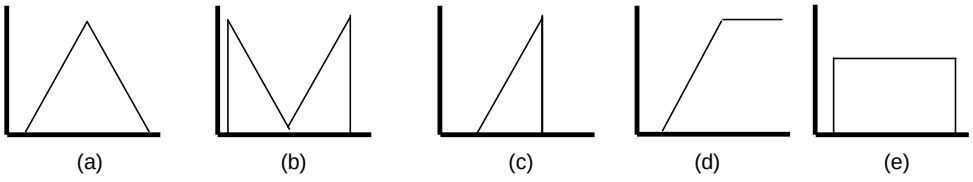


Figure 6.3: The types of functions definable using the parameters C, S1, and S2.

Hereafter, we will call a set of three parameters that define a preference function a *PF-tuple*.

In Figure 6.4, we list the parameters that, once specified, will create an instance

[8]Functions (b) and (e) can be represented using the 3 parameters because the STP constraint to which it is attached has upper and lower bounds. Therefore, there are effectively 5 parameters defining the functions.

PrepDo:

latestFinish	Upper bound for the end time of the Do step
dayStart	PF-tuple for the starting time of the "Day start" step
dayStartDur	PF-tuple for the duration of the "Day start" step
prepDo	PF-tuple for the separation between Prep and Do
prepDur	PF-tuple for the duration of the Preparation step
do	1-2 PF-tuples for the starting time of the Do step
doDur	PF-tuple for the duration of the Do step

Repeat:

makeSpan	Upper bound for the difference between first and last events
dayStart	PF-tuple for the starting time of the "Day start" step
dayStartDur	PF-tuple for the duration of the "Day start" step
numActions	Number of times to repeat the action
actionDur	PF-tuple for the duration of each action
actionSep	PF-tuple for the separation between each action

Process:

latestFinish	Upper bound for the end time of the final step in the process
makeSpan	Upper bound for the difference between the first and last events
dayStart	PF-tuple for the starting time of the "Day start" step
dayStartDur	PF-tuple for the duration of the "Day start" step
numActions	Number of times to repeat the action
actionDur	PF-tuple for the duration of each action
actionSep	PF-tuple for the separation between each action

Figure 6.4: Parameters for the three Autominder plan templates.

of each plan template. Using the templates and parameters in Figure 6.4, we can randomly generate a set of plans with typical Autominder plan structure. The structure of the DTPPs that encode the plans differ from the randomly generated DTPPs of the previous chapters in two main ways. First, a large portion of the constraints are STPP constraints rather then disjunctive constraints. Second, a majority of the disjunctive constraints are without preference functions—*non-overlapping* functions used to temporally separate steps.

As a result, the two DTPP algorithms can handle problems with much larger numbers of constraints. The `Solve-DTP` algorithm used by both DTPP algorithms is essentially unaffected by STPP constraints: it simply moves them to the front of the

constraint ordering and processes them all once. Similarly, the GAPS algorithm used by GAPD is essentially unaffected by the addition of constraints without preferences. Since the constraints without a preference function must exist in every component STP, they can be propagated into the STP solver at the very beginning and ignored in later iterations.

6.5.2 Experimental setup

Autominder is instrumented with two different modules that simulate sensor input, allowing Autominder components to be tested without situating it in a real environment. The first module simulates the movements of an ideal user who follows Autominder's every instruction. With this simulation module, Autominder will generate an initial plan and retain that plan until the plan is fully executed, since the "user" will always take actions that are consistent with the current plan.

The second, more interesting, simulation module mimics the behavior of a non-compliant user. With probability p, the user will execute an action t minutes before or after the allowable time, where t is chosen randomly. When a step is started or finished outside of the allowable bounds, it violates the current component STP, and the constraint solver must look for a new one that is consistent with the user's behavior so far. If one does not exist, it fails, and recovers only by relaxing one or more constraints in the problem. As time progresses, the likelihood of finding a new component STP reduces, since past choices limit the number of applicable component STPs.

Using the non-compliant simulation module, we set Autominder to accept a set of plans at the start of its run, automatically merge each plan into the daily plan (some plans were rejected), and automatically advance time until the end of the plan

is reached. Time automatically advanced one minute if one of two conditions were met: a real-world minute passes; or all Autominder modules have finished processing.

During each run, we logged information related to the DTPP solvers: the type of update, the length of time required to make the update, the objective value of the solution before and after the update, whether the solution is the optimal, and the number of constraints or events that were modified, added, or removed since the last update. By comparing the log files of each problem with and without the dynamic elements enabled, we were able to quantify the effect of the dynamic elements and judge whether the algorithms can be practically used in Autominder.

We generated problems of three different sizes: 4 plan fragments, 8 plan fragments, and 12 plan fragments, where each fragment contained between 4 and 12 events. The 12 plan size resulted in 150-300 constraints, whereas the 4 plan size produced DTPPs with 20-60 constraints. For each problem size, we chose a different makespan for the total plan, which allowed us to make sure each problem was "interesting"; meaning, they are not extremely over- or under-constrained[9]. Problems with very large makespans are uninteresting because solutions usually have the maximum objective value possible, i.e., every constraint is satisfied to the maximum value. Problems with very small makespans are uninteresting because they are either over-constrained and hit failure often, or they result in Autominder rejecting many of the plan fragments.

For each problem size, we produced 20 instances and simulated a full execution of the plans in each instance in Autominder using the non-compliant simulation module. To isolate the performance gain attained by the dynamic elements of the DTPP solvers, we instrumented Autominder to solve each DTPP in the sequence

[9]The makespan of a plan is the time distance from the start of the first action to the end of the last action.

twice: once using the dynamic elements and once with the elements turned off. When dynamic elements are turned off, the DTPP solver simply treats the modified DTPP as a new problem.

An alternative setup would be to do two simulation runs, once for each version of the solvers. However, the solution chosen by the solver has a strong impact on the rest of the simulation. Since the two simulations would diverge quite early in the run, a comparison of the two would not be very informative. In other words, the dynamic DTPP to be solved is not defined at the beginning of the Autominder run. During the active phase, each solution to modified DTPPs influences future changes in the dynamic DTPP.

We set the makespan for the 4, 8, and 12 plan instances to 700, 1000, and 1300 minutes, respectively, which kept the instances in the "interesting range". Approximately 75 percent of the plans successfully merged into Autominder's plan for each instance.

6.5.3 Experimental results

Dynamic DTPP_Maximin

Table 6.1 shows the results for the test using the `DTPP_Maximin` algorithm, both with and without the dynamic techniques enabled. We report the average running times for each Merge operation and each Tighten operation. A Merge corresponds to a new plan fragment being added to Autominder's current plan, and therefore the addition of new constraints and events to the DTPP. A Tighten operation corresponds to an update that tightens the bounds of one or more constraints in the plan.

