Managing Complex Scheduling Problems with Dynamic and Hybrid Constraints

by

Peter J. Schwartz

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
2007

To my wife, Courtney, for her strength, for her patience,

and for always believing in me even when I doubted myself.

Acknowledgements

The work described in this dissertation was only possible with the help of many other people. My advisor, Martha Pollack, has been a constant source of encouragement and insight over the past six years. Through her guidance, I have learned more and grown more than I thought I could. Even when I faced obstacles and setbacks, she continued to patiently support and motivate me. She has served as an example of dedication and wisdom, and I hope to carry her passion for knowledge and learning with me as I continue forward.

I would also like to extend my gratitude to the other members of my committee for their many insights and feedback. Professor Stéphane Lafortune served on my committee when I initially presented my proposal, and when his schedule became too busy to continue, Assistant Professor Amy Cohn was kind enough to step in on short notice to serve in his place. I was also fortunate enough to have Professor Ed Durfee and Professor John Laird serve on my committee throughout the process. Each of my committee members provided intelligent and insightful feedback that has helped to strengthen and improve this thesis.

I am very grateful to the people of the NASA Ames Research Center for patiently helping me to understand the complexities of air traffic control and for providing me with the raw data on which I was able to test my ideas. Specifically, Michelle Eshow and

Gregory Wong were kind enough to hold several phone conversations with me and respond to countless emails. I am sure that they had no idea how much work it would be when they first agreed to help me collect this air traffic control data. I would also like to thank the other members of NASA Ames who participated in this process, including Tom Turton, Shawn Engelland, Wardell Lovett, Paul Borchers, Karen Tung, Jinn-Hwei Cheng, and John Robinson; they are a credit to their organization.

I would like to thank my fellow graduate students at the University of Michigan who have helped me along the way. In particular, Bart Peintner and Mike Moffitt, who worked closely with me to help develop my research ideas, as well as Amy Kao and Erin Rhode, who helped me greatly to actually implement my ideas into a working system. Additionally, I would like to thank Julie Weber, Matt Rudary, Brett Clippingdale, Mark Hodges, Andy Nuxoll, Mark Schaller, and Joe Taylor for being such good friends and comrades as we have all done our best to work through graduate school together.

I very much appreciate all of the support I received from my family. I would like to thank my parents, who always encouraged me, my sisters Kate and Susan, who were always there to help lift my spirits, and my brother John, who, more than anyone, really took the time to understand my research and to help me brainstorm new ideas. I would also like to thank the Browns for making many trips from Maryland to Michigan to visit Courtney and me and to help make our house a home.

Above all, I have to thank my wife, Courtney. She is the one person without whom I could not have completed this dissertation. Courtney would not let me give up on my goals, and sacrificed much to help me achieve them. She holds me to a higher standard, and I am a better person because of it.

Table of Contents

List of Figures

List of Tables

Abstract

The task of scheduling can often be a difficult one because of the inherent complexity of real-world problems. In the field of Artificial Intelligence, many representations and algorithms have been developed to automate the scheduling process.

Many state of the art scheduling systems deal with this complexity by making assumptions that simplify the algorithms, but in doing so, miss some opportunities to improve performance. Scheduling problems are temporal in nature, and so they often contain constraints that change over time. Many scheduling systems assume that the problems they are solving are all independent, and so they ignore the similarities between subsequent sets of scheduling constraints. Additionally, scheduling problems often contain a mixture of finite-domain and temporal constraints. Many of the systems that can solve problems of this type do so by creating finite-domain variables to represent the constraints, but then ignore the distinction between the different types of variables when searching for a solution.

In this dissertation, I identify opportunities to improve performance by exploiting structure where it has previously been overlooked. Following this approach, I develop a set of techniques that apply to a wide variety of situations that can arise in real-world scheduling problems. First, I consider dynamic scheduling problems with constraints that change over time. To address such problems, I introduce a new representation called the

Dynamic Disjunctive Temporal Problem, along with several techniques to improve both efficiency and stability when solving one. Second, I consider scheduling problems in which a mixture of finite-domain and temporal variables can interact through hybrid constraints. I introduce the Hybrid Scheduling Problem to represent such problems, and I present a set of techniques that capitalize on the distinction between variable types to improve efficiency across the problem space. Finally, I conclude by proposing several ways that the dynamic and hybrid representations and techniques can be combined. To compare many of the techniques presented throughout this dissertation in the context of structured, real-world problems, I use them to solve scheduling problems based on actual air traffic control constraints recorded from the Dallas/Fort Worth International Airport.

Chapter 1

Introduction

1.1. Introduction

The management of complex schedules is an important but often difficult task. For example, if we need to organize a meeting, we may need to determine when the meeting will start, how long it will last, who will attend, and where it will take place. These decisions can interact with each other (e.g., some meeting rooms may already be reserved at certain times, or the time it takes attendees to travel to the meeting may depend on its location) and may need to change over time (e.g., one of the attendees cannot make it on time because she is stuck in traffic, or the projector breaks in the scheduled meeting room so a different room must be chosen).

Automating the process of solving and detecting inconsistencies in sets of complex scheduling constraints could be helpful in such a situation; in other situations, it may be necessary. Effective schedule management is critical in high risk, low reaction time applications such as emergency response, air traffic control, space missions, or military operations. It is therefore worthwhile to study how to develop such automated schedule management systems.

1.2. Office Example

I will use the example of scheduling an office meeting to illustrate various representations and techniques throughout this dissertation. This example will demonstrate the complexities of many real-world scheduling problems, but should be familiar to most readers.

It is 10:00am, and Bob just got a call from his project manager, Alice. Alice would like to have a team meeting to find out how the project is progressing, and she wants Bob to set it up. She would like Bob, his fellow engineer Carl, and one of the programmers, Donna or Evan, to each give a 15-minute presentation. The meeting will last one hour—15 minutes for each of the three presentations, and 15 minutes for discussion afterward.

The meeting must be held in a room with a digital projector, so the possible locations are the conference room in Alice's building (available 12:00pm to 3:00pm) and the team room in the engineers' building (available 1:00pm to 2:00pm and 3:00pm to 5:00pm). If the meeting is in the conference room, Bob and Carl will need to travel 45 minutes to get there; if the meeting is in the team room, Alice will need to travel 45 minutes to get there. Either way, Donna or Evan will have to travel 30 minutes to get there.

Bob and Carl each need 45 minutes to prepare their presentations. They both need to attend a lunch meeting (in their own building) from 12:00pm to 1:00pm. Bob calls Donna and Evan to set up the meeting, and he finds out that they need to meet for one hour to prepare their presentation. He also finds out when everyone is available, as listed in Table 1.1.

Bob schedules the team meeting for 2:00pm. It will be held at the conference room in Alice's building, and Evan will give the presentation for the programmers. With the schedule in place, Bob starts to prepare his presentation at 10:00am. Everything is going according to plan until he gets a phone call from Donna at 12:00pm. She has spent the last hour working with Evan on their presentation, but the two of them will need an additional half hour to finish. Looking at the available times, Bob sees that it is necessary to push the team meeting back to 3:00pm. The conference room in Alice's building is not available at that time, so it is necessary to change the meeting location to the team room in the engineers' building, meaning that Alice will have to travel to the meeting instead of the Bob and Carl.