We do not report the preference levels in Figure 6.1 because both versions of the algorithm produced solutions with the same objective value. The algorithms found

	4 plans		8 plans		12 plans	
	Off	On	Off	On	Off	On
Avg Merge time	.003	.003	.045	.034	0.812	0.449
Min Merge time	.000	.000	.001	.000	0.002	0.000
Max Merge time	.009	.006	.134	.109	4.647	2.888
Avg Tighten time	.009	.000	.150	.011	1.359	0.068
Min Tighten time	.001	.000	.115	.003	0.879	0.030
Max Tighten time	.040	.012	.294	.096	5.385	2.753

Table 6.1: Running times (ms) for `DTPP_Maximin` tests. "Off" and "On" denote whether the dynamic elements of the algorithm were enabled.

the optimal solution in nearly all instances. The sole exception was a merge operation for a 12 plan instance: the non-dynamic version found the optimal solution, but did not prove it was optimal.

Notice that for the Merge operations, the additional pruning significantly decreased the average running time for the solver. Although we had hoped for this result, it was not certain given the concept described in Section 4.7.3, in which running time increases despite a reduced search space. Again, the significant power of the variable ordering heuristics is inhibited by the requirement that the original constraints must stay in the same order. This phenomenon did not have a significant impact in the Autominder tests. One possible explanation is that each addition to the DTPP consisted of a relatively self-contained set of constraints that interacted with the existing constraints only through non-overlapping constraints.

The pruning techniques for the tightening operation achieved significant savings as well. For the 12 plan fragment case, the averaging tightening time was over 95% less using pruning. Each problem instance for the 8 plan fragment case required an average of 107 tightenings, as opposed to an average of 10 merges[10], so it is more important to do well on tightenings than on merges. In summary, the extra

[10]There exist more than 8 merges because the algorithm must be reset after each complete failure.

	4 plans		8 plans		12 plans	
	Off	On	Off	On	Off	On
Avg Util value	316	316	1313	1314	2596	2598
Min Util value	316	316	1305	1308	2582	2588
Max Util value	317	317	1321	1320	2604	2603
Avg Tighten Time	.109	.055	4.24	3.77	7.83	7.08
Min Tighten Time	.004	.000	0.15	0.03	0.92	0.19
Max Tighten Time	.765	.607	12.02	11.80	15.40	15.45
Percent Optimal	100	100	77	78	54	57

Table 6.2: Results for the `GAPD` Autominder test. The table shows statistics for the outcome of each call to `GAPD` following a tightening operation. "Percent Optimal" refers to the percentage of calls that provably found the optimal solution. "Off" and "On" denote whether the dynamic elements of the algorithm were enabled.

complexity of maintaining the solver state is worth the effort.

Dynamic GAPD

We repeated the same tests for the `GAPD` algorithm and present the results in Table 6.2.

For the 4 plan problems, both versions of the solver found the optimal solution after each tightening. The dynamic version exited in about half the time on average. For the 8 plan problems, both algorithms found the optimal solution for a smaller percentage of tightenings, but the dynamic version produced slightly better solutions with a 10% time savings. For the largest problems, the solvers found the optimal solution about half of the time and the dynamic version produced slightly better solutions in slightly less time. For perspective, consider the `GAPD` experimental results in Section 5.4. The relative improvement from the 15 second mark to the 90 second mark in the anytime curves ranges from 5-10%—this is roughly an upper bound for the performance increase attainable by `dynamic GAPD` over `GAPD`. Since regular `GAPD` found the optimal solution for a significant percentage of problems, we could expect the relative improvement to be even lower. However, it is possible for the dynamic

version to significantly increase the percentage of calls that find the optimal solution (although it does not in our experiments).

Overall, the performance of `dynamic GAPD` relative to `GAPD` is unimpressive. We believe two factors contributed to this:

- The performance of `GAPS` exhibits the following qualitative behavior: fast improvement of solution quality in a beginning phase, followed by a longer phase where it proves the best solution is optimal. In this second phase, solution quality increases very little (because it is very close to optimal). We believe that both the dynamic and regular versions of `GAPD` are spending most of the computation time in the second phase.

- The Autominder problems are highly structured. Unlike randomly generated problems, these problems often have solutions in the "easy" part of the space. This explains the high rate of finding optimal solutions in the 8 plan case despite the relative large DTPP size (100-150 constraints).

We tested the first factor by rerunning the tests using a 500ms cutoff instead of a 15 second cutoff. We found that for the 8 plan case, there remained little difference between the average preference values, but there was a greater difference in the percent of calls that proved optimality (55% vs. 62%). The result for the 12 plan case were similar: no difference in solution quality, but a more pronounced difference in the number of calls that proved optimality (19% vs. 35%).

The influence of the second factor cannot be easily tested experimentally. However, that the lowest preference value is so close to the maximum suggests that the influence is great. Despite our attempts to generate problems that were "interesting", the highly structured Autominder plans had regions that were under-constrained.

Both algorithms took advantage of this and quickly found near-optimal solutions even for very large problems.

We can indirectly test the second factor by comparing `GAPD` and `dynamic GAPD` on unstructured (i.e., randomly generated DTPPs). We generated 50 instances of DTPPs of three different sizes: (1) 10 events and 30 constraints; (2) 20 events and 60 constraints; and (3) 60 events and 120 constraints. All preference functions were semi-convex with a maximum of 10 preference levels. For each instance, we enacted a series of 5 tightenings, where each tightening consisted of 5 constraints being reduced by 1/3. For example, if an STPP constraint had the base interval $[10, 40]$, we reduced it to an interval of width 20, such as $[15, 35]$. Then, after each tightening, we ran both `dynamic GAPD` and `GAPD`, recording the objective value and whether the optimal solution was found. We let each algorithm run for 2 seconds after each tightening.

	10/30	30/60	60/120
Dynamic GAPD objective value	168.5	268.6	431.4
GAPD objective value	167.4	262.2	427.1
% increase	0.7	2.5	1.0

Table 6.3: Results from applying `dynamic GAPD` to randomly generated problems.

The results are summarized in Table 6.3. The table shows that even for unstructured problems, the solution quality of `dynamic GAPD` only slightly surpasses `GAPD`. The averages in the table hide the fact that there was often a 5-7% difference between solution quality of `GAPD` and `dynamic GAPD`; `GAPD` scored better than `dynamic GAPD` on about a third of the problems. We conclude that the cost of updating the large amount of state in GAPD absorbs most of the gains promised by making the algorithm dynamic.