Person	**Available Times**
Alice	11:00am – 12:00pm, 2:00pm – 5:00pm
Bob	10:00am – 12:00pm, 1:00pm – 5:00pm
Carl	11:00am – 12:00pm, 1:00pm – 5:00pm
Donna	11:00am – 12:00pm, 2:00pm – 5:00pm
Evan	10:00am – 12:00pm, 1:30pm – 4:00pm

Table 1.1. Times when each team member is available.

This example might seem complex, but such complexity is common in many real-world application domains. In most cases, this complexity must be dealt with by a human. In the meeting scenario, it is up to Bob to sift through everyone's schedule to find an appropriate time for the team meeting. When Donna calls to say that she and Evan need an extra half hour to prepare, it is up to Bob to adjust the schedule to deal with the change. The goal of my research is to extend previous work on constraint-based reasoning to develop new techniques to automate the process of scheduling with finite-domain and temporal constraints that interact and change over time.

1.3. Problem Statement

The characteristics of the scheduling problems that my research will address can be divided into four categories:

1) **Temporal constraints.** Constraints on when events may occur relative to one another, such as when the meeting can start and how long it will last.
2) **Finite-domain constraints.** Constraints on decisions over finite sets of options, such as who will perform a task or attend a meeting.
3) **Interaction.** Interactions between the temporal and finite-domain constraints, such as no two meetings overlapping if any employee is attending both.
4) **Change.** Changes in the constraints over time, such as an employee calling in sick or a project being postponed.

Previous research has already explored representations and algorithms to handle temporal constraints and finite-domain constraints independently, as in (1) and (2) above. I will review this research in Chapter 2. My research will therefore focus on (3) and (4)—temporal and finite-domain constraints that interact, and constraints that change over time. The representations and techniques presented in this dissertation can be partitioned into the following two classes:

1) **Dynamic techniques.** I will extend existing research on dynamic finite-domain constraint satisfaction to develop representations and techniques to improve

efficiency and stability when solving scheduling problems with temporal constraints that change over time.

2) **Hybrid techniques.** I will extend existing research on hybrid constraint satisfaction to develop representations and techniques that improve efficiency when solving scheduling problems with temporal and finite-domain constraints that interact.

Throughout this dissertation, I will describe many different algorithms and techniques to solve scheduling problems with dynamic and hybrid constraints. The first metric by which I will compare the various techniques I present is efficiency. An algorithm for hybrid scheduling problems must be fast enough to produce a solution in a reasonable amount of time despite the complexity of the problems. An algorithm for dynamic scheduling algorithms must be fast enough to keep up with scheduling changes and provide timely feedback. In the office example, Bob must create a schedule and then update that schedule fast enough to let everyone know when and where the meeting will take place so they can prepare.

When considering the efficiency of an algorithm, it is important to carefully choose and define the measure of efficiency that will be used. The difficulty here stems from the fact that the run time of a scheduling problem depends heavily on the number of variables and tightness of constraints in the problem. In my experiments, I purposefully vary both the number of variables and the tightness of constraints between problems in order to test a range of problem types, which results in extreme variability in run time. How, then, should one compare the efficiency of two algorithms?

For time-critical applications, such as emergency response or air traffic control, it is often desirable for the system to meet strict deadlines, so efficiency should be measured in terms of the worst case run time. For other applications, such as an employee scheduling meetings or a receptionist scheduling appointments, users will most likely want to know how long they should expect to wait for the system to produce a solution. In applications like these, it is desirable for the system to solve different problems in a similar amount of time, so efficiency should be measured in terms of the variability in run time. Unfortunately, because of the inherent extreme variability in run time between problems, the worst case and variability of the run times are actually characteristics of the problem set more so than they are characteristics of the algorithms.

For many applications, whether the worst case or variability of run time is important or not, it is desirable to find a solution to a scheduling problem as quickly as possible on average. For example, while an air traffic controller might need a solution schedule by a deadline in order to maintain safety, the controller would probably prefer to have the solution as soon as possible to communicate it to the pilots so they know what to expect. In an office, an employee might prefer that the time required by the scheduling system to produce a solution be somewhat steady, but he or she would probably also prefer to have the solution as soon as possible to move on to other things. Reducing average run time is of interest in many application domains, and (as described in the next paragraph) it is possible to compare the average case run time of two algorithms empirically despite the extreme variability between problems. I will therefore measure the efficiency of an algorithm based on its ability to reduce run time on average.

Given two algorithms that can solve the same type of problem, I will compare their relative efficiency by solving the same set of test problems using both algorithms and observing the run time required by my own implementations. Because each problem is solved using both algorithms (as opposed to solving one test set with one algorithm and a different test set with the other algorithm), the data is *paired*. Given this paired data, I will follow common practice by using the Student's paired t-test, which considers the mean and standard error of the difference between each pair, to determine whether the difference in run time between the two algorithms is statistically significant. The Student's paired t-test makes it possible to compare the average run time of two algorithms despite the extreme variability in run time between problems.

The second metric by which I will compare the various techniques is stability. Stability is the degree to which subsequent solutions are similar to each other, so this metric only applies to the dynamic techniques. In a real-world application, the solution schedules generated by a solver will dictate when certain events or actions should take place. These might be the events or actions of another automated process, or possibly a human user. Maximizing stability can reduce the amount of computation required of another automated process that performs computation over the schedule; it can also reduce the amount of frustration experienced by a human user in reaction to a change in the solution schedule. In the office example, after Bob reschedules the meeting, everyone else must adjust their own schedules to compensate. When solving one problem in the sequence, stability is measured relative to the solution schedule of the previous problem in the sequence. I will measure stability in two ways: 1) the similarity between the solution of one problem and the solution of the previous problem, and 2) the amount of time required to find the most similar solution possible to the solution of the previous problem. I will discuss exactly how I quantify the similarity between solutions later as I discuss dynamic techniques.

Rather than focusing my efforts on techniques that offer dramatic performance gains when applied to special cases or subsets of problems, my goal is to develop broadly applicable techniques that improve efficiency and/or stability regardless of the situation. To achieve this goal, I will consider the performance of existing techniques in a variety of circumstances across the problem space. Some of the techniques that I develop will apply to situations in which existing research is found to be deficient, while other techniques will apply across the board. The end result will be a set of techniques that, used in combination, can improve performance when applied to a wide variety of situations.

Given the types of problems I will consider and the goals of my research, my problem statement can be summarized as follows:

Given a scheduling problem with hybrid and/or dynamic constraints, improve efficiency and (when applicable) stability in a broad variety of scheduling situations by extending the existing research and developing new representations and techniques.

1.4. General Approach

Automated schedule management systems have been studied widely in both Artificial Intelligence (AI) and Operations Research (OR), and in fact, some work has attempted to compare and integrate the two approaches (Baptiste, *et al.* 1995; Barták 1999; Gomes 2001). Generally speaking, AI algorithms use search and constraint satisfaction processing to find a satisfactory solution as quickly as possible, although they can be extended for optimization. On the other hand, OR algorithms generally use mathematical techniques such as linear programming and mixed integer programming to find or approximate an optimal solution, although they can be used for satisfaction. The approach taken in AI research has generally been to retain rich expressivity by sacrificing efficiency, whereas the approach taken in OR research has generally been to sacrifice expressivity in order to gain efficiency. While I will present several methods for optimizing stability, the main focus of my research is to find satisfactory solutions to large, highly expressive scheduling problems, so I will adopt the AI approach.