Conclusion

In summary, the dynamic version of `DTPP_Maximin` algorithms achieve substantial gains over its non-dynamic counterpart, while the dynamic version of `GAPD` showed only a slight performance improvement on Autominder problems. Both `DTPP_Maximin` and `GAPD` proved to be practical for their intended application: they took sufficient advantage of the fact that many Autominder constraints are STPP constraints (no disjunctions) or DTP constraints (no preferences), while still effectively managing the more expressive constraints.

CHAPTER VII

Conclusions

This dissertation extended instances of the Temporal Constraint Satisfaction Problem (TCSP) to include preferences. The extension addressed limitations of TCSPs in the context of planning and scheduling applications: the hard-bounded constraints in TCSPs often do not adequately model reality, and consequently the schedules produced by TCSP algorithms are brittle, making failure recovery necessary when no real-world failure exists.

The addition of preferences changes TCSPs from a constraint satisfaction problem to a constraint optimization problem, requiring the introduction of an objective function. We addressed two objective functions, both of which aggregate local preference values that indicate how well a schedule individually satisfies each constraint. The first was the *min* function, which leads algorithms to maximin-optimal schedules. The second was the *sum* function, which leads algorithms to utilitarian-optimal schedules. We also addressed two types of TCSPs: Simple Temporal Problems (STPs) and Disjunctive Temporal Problems (DTPs).

For each combination of TCSP-type and objective function, our focus was to develop algorithms that were practically applicable to real-world problems. In some cases, this focus led to anytime algorithms in which the cost of "upgrading" to pref-

erences was minimal or even zero. In other cases, we aimed to keep worst-case complexity of the optimization algorithms comparable to the equivalent satisfaction problem. In all cases, we empirically evaluated the algorithms on randomly generated problems, finding which aspects of the problems were "difficult" for each algorithm. We also evaluated the DTPP algorithms when integrated into Autominder, a planning and scheduling system.

The overall contribution of this thesis is a set of algorithms for solving different types of TCSPs with preferences. Together, the algorithms show that preferences can be added at very little cost to systems that already use STPs or DTPs.

In this chapter, we start by stating the result for each tested hypothesis. Then, we explain the individual contributions of this dissertation. Finally, we briefly discuss directions for future work.

7.1 Outcome of Hypotheses

In this section, we reiterate the hypotheses originally presented in Section 1.4, interleaving the results that apply to each hypothesis.

1. Greedy search is effective for finding high-quality, approximate utilitarian solutions to STPPs.

   (a) For STPs with *semi-convex* preference functions and less than 10 preference levels per function, greedy search will find solutions above 80% of optimal for 95% of problem instances.

   (b) For STPs with *unrestricted* preference functions and less than 10 preference levels per function, greedy search will find solutions above 70% of optimal for 95% of problem instances. This property will hold in cases

where the number of local minima in each preference function is less than 5.

Part (a) of this hypothesis turned out to be overly ambitious. For semi-convex problems, greedy search found solutions above 80% of optimal for an average of **84.7**% of problems with sizes up to 24 constraints—much less than the 95% predicted. Of course, decreasing the percent of optimality required significantly changes the result: greedy search found solutions above 70% of optimal for 97% of the same problems.

Part (b) of the hypothesis proved to be true. In unrestricted problems, greedy search found solutions with value above 70% of optimal for an average of 96% of problems. See Section 3.5.5 for a more detailed explanation.

2. Finding maximin optimal solutions to DTPPs requires only slightly more effort than solving DTPs with hard constraints in the worst case.

   (a) For DTPs with *semi-convex* preference functions, the *additional* cost of finding a maximin optimal solution over finding a solution to the level DTP at the *solution level* is cubic in the number of events in the DTPP for the worst case.

   (b) For DTPs with *unrestricted* preference functions, the *additional* cost of finding a maximin optimal solution over finding a solution to the *largest* level DTP is cubic in the number of events in the DTPP for the worst case.

   Both parts of this hypothesis were shown to be false in the analysis presented in Section 4.5.

For DTPs with *semi-convex* preference functions, the additional cost of finding a maximin optimal solution over finding a solution to the *lowest level* DTP (not the solution level) is cubic in the number of events in the DTPP for the worst case. This hypothesis became true, however, for problems in which the search-order partition property held (see Section 4.5.1).

For DTPs with preference functions *in which every peak is at the top level*, the additional cost of finding a maximin optimal solution over finding a solution to the *largest* level DTP is cubic in the number of events in the DTPP for the worst case.

This analysis is not as favorable as we had originally thought. However, when anytime performance is considered, it is clear that upgrading to preferences has no cost even in the average case.

3. Although finding utilitarian solutions to DTPPs has extremely high worst-case complexity, an anytime algorithm can be constructed that is clearly superior to a leading algorithm that solves DTPs without preferences [79]. Specifically, there will be no situation in which the DTP algorithm outperforms the DTPP algorithm.

   This hypothesis was trivially satisfied by the general approach to `GAPD`. The initial solution for `GAPD` was found using an efficient DTP solver, meaning the first solution was found in the same amount of time as would have been required if preferences were not included. After the first solution is found, `GAPD` quickly improves it, using techniques discussed in Section 5.3.

4. The DTPP algorithms (maximin and utilitarian) can be made *dynamic*, meaning that the modified algorithm can solve a sequence of related DTPPs faster

than the unmodified algorithm can solve the individual DTPPs separately.

This hypothesis was true for the `dynamic DTPP_Maximin` algorithm, which found optimal solutions in about half the time as the non-dynamic version. Technically, the hypothesis was true for the dynamic `GAPD` algorithm as well. However, the performance increase was marginal, indicating that the `dynamic GAPD` algorithm has little practical benefit over the static version.

7.2 Contributions

We now reiterate and explain the individual contributions of this dissertation.

7.2.1 Algorithms for finding utilitarian-optimal solutions to STPPs

In Chapter III, we introduced a method for finding high-quality utilitarian solutions to STPs with *semi-convex* and *unrestricted* preference functions. The method involves a conversion of the search space using a procedure and structure called a *preference projection*. We presented several heuristics for greedily searching the converted space and showed which of these heuristics perform best and at what cost.

As noted above, the best of these heuristics found solutions whose values were above 70% of optimal for 95% of problem instances. Importantly, given that the best heuristics broke ties randomly, running `STPP_Greedy` multiple times significantly improved solution quality.

We extended `STPP_Greedy` to an anytime and complete algorithm, called `GAPS`, for finding exact utilitarian-optimal solutions to STPs with *unrestricted* preference functions (Section 3.3). `GAPS` leverages the anytime performance of `STPP_Greedy` while ensuring completeness. The algorithm combined the general strategies of greedy search, divide-and-conquer, and search-space pruning. The result was an algorithm

that quickly found solutions with value >85% of the optimal value and eventually proved it had found the optimal solution. Before our work, all research concerning preferences in TCSPs had focused on preference functions that are either convex or semi-convex.