Previous work in AI on constraint-based reasoning has tended to focus only on systems of temporal or finite-domain constraints; much less work has investigated the mixture of finite-domain and temporal constraints (e.g., Moffitt, Peintner, and Pollack 2005; Sheini and Sakallah 2005). In addition, while there has been some previous work on dynamic constraint systems, it has generally been restricted to finite-domain constraints (e.g., van Hentenryck and Provost 1991; Schiex and Verfaillie 1993). The goal of this dissertation is to develop representations and techniques that build on this existing body of research to overcome these limitations.

In pursuit of this goal, I will follow the traditional AI approach of exploiting structure to handle complexity. To identify structure within a problem, I will rely on the key insight that all variables are not created equally. Instead, the variables of many scheduling problems can be partitioned into naturally occurring subsets. The variables are unequal in the sense that the constraints over the different subsets are usually unequally tight.

Based on the idea that all variables are not created equally, I will combine and extend three basic ideas common in the AI literature on constraint solving: 1) break the problem apart into smaller subproblems, 2) try to solve the most difficult part of the problem first, and 3) record the results of past work to avoid repeating that work in the future. These are common sense techniques that most people use when solving real life problems and can be applied in specific ways to improve algorithms for solving scheduling problems. As I describe the algorithms and techniques that apply to my research, I will explain how they relate back to these basic ideas.

1.5. Contributions

The main contributions of this dissertation will be sets of techniques that can improve performance when solving complex scheduling problems with hybrid or dynamic constraints in a wide variety of situations. I will briefly lay out these techniques here.

1.5.1. Dynamic Scheduling Problems

When considering scheduling problems that change over time, I will build on existing research on a related problem known as the Dynamic Constraint Satisfaction Problem (DCSP) (Dechter and Dechter 1988). The DCSP represents a sequence of finite-domain constraint satisfaction problems in which each problem is a modification of its predecessor. I will extend this representation by defining the Dynamic Disjunctive

Temporal Problem (DDTP) to represent scheduling problems with dynamic constraints. I will adapt the two techniques most commonly applied to improve performance when solving DCSPs, nogood recording and oracles, and apply them to DDTPs. I will show that oracles may be interpreted in different ways when applied to DDTPs, and I will compare two such interpretations, meta-value oracles and temporal bounds oracles.

In addition to demonstrating that nogood recording and oracles are both effective when applied to DDTPs, I will also describe a large set of problems for which these two techniques are deficient. Specifically, nogood recording can be applied whether the previous problem was consistent or not, while oracles can only be applied when the previous problem was consistent. I introduce an analogue to oracles that I call justification testing, which applies to a problem in the sequence that follows an inconsistent one. I will demonstrate that justification testing can be far more effective than nogood recording in this situation.

As pointed out earlier, I will compare the performance of dynamic techniques along two dimensions—efficiency and stability. I will present a family of local and global temporal stability metrics that apply across a variety of application domains. By quantifying temporal stability in this manner, I will compare dynamic techniques based on the similarity between the subsequent solution schedules that they produce. I will show that some of these temporal stability metrics have the special property that they are monotonically non-decreasing with the depth of the search tree, allowing the use of several well-known search optimization techniques to improve stability. Using an optimization algorithm, I will compare dynamic techniques based on the time required to find the solution that is as similar as possible to the solution of the previous problem.

The combination of nogood recording, oracles, and justification testing, along with a family of temporal stability metrics, will provide a means of improving efficiency and stability when solving dynamic scheduling problems in a variety of situations and application domains.

1.5.2. Hybrid Scheduling Problems

To handle scheduling problems that contain a mixture of finite-domain and temporal variables and constraints, I define the Hybrid Scheduling Problem (HSP). The HSP representation is similar to representations from similar research on mixed variable types in the field of formal verification known as Satisfiability Modulo Theory (SMT). I will describe how state-of-the-art SMT solvers have ignored the distinction between variable types in order to simplify the standard search algorithm. Rather than ignoring this structure, I will exploit it to modify the meta-level variable ordering heuristic and improve efficiency.

I will show how an HSP can be quickly broken apart into a finite-domain and a temporal subproblem, and I will show how the problem space can be organized according to the relative tightness of the constraints on those subproblems. I will describe how solving the subproblems before the complete HSP can save time by quickly detecting any local inconsistencies. When solving the complete HSP, I will show that focusing the search of all problems in a problem set on one type of variable before the other can improve efficiency further.

While focusing search on one subset of variables across all problems in a domain improves efficiency on average, it can actually hurt performance for some individual problems. To address this issue, I will present a method for identifying which type of variable is most constrained, so the search of each individual HSP can be focused on the most-constrained subset of variables. I will introduce the concept of solution density (based on work in distributed CSPs) to measure the tightness of each type of constraint, and I will present a simple method for estimating solution density while solving the subproblems of an HSP. I will then show how a classifier based on linear regressions of estimated solution densities can intelligently choose the most-constrained subset of each individual problem. The combination of detecting local inconsistencies, focusing search, and choosing the most-constrained subset can improve performance across the problem space of hybrid scheduling problems.

1.5.3. Dynamic Hybrid Scheduling Problems

Finally, I will begin an analysis of the relationship between dynamic and hybrid techniques, considering how they can be combined. First, I will cast hybrid problems as dynamic problems in which the complete HSP is a modification of each of its subproblems. I will demonstrate that recording subproblem nogoods can improve efficiency when combined with any of the other hybrid techniques, and I will discuss how recording subproblem oracles could do the same.

Because many real-world applications contain both dynamic and hybrid variables and constraints, I will then define the Dynamic Hybrid Scheduling Problem (DHSP). I will examine the structure of a dynamic sequence of HSPs and their subproblems, and I will propose some ideas for exploiting this structure. Finally, I will provide some thoughts on combining previous research on finite-domain stability with my own research on temporal stability to define metrics of hybrid stability.

1.6. Test Data

As I present the various techniques throughout this dissertation, I will compare each of them empirically based on randomly generated scheduling problems. Randomly generated problems offer several advantages. First of all, by carefully controlling key parameters, it is possible to generate a wide variety of problems that cover a broad portion of the problem space. Second, randomly generated problems can be created quickly, so it is possible to collect a large number of problems to provide meaningful results.

The drawback of randomly generated problems is that they may not contain the structure that real-world problems do. It would therefore help to compare these techniques using scheduling problems from a real-world application domain. An application domain that is adequate for my needs must fit the following criteria:

1) **Hybrid.** The scheduling problems must contain both finite-domain and temporal variables that interact with one another, making it possible to compare hybrid techniques.
2) **Dynamic.** The constraints of the scheduling problems must change over time, making it possible to compare dynamic techniques.
3) **Tightly constrained.** The constraints of the scheduling problems must be tight enough to cause a significant amount of backtracking during search, so the different techniques will have a chance to affect performance.

4) **Electronically recorded.** The constraints of the problems must be recorded in electronic format, so they can be automatically converted into a format that is readable by a solver.

Criteria (1) and (2) are easy to satisfy—many application domains contain both hybrid and dynamic variables and constraints, as was pointed out earlier. Similarly, many application domains are sufficiently complex to meet criterion (3).

Electronically recording the constraints as in criterion (4) is a different matter. Even though many computerized scheduling systems are in use for a variety of application domains, these systems generally only record solutions, not constraints. For example, a user might add a meeting to his or her schedule using a personal calendar system. To add the meeting, the user generally does not input constraints such as who must attend, where it can take place, and all of the possible times. Instead, the user considers all of these constraints, solves the scheduling problem, and then inputs the solution as a meeting with a fixed start time, end time, and location. Still, there are several scheduling domains in which an automated system has been developed to perform scheduling tasks, and some of these systems do record constraints, not just solutions.

The issue is complicated, however, by the fact that criteria (3) and (4) are often at odds with each other. Usually, if an application domain is complex enough to meet criterion (3), the problems are too difficult for an automated system to solve in a reasonable amount of time. Instead, these problems are given to a human to solve, and so the constraints are not recorded electronically. The situation is somewhat of a Catch 22—complex constraints are not recorded electronically because an adequate automated solver does not exist, and it is difficult to develop an adequate solver because complex constraints are not recorded electronically.

Although I was not able to find an application domain to avoid this problem entirely, I was able to find one that fit three of the four criteria—air traffic control. I will describe this domain in detail in Section 3.2, so I will provide a very brief overview here. As an aircraft approaches an airport, a human air traffic controller is responsible for assigning it a runway and scheduling its actual arrival time, so this domain meets criterion (1). New aircraft entering the airspace, weather conditions, and unexpected runway closures can all force changes to the constraints on the schedule, so this domain meets criterion (2). To reduce the workload on air traffic controllers, the Federal Aviation Administration (FAA) teamed up with the National Aeronautics and Space Administration (NASA) in the 1990s to develop the Center-TRACON Automation System (CTAS). CTAS automates a majority of the scheduling tasks performed by air traffic controllers, and it is capable of recording the constraints as they change over time.

Unfortunately, the air traffic control domain does not satisfy criterion (3)—the problems are not difficult enough to cause significant backtracking. In fact, this is exactly the reason that the FAA and NASA were able to develop an automated scheduling system. Based on the types of constraints, the problems in this domain are actually NP-complete. However, because the actual problem instances are typically very under-constrained, CTAS actually applies a polynomial-time algorithm that can find a solution the vast majority of the time. If this algorithm fails to find a solution, CTAS can notify the air traffic controllers to solve the problem manually, temporarily retracting optional constraints meant to improve traffic flow if necessary. Hence, the fact that the problem instances are under-constrained is a double-edged sword with respect to our goal here—on one hand, it was possible to develop a solver and record the constraints electronically, but on the other hand, even though the problem domain is complex, the individual problem instances are generally too simple to test the techniques in this dissertation.

I will overcome this deficiency in the air traffic control domain by adding randomly generated constraints to the problem instances. As explained further below, even though these constraints are random, their structure is not, so they do not contradict the original reason for testing with real-world problems in the first place. By adding more constraints to the problems, I can tighten them enough to satisfy criterion (3). I will describe exactly how I modify the original air traffic control problems as I compare the techniques presented throughout this dissertation.

1.7. Dissertation Overview

In the next chapter, I will present the basic concepts and existing research relevant to the work in this dissertation. In Chapter 3, I will present representations and techniques for improving efficiency and stability when solving scheduling problems with dynamic constraints. In Chapter 4, I will present methods for improving efficiency when solving scheduling problems with hybrid constraints. Then in Chapter 5, I will present several representations and ideas for combining and extending the dynamic and hybrid techniques presented in Chapters 3 and 4. Finally, in Chapter 6, I will conclude by summarizing the major contributions of this dissertation and presenting ideas for future work.

Chapter 2

Background

2.1. Introduction

To properly address the problem of scheduling with dynamic and hybrid constraints, it is necessary to understand the relevant research in the field. In Chapter 1, I stated that I will apply representations and techniques from the field of AI to solve these complex scheduling problems. In this chapter, I review the major concepts of AI on which my research is built.

The Constraint Satisfaction Problem (CSP) is a very common problem representation in the AI literature. It is useful for describing problems in which a set of decisions must be made, but because these decisions can interact with one another, not all combinations of choices are valid. In the office scheduling example, the decisions that need to be made include who will attend the meeting, where it will take place, and when it will start. The problem is to figure out whether or not there exists any combination of choices that is valid. In a CSP, each decision to be made is called a *variable*, each possible choice for each decision is called a *value*, the interactions between decisions are called *constraints*, and combinations of choices are called *assignments*.

Definition 2.1. A *Constraint Satisfaction Problem (CSP)* (Montanari 1974) is a triple ⟨*V*,

D, $C\rangle$, where V is a set of variables, D is a set of domains, where each domain is a set of possible values for a variable of V, and C is a set of constraints.

Definition 2.2. A *constraint* $c \in C$ of CSP $\langle V, D, C\rangle$ is a relation over a subset of the variables in V, called the *scope* of c. A partial assignment over the scope of c is called a *tuple*. Any tuple that is in this relation *satisfies* the constraint, and any tuple that is not in this relation *violates* the constraint.

Definition 2.3. An *assignment* is a conjunction of bindings. A *binding* $v \leftarrow x$ means that the variable v has been bound to the value x, where x is in the domain of v. An assignment is *complete* for a CSP $\langle V, D, C\rangle$ if it contains one binding for every variable of V, and it is *partial* otherwise.

Definition 2.4. A set of constraints C over variables V with domains D is *consistent* iff there exists at least one assignment over all of the variables of V that does not violate any constraint of C; otherwise, it is *inconsistent*. Given a CSP $\langle V, D, C\rangle$, we say that an assignment A over V is consistent with C iff it does not violate any constraint of C, and we say that the CSP is consistent iff there exists a consistent complete assignment over V.

In practice, it is often the case that merely deciding whether or not a CSP is consistent is not sufficient—the actual goal is to find a consistent complete assignment if one exists. This assignment is called a *solution*.

Definition 2.5. A *solution* to a CSP $\langle V, D, C\rangle$ is a complete assignment of values to the variables of V that satisfies all of the constraints of C. In other words, CSP is consistent iff it has at least one solution.

If a CSP is inconsistent, then it has no solution. In this case, an explanation of the inconsistency is often desired. This explanation can be given in the form of a *justification*.

Definition 2.6. A *justification* is a minimal subset of variables whose constraints cause an inconsistency.

Stated another way, if J is a justification of the inconsistency of a CSP $\langle V, D, C\rangle$, and if C' is the subset of constraints in C over the variables in J, then $\langle J, D, C'\rangle$ is an inconsistent subproblem of $\langle V, D, C\rangle$. At least one of the constraints on J must be changed in order to make the CSP consistent. Producing a justification for an inconsistent CSP is important when solving problems that change over time, particularly in an interactive setting.

Several different relationships might exist between the constraints of a CSP. To understand some of the techniques that are used to solve CSPs, it will be helpful to define some of these relationships.

Sometimes it is possible to infer the existence of a new constraint by reasoning about a set of known constraints. This new constraint is called an *induced* constraint.

Definition 2.7. A constraint c is *induced* by a set of constraints C iff every assignment that violates c also violates at least one constraint of C. Stated another way, a constraint c is induced by a set of constraints C iff every assignment that satisfies every constraint of C also satisfies c.