7.2.2 Algorithm for finding maximin-optimal solutions to DTPPs

In Chapter IV, we presented `DTPP_Maximin`, which finds maximin optimal solutions to DTPPs. This algorithm operates on an extended version of preference projections. The analysis showed that for when certain conditions hold, worst-case time requirement for finding an optimal solution barely exceeded the worst-case requirements for finding solutions to a DTP of the same size.

For the case in which all preference functions are semi-convex, the complexity was only $|A| \cdot |X^3|$ greater than that of solving the lowest level DTP. A is the set of all preference values and X is the set of all events to schedule. For the case in which all preference functions peak at the highest level, and for a third case called the search-order partition case, the complexity was $|A| \cdot |X^3|$ greater than solving the level DTP above the level that contains all optimal solutions. For problems that were both semi-convex and satisfy the search-order partition, the complexity of `DTPP_Maximin` is equal to the complexity of solving the level DTP *at* the solution level.

The average-case performance is not as impressive, due to several factors discussed in Section 4.7.3. However, when anytime performance is considered, there is still no justification for using a hard constraint algorithm in its place. It finds the first solution at least as fast as a leading hard constraint algorithm and quickly improves the solution.

7.2.3 Algorithm for finding utilitarian-optimal solutions to DTPPs

In Chapter V, we presented an anytime and complete algorithm, called `GAPD`, for finding utilitarian-optimal solutions to DTPPs. The algorithm leverages hard DTP algorithms to find an initial solution in time equal to the non-preferences case, ensuring that the cost of upgrading to preferences is zero. Once the initial solution is found, it uses `GAPS` to quickly improve the solution. After the initial phase, `GAPD` uses a model of `GAPS` performance to choose where to spend available computation time.

Although `GAPD` is complete, it is designed for anytime performance, not for finding the optimal solution quickly. Another algorithm for solving DTPPs [77] is designed for quickly finding the optimal solution, and outperforms `GAPD` on small problems. However, as problem size increases, the value of `GAPD`'s anytime focus becomes evident (Section 5.4.5).

7.2.4 Dynamic algorithms for solving DTPPs

In Chapter VI, we defined data structures and algorithms for solving and updating dynamic DTPPs, where constraints in a DTPP change as time progresses. Information from previous searches is leveraged when finding new high-quality solutions after one or more constraints change. For the `DTPP_Maximin` algorithm, significant speedups were realized after implementing the dynamic version. For `GAPD`, the results were less impressive due to the large quantity of state that must be updated before restarting the algorithm. The main contribution of this chapter was empirical evidence that even the non-dynamic versions of `DTPP_Maximin` and `GAPD` can be practically integrated into a real world application. Our results showed that `dynamic DTPP_Maximin` found the optimal solutions well under the allocated time and that

GAPD found the optimal solution about half the time, even for large problems.

7.3 Future Work

7.3.1 Renewable resources

The applicability of TCSPPs is limited by their implicit representation of renewable resources. Many real scheduling problems involve reasoning simultaneously about both temporal and resource constraints. DTP(P)s can reason about renewable resources, but only implicitly through non-overlapping constraints. For example, to encode the fact that two actions, A and B, use the same resource and cannot coincide temporally, a non-overlapping constraint can be included: $A_S - B_E < 0 \vee B_S - A_E < 0$. This is a qualitative constraint encoded as a quantitative constraint and is reasoned about using methods designed for quantitative constraints. Since the qualitative constraint in this case is strictly less expressive, there exists a possibility that non-overlapping constraints can be more efficiently handled using a special mechanism.

We note that we are interested in a problem somewhat different than job shop scheduling, where the goal is usually to maximize the use of some set of resources. We are interested in scheduling problems in which the goal is to maximize temporal preference, while *respecting* resource constraints. In other words, resources are a secondary concern, rather than the primary concern. The focus on time rather than resources is often more appropriate when reasoning about the plans and schedules of people.

7.3.2 Consumable resources

In addition to renewable resources, we would like to model metric consumable resources as well. Consumable resources are those that are used in a particular quantity, such as energy or area. We believe that preference functions, which in this

thesis were used to describe human-level preferences, can be used to model metric consumable resource usage. For example, imagine a set of physical agents sharing a common pool of energy. The goal may be to have each agent execute their plans in a way that minimizes total energy usage, which can be modeled using preference functions between the start and end of actions that require energy. Previous work has addressed this basic problem for STPs and consumable resources, but they were solving a satisfaction problem, not an optimization problem [30]. Furthermore, they modeled consumable resources in a structure separate from the temporal elements.

We first want to explore how algorithms in this dissertation can be applied to specific problems that include metric resources. Second, we want to identify important classes of problems that cannot be directly modeled with a DTPP and then modify our representations and algorithms to reason about them.

7.3.3 Over-constrained DTPPs

Another limitation of DTPPs (and STPPs) is that they assume all constraints must be satisfied and all events must be included in a schedule. This assumption burdens the knowledge engineer with the task of deciding how many events and constraints to include in the problem. Ideally, the algorithm should use the relative importance of events to decide what to include and exclude—to try and fit as much as possible into a plan (increase plan capacity) while maintaining a certain level of quality in the plan. We have done preliminary work on this problem using very simple evaluations of plan quality and plan capacity [60].

7.3.4 Hybrid DTPPs

The DTP formalism is very expressive—many different types of constraints, both qualitative and quantitative, can be encoded using DTPs. However, the representa-

tion of constraints involving non-temporal elements is usually clumsy and inefficient. Since many applications require simultaneous reasoning about temporal and logical constraints, it is desirable to define formalisms and algorithms that support efficient reasoning about both. Our work in [49] is a step toward effectively reasoning about non-temporal constraints and DTP constraints. Our work in [77] is a strong step toward effectively reasoning about non-temporal constraints and DTPP constraints. However, more work is needed before reasonably-sized hybrid DTPPs can be solved or at least approximately solved.

APPENDICES

APPENDIX A

GAPS Implementation Details

In Chapter III, we presented the basic `GAPS` algorithm, hiding many details crucial for good performance in practice. In particular, our implementation of the priority queue allows a significant reduction both in the space necessary to represent a large number of STPP subspaces and the time necessary for accessing and maintaining the queue.

We start by repeating the `GAPS` algorithm in Figure A.1, and then expand certain functions.

The first step of the algorithm calculates the preference projections (line 1), which compose the first subspace to be processed (subspace (a) in our example partition, repeated in Figure A.2). The iterative part of the algorithm begins by calculating the upper bound for the subspace denoted by *currentSS* (line 5). The upper bound is the sum of the highest possible level for each projection. In subspace (d), for example, the upper bound is 8 (2+2+4)[1]. Next, if the upper bound is greater than the best solution found thus far, STPP_Greedy is called to find a consistent component STP within *currentSS* (line 6). If the value of the greedy solution exceeds the best found so far, the solution is stored (lines 8-10).