Finally, an important related concept is the *complement* of a constraint.

Definition 2.8. The *complement* of a constraint c, written $\neg c$, is the complement of the relation that c represents. Thus, any partial assignment that is consistent with c violates $\neg c$, and any partial assignment that violates c is consistent with $\neg c$. This means that a constraint $c \in C$ is violated iff $\neg c$ is in or is induced by C.

A variety of algorithms have been developed to solve CSPs. These algorithms can be separated into three major categories—inference, local search, and backtracking search.

Inference is the process of deriving induced constraints. In most cases, the consistency of a CSP can be determined through inference alone, but (depending on the type of CSP) this process may require time and space that is exponential in the number of variables. Many CSP solvers infer induced constraints to improve the efficiency of one of the search methods described below. Recall from Definition 2.5 that a solution to a CSP $\langle V, D, C\rangle$ is a complete assignment that satisfies all of the constraints of C. Since each solution must satisfy every constraint of C, it must also satisfy any induced constraint c. This means that adding induced constraints to a CSP via inference does not add or remove any solutions, so it can improve efficiency without sacrificing soundness or completeness.

The local search method begins with a complete (but not necessarily consistent) assignment. If the assignment is inconsistent, one or more of the bindings are changed in order to repair the inconsistencies. In general, the heuristics of a local search algorithm are designed to repair as many inconsistencies as possible and rely on some form of randomness to escape from local minima. This form of search is not systematic, however, so it is inherently incomplete and is difficult to combine with the strong pruning techniques of systematic search.

Backtracking search is a systematic depth-first search through a tree of partial assignments. It works by repeatedly selecting an unassigned variable, assigning a value to it, and checking the partial assignment against the constraints until a solution is found. If a branch of the search tree represents an inconsistent partial assignment, then that branch is called a *dead end*. When a dead end is encountered, the algorithm backtracks by unbinding the most recently bound variable and testing a different value. If the algorithm runs out of variables to backtrack over before it finds a solution, then it has proven that no solution exists. Backtracking search is only complete for CSPs in which the domains of the variables are limited to finite sets. The backtracking search algorithm will be described in detail in the next section.

So far, I have described the CSP in very general terms. The CSP, however, can be broken down into various classes. Each class is defined by the limitations imposed on the domains of the variables and the types of constraints that are allowed. These limitations restrict the set of problems that each class can represent, but allow for improved efficiency through the use of special-purpose reasoning techniques. I will describe three specific classes of constraint satisfaction problems—finite-domain constraint satisfaction, Boolean satisfiability, and temporal constraint satisfaction.

2.2. Finite-Domain Constraint Satisfaction

A finite-domain CSP is a CSP in which the domains of the variables are limited to finite sets. This means that each constraint can be described with a finite set of tuples over the constraint's scope that satisfy it. A finite-domain CSP is a natural representation for the finite-domain constraints of the problems I will consider in my research. For example, the location of a meeting can be represented as a finite-domain CSP variable, where the domain of the variable is the set of available locations. If two meetings are scheduled at the same time, then a finite-domain constraint can represent the fact that the meetings must not be assigned to the same location.

2.2.1. Backtracking Search

Determining whether or not a satisfying assignment exists for a given finite-domain CSP is an NP-complete problem. Inference is one method for solving a finite-domain CSP. Recall that a CSP is inconsistent iff there exists a constraint c such that the constraints of the CSP include or induce both c and $\neg c$. Inference can prove that a finite-domain CSP is inconsistent by finding such a constraint, or it can prove that a finite-domain constraint is consistent by proving that no such constraint exists. Solving a finite-domain CSP through inference alone, however, may require time and space exponential in the number of variables (Cooper 1990). One of the most common approaches to solving a finite-domain CSP is therefore to limit inference until it is tractable—but incomplete—and combine it with backtracking search. A finite-domain CSP ⟨*V*, *D*, *C*⟩ can be solved by calling the procedure **Backtrack-Search**(*V*, *D*, *C*, ∅), which is shown in Figure 2.1. The working domain (line 4) of a variable is the subset of values from the variable's original domain that have not been pruned yet.

```
Backtrack-Search(V, D, C, A)
// V is the set of unassigned variables
// D is the working domains
// C is the set of constraints
// A is the current partial assignment
1    if V = ∅, return the solution assignment A
2    select an unassigned variable x from V
3    V ← V – {x}
4    while the working domain of x, D(x), is not empty
5        select a value v from D(x)
6        D(x) ← D(x) – {v}
7        A' ← A ∪ {x←v}
8        if A' does not violate any constraint of C then
9            sol ← Backtrack-Search(V, D, C, A')
10           if sol ≠ FAIL, return the solution assignment sol
11   return FAIL
```

Figure 2.1. The **Backtrack-Search** algorithm.

Backtracking search is only a complete algorithm for solving CSPs with finite domains. If the domains were not limited to finite sets, backtracking search would be incomplete because it would be impossible to test every individual value of an infinite domain. Like inference, the worst case time requirements of backtracking search are exponential in the number of variables; in the best case, it is possible for backtracking search to find a solution on its first try without ever backtracking, but this is not guaranteed. The advantage of backtracking search is that its worst case space requirements are linear in the number of variables and domain sizes, whereas inference could require space that is exponential in the number of variables.

2.2.2. Efficiency Techniques

Many techniques have been developed to improve the efficiency of the basic backtracking search algorithm. I will briefly review three types of these techniques—look-ahead, look-back, and nogood recording. Many other techniques have been developed besides these, and will be utilized as appropriate in the thesis. For more information on efficiency techniques for finite-domain CSPs, see (Kumar 1992) and (Dechter 2003).

Look-ahead techniques use inference to prune values from the domains of unassigned variables. Forward checking, a simple form of look-ahead, scans through the domain of each unassigned variable and removes any value that, if assigned to its variable, would cause a constraint violation. If all of the values in the domain of an unassigned variable are pruned, then the search algorithm can backtrack before a constraint has been violated. Stronger forms of look-ahead include arc-consistency and path-consistency (Mackworth 1977).

Look-back techniques keep track of conflicts in order to backtrack over multiple variable bindings to the point at which the conflict occurred. Conflict-directed backjumping (Prosser 1993), for example, maintains a conflict set for each variable *x* that records the set of assigned variables whose bindings caused a value to be pruned from the domain of *x*. When a dead end is encountered, CBJ allows the backtracking search algorithm to backtrack over multiple variable bindings without sacrificing completeness. A stronger form of look-back is Ginsberg's (1993) dynamic backtracking.

Finally, nogood recording (NGR) (Scheix and Verfaillie 1993) is another technique that has been shown to improve efficiency when solving finite-domain CSPs. Although it is not necessary to understand NGR in the current context of static CSPs, it will be important to understand this technique later on when it is used in the context of dynamic CSPs, which change over time. When a CSP solver encounters a dead end and has to backtrack, it is possible that it will encounter another dead end later on in the search for exactly the same reason. A dead end in the search tree represents an induced constraint that was violated because it was not listed explicitly. To avoid wasting time searching a dead-end sub-tree, the solver can list the induced constraint explicitly in the form of a nogood.

Definition 2.9. A *nogood* of CSP $\langle V, D, C\rangle$ is a pair $\langle A, J\rangle$. A is an assignment to a subset of the variables of V, and J is a subset of V whose constraints prevent A from participating in any solution. That is, J is the justification of the nogood.

As a CSP solver searches for a solution, it records a nogood each time it finds a dead end. As search continues, the CSP solver checks each partial assignment it considers against each nogood it has recorded. If it finds a nogood $\langle A, J\rangle$ such that A is a subset of the partial assignment currently being considered, then that partial assignment can be pruned immediately. In other words, NGR records induced constraints in order to avoid violating them in the future. Because there could be an exponential number of induced constraints, solvers that use NGR generally include a mechanism for discarding nogoods as new ones are recorded. These mechanisms usually discard nogoods with larger assignments because these larger nogoods require more space and match fewer partial assignments than the smaller ones do. Despite the overhead required to record, match, and discard nogoods, the increased pruning made possible by NGR allows this technique to significantly improve the efficiency of backtracking search (Scheix and Verfaillie 1993; Zhang, *et al*. 2001). NGR is a one example of recording past work to avoid repeating it in the future.

2.2.3. Variable Ordering Heuristics

An important factor in the efficiency of CSP solving is the order in which constrained variables are considered during search. Whenever a dead end is encountered during backtracking search, a subset of complete assignments is pruned from the search space. The number of complete assignments pruned is equal to the product of the domain sizes of each unassigned variable. If an inconsistency in a partial assignment is discovered closer to the root of the search tree, then more of the variables are unassigned, and so a larger subset of complete assignments will be pruned. Pruning large portions of the search space in this manner can improve the efficiency of backtracking search by reducing the amount of time wasted searching branches of the search tree that cannot contain a solution. In other words, efficiency can be improved if inconsistencies are exposed quickly. Because inference is only used in a limited form when combined with backtracking search (to avoid exponential space requirements), it is possible that the current partial assignment cannot be a solution, but it is not possible to infer this fact. In this case, it is up to the variable ordering to expose the failure as quickly as possible.

Studies have shown that the most-constrained variable ordering heuristic (also called "fail-first" or "least-commitment") is effective at quickly exposing failures (Gent, *et al.* 1996). The most-constrained variable ordering heuristic chooses the variable with the fewest values remaining in its working domain (that is, the variable that is most-constrained relative to the previously assigned variables). If the value assigned to the variable leads to a dead end, then there will be relatively few values remaining to be tested. If more than one variable are tied with the fewest values remaining in their working domains, then the tie is broken by examining the constraints affecting each variable and choosing the one that interacts with the other unassigned variables the most (that is, the variable that is most-constrained relative to the unassigned variables). This will help to increase pruning and, therefore, efficiency. Different heuristics use different methods to measure the tightness of a variable's constraints. Nogood recording can improve the accuracy of these measurements by listing some of the induced constraints explicitly, thereby enhancing the effectiveness of the most-constrained variable heuristic.

A special case of the most-constrained variable heuristic is unit propagation—any variable with a single value in its working domain is selected immediately. Since the variable has only one value in its domain, the decision has already been implicitly made to bind the single remaining value to that variable. Once the variable is explicitly bound to its value, a look-ahead technique may prune the domains of unassigned variables even further.

2.2.4. Constraint Optimization

With very little modification, a standard backtracking search algorithm can be converted from a constraint satisfaction algorithm to a constraint optimization algorithm.

In a Constraint Optimization Problem (COP), a preference function ranks assignments against one another. In the office example, Alice might prefer to have the meeting in the conference room of her own building, or she might prefer to start the meeting as early as possible.

Without loss of generality, assume that an assignment with a lower preference value is preferred to an assignment with a higher preference value. As shown in Figure 2.1, the standard backtracking search algorithm stops as soon as it finds a solution. Instead of stopping, the algorithm could continue to search for more solutions, always storing the solution with the lowest preference value. When the algorithm has exhausted the search space, it is guaranteed to have found an optimal solution.

Some preference functions have the special property that they are monotonically non-decreasing with the depth of the search tree. A preference function is monotonically non-decreasing iff for any node *n* in the search tree, the preference values of all nodes in the subtree rooted at *n* are no lower than the preference value at *n*. For example, if each binding incurs a non-negative cost, and if cost is additive over bindings, then this cost function is a monotonically non-decreasing preference function.

A monotonically non-decreasing preference function allows for the use of several additional efficiency techniques. First, the commonly used branch and bound approach compares the preference value at each node of the search tree to the preference value of the best solution found so far. If the preference value at a node is greater than value of the best solution found so far, the algorithm can backtrack because the subtree rooted at that node cannot contain a better solution. Second, valued nogoods (Dago and Verfaillie 1996) record a partial assignment *A* along with a value *v* such that no extension of *A* has a preference value lower than *v*. If the current partial assignment is a superset of *A*, and if the preference value of the best solution found so far is less than *v*, then the solver can backtrack immediately. Both valued nogoods and branch and bound improve efficiency by preventing the solver from searching parts of the search space where it is impossible to find a solution that is any better than the best solution found so far.

2.3. Boolean Satisfiability

The Boolean satisfiability problem (SAT) is a special form of the finite-domain CSP in which each variable is Boolean with the domain {TRUE, FALSE} and the constraints are given as a logical combination of these Boolean variables. Any logical combination of Boolean variables can be converted to conjunctive normal form (CNF), which is a conjunction of clauses, where each *clause* is a disjunction of literals, and each *literal* is either a Boolean variable or the negation of one. It is common practice in the SAT literature to assume that the constraints of a SAT instance are given in CNF as a set of clauses.

Definition 2.10. A *Boolean Satisfiability Problem (SAT)* is a pair $\langle V, C\rangle$, where V is a set of Boolean variables, and C is a set of clauses over the variables of V.

Any finite-domain CSP (even those with variable domains of more than two values) can be transformed into an equivalent instance of SAT in polynomial time. This can be accomplished by creating one Boolean variable for each value in the domain of each variable in the original CSP. For example, suppose the original CSP contains a variable x with the domain {A, B, C}. First, create one Boolean variable for each value in the domain; call them x_A, x_B, and x_C. For each Boolean variable x_i, $x_i \leftarrow$TRUE in the SAT transformation represents $x \leftarrow$i in the original CSP.

A set of clauses in the SAT transformation can force exactly one value to be assigned to each variable of the original CSP. In this example, this clause requires at least one of the values to be assigned:

$$x_A \lor x_B \lor x_C$$

It is possible to prevent more than one of the values from being assigned by adding one clause for each pair of values:

$$\neg x_A \lor \neg x_B$$

$$\neg x_A \lor \neg x_C$$

$$\neg x_B \lor \neg x_C$$

Finally, it is possible to encode the original CSP constraints as clauses over the Boolean variables. Suppose that the original CSP contains a second variable y with the domain {A, B, C} and the constraint that $\{x \leftarrow \text{A}, y \leftarrow \text{B}\}$ is an illegal assignment. This can be represented in the SAT transformation with a clause that forces any solution to avoid at least one of these variable assignments:

$$\neg x_A \lor \neg y_B$$

Altogether, this transformation is linear in the size of the constraints of the original CSP, and quadratic in the size of the largest variable domain. This transformation shows, for example, that the decisions in the office example as to where the meeting should take place and who should attend can also be represented with Boolean variables and constraints.