[1]A tighter upper bound is calculated within STPP_Greedy. The tighter bound is passed on to a subspace's children.

```
GAPS(STPP)
1.  currentSS ← project(STPP)
2.  ssQ ← priority queue of subspaces, initially empty
3.  bestValue ← -1; best ← NO_SOLUTION
4.  REPEAT
5.      IF bestValue < calcUB(currentSS)
6.          gSol ← STPP_Greedy(currentSS)
7.          value ← sumPreference(gSol)
8.          IF value > bestValue
9.              best ← gSol
10.             bestValue ← value
11.         END
12.         addToQueue(ssQ, partition(currentSS, gSol))
13.     END
14.     currentSS ← getNextSS(ssQ);
15. WHILE (ssQ not empty)
RETURN best
```

Figure A.1: A high-level description of the STPP algorithm `GAPS`.

Next, *currentSS* is partitioned based on the greedy solution *gSol* and adds the returned subspaces to a priority queue (line 12). Finally, a new subspace is retrieved from the priority queue (line 14) and the next iteration begins. The iterations continue until the priority queue empties, a signal that the optimal solution has been found. Not shown are additional stopping conditions that allow the algorithm to quit after a time threshold or after a specified number of iterations.

Lines 12 and 14 hide most of the complexity of the algorithm. The `partition` method within Line 12 was described in detail in Section 3.3.1, but the description makes no comment on the representation of the subspaces or how they are stored in the queue. The naive representation, which is to create a copy of all constraints included in the subspace, is clearly impractical, given that thousands or millions of subspaces can exist in the queue at once. For the same reason, implementing the queue as a list would require significant insertion and removal costs. Therefore,

we developed a subspace representation and queue implementation that minimized memory usage while allowing quick insertion and removal.

A.1 Subspace Representation

In the partition method, subspaces are created by copying and splitting preference projections in the subspace's "parent". The result of each split is a new preference projection that represents a subtree of the original preference projection tree. The split either defines a new root for the tree or sets a limit on the level. For example, refer to the second small partition in Figure A.2 — from subspace (b') into (c1), (c2), and (c'). The middle tree is split into three trees: the tree in (c') retains the same root, but is cut above level 2; the trees in (c1) and (c2) are given new roots, but retain the same cut level (implicitly 4).

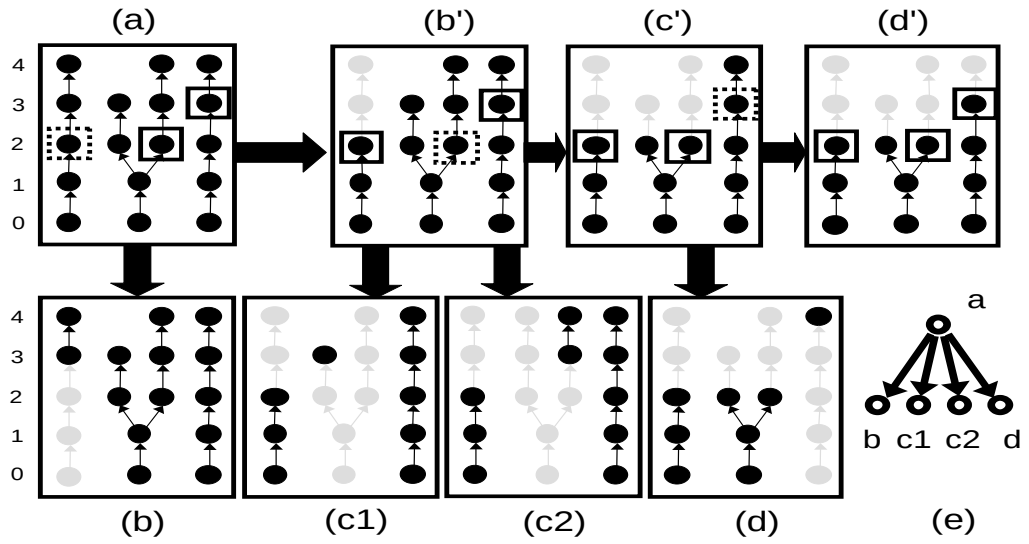


Figure A.2: Subspace partitioning process used in GAPS.

We can therefore represent each tree with two elements: the STP constraint that forms its root and the level at which it is cut. Thus, we can represent a subspace with a set of pointers to root constraints and a set of limits. If we have m constraints,

a subspace can be represent with $2m$ elements.

There is additional structure that lets us further reduce these requirements. Recall that each subspace is produced by modifying a single tree. Therefore, we only need to store the root and cut level for the modified tree in each subspace; the root and cut level for other constraints in the subspace can be extracted from the parent and siblings of the subspace. To illustrate this fact and to explain how the `getNextSS` method operates in Line 14 of the algorithm, we will describe how subspace (c2) in Figure A.2 is created using only its parent and the root of each of its siblings. Figure A.2(e) shows the relationship between (c2), its parent (a), and its siblings (b), (c1), and (d).

To define (c2), we must define the root and cut level of each of its three constraints. For the second constraint (MC2), the root and cut level is stored internally. Note that the cut level for this constraint is the same as the subspace (a), the parent of (c2), because (c2) was given the part of MC2 that was above the greedy solution.

To get the root of the first constraint (MC1), it uses only the parent (a) and its sibling (b), which was created during the partition of the first constraint. Since (b) was created before (c2), the trees in (c2) will be the part of MC1 *not* represented in (b). Therefore, the cut level will be the level below the root of (b) and the root of (c2) will be the same root found in (a). (See Figure A.2 to verify this).

To define the third constraint of (c2), we simply copy the root and cut level from its parent (a). Subspace (d) does not need to be referenced because (c2) was created before (d).

We now summarize the algorithm for recreating a subspace that was created by partitioning the i^{th} constraint of some subspace S:

1. For the i^{th} tree

   (a) The root of tree i is stored internally.

   (b) Let the cut level for tree i be the same as the cut level for tree i in S

2. For all trees $e < i$

   (a) Let the root be the same as the root for tree e in S

   (b) Let the cut level be 1 less than the root level for a sibling created by partitioning the e^{th} constraint of S[2]

3. For all trees $e > i$

   (a) Let the root be the same as the root for tree e in S

   (b) Let the cut level be the same as the cut level for tree e in S

Notice that the cut levels referenced in this algorithm are all derived from the root of some other subspace (step 3b). Therefore, the cut levels do not need to be stored. For each subspace, we only need to store the root of a single projection tree and a pointer to all of its children and a pointer to its parent.