As with the general form of the finite-domain CSP, an instance of SAT can be solved with backtracking search and the efficiency techniques described in the previous section. The special form of the SAT representation, however, allows for several other techniques that can improve performance even further. Among these techniques are the 2-watched literals scheme, which speeds up the forward-checking process by quickly identifying clauses for unit propagation, and the Variable State Independent Decaying Sums (VSIDS) variable ordering heuristic, which employs a clever counting scheme to select variables that appear in the most nogood clauses (this is another example of the most-constrained variable heuristic). Both of these techniques were introduced in the Chaff SAT solver (Moskewicz, *et al*. 2001), which marked a significant step forward in the efficiency of SAT solvers. For the purposes of this dissertation, it is not necessary for the reader to understand the details of these techniques. I mention them here because they represent the state of the art in SAT solving, and as such I have incorporated them into my implementations as described in later chapters.

2.4. Temporal Constraint Satisfaction

Temporal CSPs are designed specifically to describe scheduling problems. Each variable of a temporal CSP represents a timepoint—a marker that indicates a particular event (e.g., the start or end of an interval, or a change in the environment). The domains of a temporal CSP are therefore limited to numerical values in order to represent times on a timeline.

Consider Evan's schedule in the office example. He plans to meet with Donna for one hour at 11:00 to prepare the presentation. He then needs to leave his office between 12:30 and 12:45 to make it to the team meeting by 1:30. A temporal variable can represent the start or end of each activity: P_S and P_E are the start and end of the presentation preparation, T_S and T_E are the start and end of traveling, and M_S and M_E are the start and end of the team meeting. I will show examples of temporal constraints in the following subsections.

One way to describe a scheduling problem with a CSP is to discretize and bound the domain of each variable in order to represent and solve the problem as a finite-domain CSP. For example, if events must be scheduled within a single day, and if each event can be limited to occur at a particular minute of the day, then the domain of each variable can be limited to the finite set of the integers in [0, 1440]. This approach, however, has several drawbacks. With so many values in the domain of each variable, backtracking search can be very inefficient when solving such a problem. The size of the domains can be reduced by increasing the granularity from every minute to, for example, every 10 minutes. While restricting the domains in this manner improves efficiency, it also has the potential to eliminate solutions, which makes the backtracking search algorithm incomplete.

Rather than limiting the variable domains, several types of temporal CSPs limit the types of constraints that can be expressed in the problem. While limiting the constraints makes it impossible to represent some problems, several types of temporal CSPs are able to represent many interesting scheduling problems while allowing for specialized inference techniques to improve efficiency. I will describe two types of temporal CSPs that are commonly used to represent scheduling problems—Simple Temporal Problems and Disjunctive Temporal Problems.

2.4.1. Simple Temporal Problems

In the Simple Temporal Problem, each constraint is limited to the form of a temporal difference constraint.

Definition 2.11. A *temporal difference constraint (TD constraint)* is a linear inequality of the form ($x - y \leq b$), where x and y represent time points and $b \in \Re$. Such a constraint is interpreted as "x follows y by no more than b units of time." A TD constraint is *satisfied* iff its time points are assigned times that are consistent with the inequality; it is *violated* otherwise.

Evan's schedule can be represented with a set of TD constraints. To do this, it is necessary to introduce one more temporal variable, *TR*, which is the temporal reference that will mark 12:00am at the beginning of the day. Evan's schedule can be represented by the TD constraints in Table 2.1.

A set of TD constraints, such as those listed in Table 2.1, define a Simple Temporal Problem.

Definition 2.12. A *Simple Temporal Problem (STP)* (Dechter, Meiri, and Pearl 1991) is a pair $\langle V, C\rangle$, where V is a set of timepoints and C is a conjunction of TD constraints over the time points of V. A *solution* to an STP $\langle V, C\rangle$ is an assignment of times to all timepoints of V such that all of the TD constraints of C are satisfied. An STP is *consistent* iff it has at least one solution.

Constraints of Schedule	**TD Constraints**
preparation takes 1 hour	$P_E - P_S \leq 60$, $P_S - P_E \leq -60$
traveling takes 30 minutes	$T_E - T_S \leq 30$, $T_S - T_E \leq -30$
meeting takes 1 hour	$M_E - M_S \leq 60$, $M_S - M_E \leq -60$
preparation starts at 11:00am	$P_S - TR \leq 660$, $TR - P_S \leq -660$
traveling starts between 12:30pm and 12:45pm	$T_S - TR \leq 765$, $TR - T_S \leq -750$
meeting starts at 1:30pm	$M_S - TR \leq 810$, $TR - M_S \leq -810$

Table 2.1. TD constraints representing Evan's schedule.

The general backtracking search procedure only applies to finite-domain CSPs. Since the variable domains of temporal CSPs are infinite, the number of possible assignments is infinite, so it is necessary to employ a special set of techniques to solve them.

To test the consistency of an STP, the common approach is to represent the STP as a distance graph and check this graph for negative cycles. The distance graph of an STP $\langle V, C\rangle$ is a graph $\langle V', E\rangle$ such that each vertex of V' represents a time point of V and each edge of E represents a TD constraint of C. A TD constraint $c = (x - y \leq b)$ is represented by an edge e from y to x with weight b. A negative cycle in the distance graph means that, for some timepoint x, the STP induces the constraint $(x - x < 0)$, which is impossible to satisfy.

With the distance graph representation, the consistency of an STP can be determined in time polynomial in $|V|$ with inference using an all-pairs shortest-path (APSP) algorithm, such as Floyd-Warshall or Bellman-Ford, which run in $O(|V|^3)$ time (Cormen, Leiserson, and Rivest 1990). The result of an APSP algorithm is the transitive closure of the TD constraints, called the *d-graph*. An edge from x to y with weight w in the d-graph implies that w is the weight of the shortest path from x to y in the distance graph. In other words, the APSP algorithm finds all of the constraints that are induced by the distance graph and represents them explicitly in the d-graph. If the STP is consistent, then a solution to the STP can be taken from the d-graph in time linear in $|V|$.

2.4.2. Disjunctive Temporal Problems

For many scheduling problems, an STP is not able to represent all of the necessary constraint types. For example, assume that Bob has not figured out the schedule yet, but he knows that Evan is unavailable after 3:30pm because of another meeting, *M2*. An STP can represent the constraint that the team meeting *M* must precede *M2*, or it can represent the constraint that *M2* must precede *M*, but it cannot represent the fact that either ordering is acceptable. To represent this type of choice, a temporal constraint must be able to handle disjunctions.

Definition 2.13. A *disjunctive temporal constraint* is a disjunction of TD constraints. A disjunctive temporal constraint is *satisfied* iff at least one of its component TD constraints is satisfied; otherwise, it is *violated*.

The disjunctive temporal constraint $(M_E - M2_S \leq 0) \vee (M2_E - M_S \leq 0)$ can represent the fact that Evan's 3:30 meeting cannot overlap with the team meeting.

If the constraints of a temporal CSP are limited to disjunctive temporal constraints, then the resulting problem is called a Disjunctive Temporal Problem.

Definition 2.14. A *Disjunctive Temporal Problem (DTP)* (Stergiou and Koubarakis 1998) is a pair $\langle V, C\rangle$, where V is a set of time points and C is a conjunction of disjunctive temporal constraints over the timepoints of V. An *exact solution* to a DTP $\langle V, C\rangle$ is an assignment of times to all timepoints of V such that all constraints of C are satisfied. A DTP is *consistent* iff it has at least one exact solution; otherwise, it is *inconsistent*.