A.2 Memory Requirements

Four elements are needed to represent each node in the tree of subspaces:

1. An integer denoting the constraint that was partitioned to created the subspace

2. A pointer to a single STP constraint, which identifies the root of the projection tree created when the subspace was created

3. A pointer to the subspace's parent subspace

[2]If multiple subspaces were created when the e^{th} partition was performed, all subspaces will necessarily have roots at the same level, so any of the subspaces can be referenced.

4. A pointer to each of its children

The first item, the index into the set of constraints, does not need to be stored in the node — it can be stored inside the projected STP constraints. The value can be accessed through the root STP constraint. The fourth item does not exist in the leaf subspaces, so the average memory requirements are very small: 3 pointers for each node. As we explain in the following section, we use the subspace tree for other purposes, so the actual memory requirements are a bit larger.

In addition to the subspace tree, `GAPS` must also store the preference projections, pointers to the best STP found so far, and the state required for `STPP_Greedy`. None of these structures require significant space.

A.3 Priority Queue Implementation

The tree of subspaces just described provides the backbone for our priority queue. If we add to each node a priority value and a pointer to its child with the highest priority, we have an efficient method for maintaining which subspace has the highest priority. After partitioning a subspace, its children are added to the tree, along with the result of a priority calculation for each. The priority is then propagated up the tree. When the `getNextSS` method is called in Line 14 of the `GAPS` algorithm, the path from the root of the tree to the highest priority subspace can be followed using the pointers to the high priority child. Moreover, the algorithm for recreating the subspace can be executed while following the path.

To monitor memory usage, we add a single variable to each node in the tree that holds the size of its subtree. The root node will then have access to the amount of memory used by the tree.

Because we need all ancestors of a subspace to recreate it, we cannot remove a

subspace from the queue after running `STPP_Greedy` on it and partitioning it. We remove a subspace from the queue after it has been fully searched — a subspace is fully searched when all of its children are fully searched. The recursion bottoms out because each single constraint partition does not always partition a subspace into two parts. For example, consider subspace (d) in Figure A.2. A single constraint partition on the third constraint would produce only one subspace, because the projection tree contains a single constraint.

Another way a subspace node can be removed is if the upper bound for all of its descendant subspaces is less than or equal to the best value found so far. To efficiently determine when this case exists, we also add to each node two integers representing the value of the best STP found in the subtree and the highest upper bound for any subspace in the subtree. These values are updated in linear time each time a subspace is processed or added.

APPENDIX B

Integer Programming Formulation

We now describe the Integer Programming formulation used for finding utilitarian optimal solutions for DTPPs with unrestricted functions. Our goal in this formulation was to reduce the number of integer variables used, reducing the number of "cuts" required by whatever algorithm CPLEX chooses for the problem.

To build up to the formulation we used, we first describe how to encode STPPs with convex functions. Then, we extend it to DTPPs with convex functions and show how to remove the convexity restriction.

B.1 Convex STPP Formulation

Our formulation builds on the linear programming formulation used in [52] to solve STPPs with convex functions. In this approach, each convex function was represented as piecewise-linear. Each constraint was defined by a pair of events, X_i and X_j, and a set of slope/offset pairs that defined each line in the piecewise-linear function, $\{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \ldots, \langle a_n, b_n\rangle\}$. For example, the first line is defined as $f = a_1(X_j - X_i) + b_1$. Figure B.1 shows an example of a preference function defined in this way.

The variables in the LP were the events and a single local preference value for each constraint between X_i and X_j, Z_{ij}. The constraints in the LP relate the Z

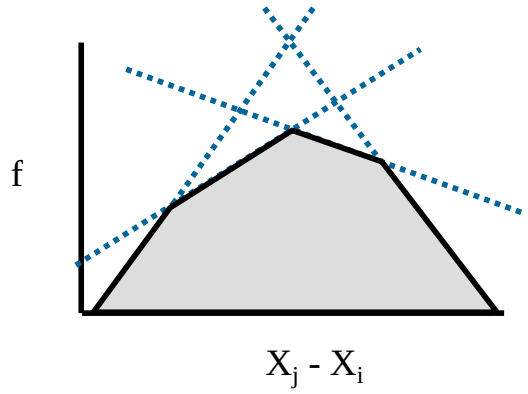


Figure B.1: A convex preference function defined by a set of linear functions.

variable for each constraint to the constraint's linear functions. Intuitively, they force Z to have a value that falls within the gray area in Figure B.1. This can be implemented by a set of linear constraints requiring the Z value to be greater than 0 and less than the vertical coordinate of each line:

$$\begin{aligned} a_1(X_j - X_i) + b_1 &\leq Z_{ij} \\ a_2(X_j - X_i) + b_2 &\leq Z_{ij} \\ &\cdots \\ a_n(X_j - X_i) + b_n &\leq Z_{ij} \\ -Z_{ij} &\leq 0 \end{aligned}$$

Since the goal is to maximize the utilitarian value, the objective function is simply $\sum Z$.

B.2 Convex DTPP Formulation

To extend this approach to handle DTPPs, we can introduce a binary variable for each disjunct i of each DTPP constraint k, Y_{ki}. We then express that at least one disjunct of each constraint must be "activated" in a solution using the following

linear constraints:

$$\sum_i Y_{ki} \geq 1, \forall k.$$

We then modify each constraint involving the Z variables to reference the Y variable associated with the constraint. The following is an updated version of the constraints above using a Y variable and a large constant M:

$$\begin{aligned}
a_1(X_j - X_i) + b_1 + M \cdot Y &\leq Z_{ij} + M \\
a_2(X_j - X_i) + b_2 + M \cdot Y &\leq Z_{ij} + M \\
&\dots \\
a_n(X_j - X_i) + b_n + M \cdot Y &\leq Z_{ij} + M \\
-Z_{ij} &\leq 0 \\
Z_{ij} - M \cdot Y &\leq 0
\end{aligned}$$

If the Y variable is "true" (equal to 1), then the M constants cancel, leaving the original constraint. If the Y variable is "false" (equal to 0), then the large M constant on the right-hand-side makes the constraint trivially satisfied, effectively removing the constraint. Thus, only linear constraints corresponding to activated disjuncts are present in the linear program. An extra constraint is needed to force Z_{ij} to '0' in the deactivated case, preventing the unbounded variable from influencing the objective function.

This is a common technique called the *big-M* method. The value of M is the smallest value necessary to ensure that any deactivated constraint is trivially satisfied.

B.3 Non-convex DTPP Formulation

To handle preference functions that are not convex, we first note that any non-convex piecewise-linear function can be split into a set of convex functions. Figure B.2 shows a non-convex function split into two convex functions.

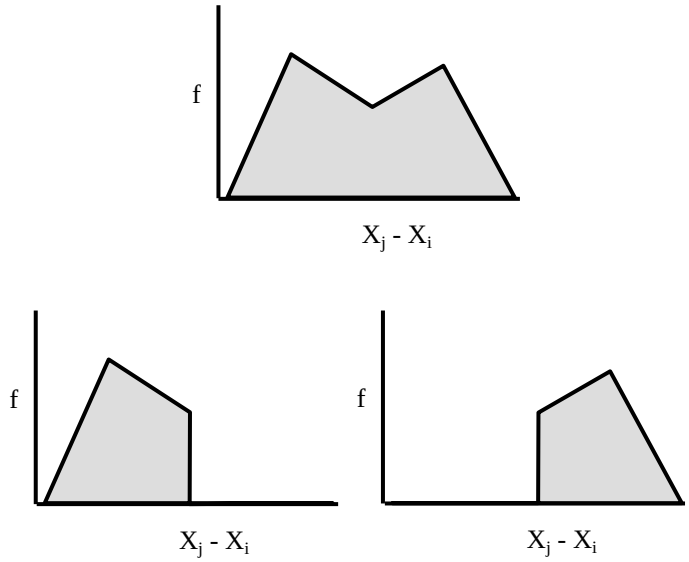


Figure B.2: A non-convex preference function split into two convex functions.

If we treat each of the new convex functions as a separate disjunct, we can use the formulation described in the previous section. The cost is the addition of more binary variables to the IP.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] James F. Allen. Toward a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[3] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[4] Kenneth J. Arrow. *Social Choice and Individual Values*. John Wiley and Sons, 1963.

[5] Gustavo Arroyo-Figueroa and Luis E. Sucar. A temporal bayesian network for diagnosis and prediction. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 13–20, 1999.

[6] Carlo Berzuini. Representing time in causal probabilistic networks. In *Proceedings of the Fifth Conference on Uncertainty in Artificial Intelligence*, pages 15–28, 1990.

[7] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[8] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *Journal of Artificial Intelligence Research*, 21:135–191, 2004.

[9] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. Preference-based constrained optimization with CP-nets. *Computational Intelligence*, 20(2):137–157, 2004.

[10] Ronen Brafman and Carmel Domshlak. Introducing variable importance tradeoffs into CP-nets. In *Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence*, pages 69–76, 2002.

[11] Ronen I. Brafman and Moshe Tennenholtz. An axiomatic treatment of three qualitative decision criteria. *Journal of the ACM*, 47(3):452–482, 2000.

[12] Roberto Cervoni, Amedeo Cesta, and Angelo Oddi. Managing dynamic temporal constraint networks. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 13–18, 1994.

[13] Luca Chittaro and Angelo Montanari. Temporal representation and reasoning in artificial intelligence: Issues and approaches. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):47–106, 2000.

[14] Berthe Y. Choueiry and Lin Xu. An efficient consistency algorithm for the temporal constraint satisfaction problem. *AI Communications*, 17(4):213–221, 2004.

[15] David A. Cohen, Martin C. Cooper, Peter Jeavons, and Andrei A. Krokhin. A maximal tractable class of soft constraints. *Journal of Artificial Intelligence Research*, 22:1–22, 2004.

[16] Amy Cohn and Cynthia Barnhart. Improving crew scheduling by incorporating key maintenance routing decisions. *Operations Research*, 51:387–396, 2003.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[18] Ernest Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann, 1990.

[19] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[20] Thomas L. Dean and Drew McDermott. Temporal data base management. *Artificial Intelligence*, 32(1):1–55, 1987.

[21] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[22] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 37–42, 1988.

[23] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[24] D. Dubois, Hélène Fargier, and Henri Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of Second IEEE International Conference on Fuzzy Systems*, page 11311136, 1993.

[25] Didier Dubois, Hélène Fargier, and Henri Prade. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence*, 6:287309, 1996.

[26] Boi Faltings, Djamila Haroud, and Ian Smith. Dynamic constraint propagation with continuous variables. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 754–758, 1992.

[27] Hélène Fargier and Jérôme Lang. Uncertainty in constraint satisfaction problems: a probalistic approach. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104, 1993.

[28] Hélène Fargier, Jérôme Lang, and Thomas Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. *Proceedings of the First European Congress on Fuzzy and Intelligent Technologies*, 3:1128–1134, 1993.

[29] M. S. Franzin, Francesca Rossi, Eugene C. Freuder, and Richard Wallace. Multi-agent constraint systems with preferences: Efficiency, solution quality, and privacy loss. *Computational Intelligence*, 20(2):264–286, 2004.

[30] Eugene C. Freuder and Richard J. Wallace. Dispatchable execution of schedules involving consumable resources. In *Proceedings Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 283–290, 2000.

[31] Judy Goldsmith. Preferences and domination. In *Algebraic Methods in Computational Complexity*, number 04421 in Dagstuhl Seminar Proceedings, 2005.

[32] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431 – 437, 1998.

[33] Steve Hanks and Drew McDermott. Default reasoning, nonmonotonic logics and frame problem. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 328–333, 1986.

[34] Ulrich Junker. Preference-based search and multi-criteria optimization. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 34–40, 2002.

[35] Keiji Kanazawa. A logic and time nets for probabilistic inference. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 360–365, 1991.

[36] Keiji Kanazawa. *Reasoning about Time and Probability.* PhD thesis, Brown University, 1992.

[37] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 674–682, 2002.

[38] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs.* Cambridge, New York, 1993.

[39] Lina Khatib, Paul Morris, Robert Morris, and Francesca Rossi. Temporal constraint reasoning with preferences. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 322–327, 2001.

[40] Lina Khatib, Paul Morris, Robert Morris, and Kristen Brent Venable. Tractable pareto optimal optimization of temporal preferences. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1289–1294, 2003.

[41] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[42] T. K. Satish Kumar. A polynomial-time algorithm for simple temporal problems with piecewise constant domain preference functions. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 67–72, 2004.

[43] Javier Larrosa and Thomas Schiex. Solving weighted CSPs by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[44] R. Duncan Luce and Howard Raiffa. *Games and Decisions*. John Wiley and Sons, New York, 1957.

[45] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[46] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie, editor, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[47] Pedro Meseguer, Noureddine Bouhmala, Taoufik Bouzoubaa, Morten Irgens, and Martí Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.

[48] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32, 1990.

[49] Michael D. Moffitt, Bart Peintner, and Martha E. Pollack. Augmenting disjunctive temporal problems with finite-domain constraints. In *Proccedings of the 20th National Conference on Artificial Intelligence*, pages 1187–1192, 2005.

[50] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[51] Leora Morgenstern. The problem with solutions to the frame problem. In *The robot's dilemma revisited: the frame problem in artificial intelligence*, pages 99–133. Ablex Publishing Corp., 1996.

[52] Paul Morris, Robert Morris, Lina Khatib, Sailesh Ramakrishnan, and A Bachmann. Strategies for global optimization of temporal preferences. In *Tenth International Conference on Principles and Practice of Constraint Programming*, pages 408–422, 2004.

[53] Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on AI*, pages 494–499, 2001.

[54] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 444–452, 1998.

[55] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[56] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[57] Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.

[58] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[59] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[60] Bart Peintner, Michael D. Moffitt, and Martha E. Pollack. Solving over-constrained DTPs with preferences. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 202–211, 2005.

[61] Bart Peintner and Martha E. Pollack. Low-cost addition of preferences to DTPs and TCSPs. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 723–728, 2004.

[62] Bart Peintner and Martha E. Pollack. Anytime, complete algorithm for finding utilitarian optimal solutions to STPPs. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 443–448, 2005.

[63] Javier Pinto and Raymond Reiter. Temporal reasoning in logic programming: a case for the situation calculus. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 203–221, 1993.

[64] Martha E. Pollack, Laura Brown, Dirk Colbry, Colleen E. McCarthy, Cheryl Orosz, Bart Peintner, Sailesh Ramakrishnan, and Ioannis Tsamardinos. Autominder: An intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*, 44(3-4):273–282, 2003.

[65] Steven Prestwich, Francesca Rossi, K. Brent Venable, and Toby Walsh. Constraint-based preferential optimization. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 461–466, 2005.

[66] Alessandro Provetti. Hypothetical reasoning about actions: From situation calculus to event calculus. *Computational Intelligence*, 12:478–498, 1996.

[67] Raymond Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pages 359–380. Academic Press Professional, Inc., 1991.

[68] Francesca Rossi, Kristen B. Venable, and Toby Walsh. mCP nets: Representing and reasoning with preferences of multiple agents. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 729–734, 2004.

[69] Francesca Rossi, Kristen B. Venable, and Neil Yorke-Smith. Controllability of soft temporal constraint problems. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 588–603, 2004.

[70] Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994.

[71] Fariba Sadri and Robert A. Kowalski. Variants of the event calculus. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 67–81, 1995.

[72] Eugene Santos, Jr. and Joel D. Young. Probabilistic temporal networks: A unified framework for reasoning with time and uncertainty. *International Journal of Approximate Reasoning*, 20(3):263–291, 1999.

[73] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal on Artificial Intelligence Tools*, 3(2):187–207, 1994.

[74] Lehnart Schubert. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In Henry E. Kyburg, Ronald P. Loui, and Greg N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, volume Volume 5, pages 23–67. Kluwer Academic Publishers, Dordrecht / Boston / London, 1990.

[75] Peter J. Schwartz and Martha E. Pollack. Two approaches to semi-dynamic disjunctive temporal problems. In *Proceedings of the International Conference on Automated Planning and Scheduling Workshop on Constraint Processing in Planning and Scheduling*, pages 74–81, 2005.

[76] Michael J. Scott and Erik K. Antonsson. Aggregation functions for engineering design trade-offs. *Fuzzy Sets and Systems*, 99(3):253–264, 1998.

[77] Hossein M. Sheini, Bart Peintner, Karem A. Sakallah, and Martha E. Pollack. On solving soft temporal constraints using SAT techniques. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, 2005.

[78] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120:81–117, 2000.

[79] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for Disjunctive Temporal Reasoning Problems. *Artificial Intelligence*, 151(1-2):43–90, 2003.

[80] Ioannis Tsamardinos, Martha E. Pollack, and Philip Ganchev. Flexible dispatch of disjunctive plans. In *Proceedings of the 6th European Conference on Planning*, pages 417–422, 2001.

[81] Ioannis Tsamardinos, Martha E. Pollack, and Sailesh Ramakrishnan. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proceedings of the International Conference on Automated Planning and Scheduling Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

[82] Peter van Beek. Approximation algorithms for temporal reasoning. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1291–1296, 1989.

[83] Johan van Benthem. Temporal logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming: Epistemic and Temporal Reasoning*, volume 4, pages 241–350. Oxford University Press, 1995.

[84] Kristen B. Venable. alpha-dynamic controllability of simple temporal problems with preferences and uncertainty. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, Doctoral Abstract*, page 999, 2003.

[85] Gerard Verfaillie and Thomas Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI Workshop on Constraint Satisfaction Issues Raised by Practical Applications*, pages 1–8, 1994.

[86] Thierry Vidal. A unified dynamic approach for dealing with temporal uncertainty and conditional planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 395–402, 2000.

[87] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In *Readings in qualitative reasoning about physical systems*, pages 373–381. Morgan Kaufmann Publishers Inc., 1990.

[88] Marc B. Vilain and Henry A. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceeding of the Fifth National Conference on Artificial Intelligence*, pages 377–382, 1986.

[89] Neil Yorke-Smith, Kristen B. Venable, and Francesca Rossi. Temporal reasoning with preferences and uncertainty. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1385–1390, 2003.

ABSTRACT

Algorithms For Constraint-Based Temporal Reasoning With Preferences

by

Bart Michael Peintner

Chair: Martha E. Pollack

Recent planning and scheduling applications have successfully used the Temporal Constraint Satisfaction Problem (TCSP) to model events and temporal constraints between them. Given a TCSP, the task is to find a schedule or a set of schedules that satisfy all constraints in the problem. A key limitation of TCSPs and related formulations is that they only represent hard constraints—constraints that are either satisfied or not. Many real-world situations are better modeled with soft constraints—constraints that are satisfied to a particular degree. Soft constraints allow knowledge engineers to express that some situations are preferable to others.

This dissertation augments TCSPs with soft constraints and presents algorithms that find optimal solutions. We represent soft constraints by augmenting hard constraints with a preference function that maps every valid solution to a local preference value that describes how well the constraint is satisfied. We then aggregate the local

preference values to attain the value of the entire solution. We studied two aggregation functions, the *sum* and *minimum* functions.

We developed a set of algorithms that optimize with respect to both aggregation functions when preferences are added to each of two common TCSP subproblems: the Simple Temporal Problem (STP) and the Disjunctive Temporal Problem (DTP). We analyzed each algorithm experimentally on random problems and showed that the algorithms for DTPs with preferences integrate practically into a planning and scheduling application called Autominder. Before integration, we modified the algorithms to handle *dynamic* DTPs with preferences, equipping them to solve a series of related DTPs with preferences faster than solving them individually.

In many real-world problems, finding the optimal solution is not necessary—it is more important to find near-optimal solutions quickly. Consequently, our focus was on maximizing anytime performance. We achieved this goal, showing empirically that our algorithms quickly found high-quality solutions even for large problems. For all algorithms, we showed that the cost of adding preferences was minimal, achieving our overall goal of making the use of TCSPs with preferences practical in all situations where hard-constraint TCSPs apply.