An STP is a conjunction of TD constraints, whereas a DTP is a conjunction of disjunctions of TD constraints—that is, a DTP is in conjunctive normal form (CNF). Since it is possible to negate a TD constraint, any combination of conjunctions, disjunctions, and negations of TD constraints can be converted into a DTP. Hence, a DTP can represent many problems that an STP cannot. The cost of this expressivity is that the DTP is an NP-complete problem (Dechter, Meiri, and Pearl 1991).

The search space of exact solutions of a DTP is infinite, so all DTP solvers developed to date search the space of component STPs.

Definition 2.15. A *component STP* of a DTP $\langle V, C\rangle$ is an STP $\langle V, C'\rangle$, where C' is the conjunction of one TD constraint from each disjunctive temporal constraint of (a subset of) the constraints of C. If a component STP includes one TD constraint from every disjunctive temporal constraint of C, then it is *complete*; otherwise, it is *partial*. If there exists a complete consistent component STP *S* of a DTP *D*, then *S* is a *solution* of *D*, and a solution of *S* (which can be taken from the d-graph of *S* in polynomial time) is an exact solution of *D*.

It is often preferable for a DTP solver to produce a solution STP rather than an exact solution. A solution STP represents a set of exact solutions, so it offers greater flexibility. Given a solution STP, an agent can extract an arbitrary exact solution in polynomial time, or it may perform additional computation to select an exact solution that best suits its purposes. Alternatively, the agent may employ a more sophisticated method of STP dispatch that never requires it to narrow its schedule to a single exact solution (Muscettola, Morris, and Tsamardinos 1998; Tsamardinos, Morris, and Muscettola 1998). Given these options, a solution STP offers greater flexibility than does an exact solution.

To search the space of component STPs, most DTP solvers transform the problem into either a finite-domain CSP or a Boolean SAT meta-level problem. Due in part to the recent advances in SAT solvers described in Section 2.3, the fastest state of the art solvers that can solve DTPs (Nieuwenhuis and Oliveras 2005; Dutertre and de Moura 2006) use the SAT transformation.

The process of creating the meta-level SAT representation of a DTP is straightforward. For each TD constraint of each disjunctive temporal constraint in the DTP, create a new Boolean variable in the meta-level SAT problem to represent it. To create the clauses of the SAT problem, replace each TD constraint of the DTP with the Boolean variable that represents it. An assignment of values to variables in the meta-level problem corresponds to a component STP of the DTP. The temporal bounds imposed by the TD constraints are not represented in the meta-level problem, so a meta-level assignment represents a solution to the DTP iff it satisfies the constraints of both the meta-level problem and the corresponding component STP.

The advantage of a meta-level transformation is that the search process can then be controlled by a well-known, efficient finite-domain search algorithm, such as backtracking search. Additionally, any efficiency techniques developed for finite-domain search algorithms can be applied to the meta-level problem. During backtracking search, most DTP solvers periodically test the TD constraints that correspond to the current meta-level assignment. Using forward checking, if a value of an unassigned meta-level variable is found that would create a negative cycle, then that value can be pruned. If the TD constraints contain a negative cycle, then the search will backtrack, and the meta-level variable assignments that correspond to the edges of the negative cycle can be recorded as a nogood.

2.5. Hybrid Constraints

Finite-domain CSPs are capable of representing finite-domain constraints, and temporal CSPs are capable of representing temporal constraints, but neither is capable of representing both effectively. The scheduling problems that I am considering in my research contain both finite-domain and temporal variables that interact with each other, so I will review existing representations that can express hybrid constraints—constraints that express a relationship between finite-domain and temporal variables.

In the office example, the team meeting *M* could not overlap with Evan's 3:30 meeting, *M2*. In the previous section, I showed how this fact could be represented with the disjunctive temporal constraint $(M_E - M2_S \leq 0) \vee (M2_E - M_S \leq 0)$. This constraint, however, only needs to be enforced if Evan is attending the team meeting. We can represent which programmer is attending the team meeting with a finite-domain variable A with domain {DONNA, EVAN}. A hybrid constraint can represent the fact that whether or not the temporal constraint needs to be enforced depends on the finite-domain assignment: $A \neq \text{EVAN} \vee (M_E - M2_S \leq 0) \vee (M2_E - M_S \leq 0)$.

Several previous representations and algorithms have been developed to handle hybrid constraints. Bockmayr and Kasper (1998) present a general framework for representing both finite-domain constraints and Integer Linear Programming (ILP) constraints, but they say little about how the two forms of constraints actually interact. Strichman, Seshia, and Bryant (2002) present a method that builds a SAT problem that is equivalent to the original hybrid problem by introducing a clause for each transitive relation among the numerical constraints, but this can require an exponential number of new clauses. I will discuss two recent systems that are designed to deal with hybrid constraints: Hybrilitis and Ario.

2.5.1. Hybrilitis

Hybrilitis (Moffitt, Peintner, and Pollack 2005) is one algorithm that was developed recently to handle scheduling problems with certain types of hybrid constraints. The developers of Hybrilitis refer to the type of problem that it can solve as a DTP_{FD} (a DTP with finite-domain constraints). In this framework, each variable x consists of a finite-domain component x_F and a temporal component x_T. The temporal constraints of a DTP_{FD} are similar to those of a DTP, except that in a TD constraint $(x_T -$

$y_T \leq b$), the bound b can be either a constant (as it is in a standard DTP constraint) or a function of the assignments to x_F and y_F.

This representation is useful when, for example, the time between two meetings depends on the locations of those meetings because a participant of both meetings must travel from one location to the other; however, it is not expressive enough to represent all of the types of hybrid constraints in the scheduling problems that I will consider in my research. For example, a meeting might include any combination of three presenters. If each presenter needs 15 minutes to speak, then the duration of the meeting depends on the assignment of three finite-domain variables, but Hybrilitis can only represent a duration that depends on the assignment of at most two finite-domain variables. While it would be theoretically possible to increase the number of finite-domain variables on which a duration depends, each additional finite-domain variable would add another dimension to the bounds matrix. This means that the size of the bounds matrix grows exponentially in the number of finite-domain variables involved in the constraint.

The authors tested three variations of the Hybrilitis algorithm. The "brute-force" approach assigns the finite-domain component of each of the variables before searching for a solution in the induced DTP. The "least-commitment" approach solves a relaxed DTP (in which the bounds matrix of each TD constraint is replaced by the maximum value from the matrix) before assigning any of the finite-domain components. Finally, the "least-commitment with forward checking" approach augments the previous approach with a propagation method that can prune values from the domains of the finite-domain components as it solves the relaxed DTP. The authors report that on a randomly generated set of test problems, the least-commitment with forward checking approach was the most efficient, while the brute-force approach was the least efficient. Least-commitment with forward checking was an order of magnitude faster than the brute-force approach.

2.5.2. Ario

Ario (Sheini and Sakallah 2005) is another recently developed algorithm that can handle both finite-domain and temporal constraints. Ario is a Satisfiability Modulo Theory (SMT) solver. SMT was developed in the field of formal verification, and it represents a class of problems that contain different mixtures of constraint types. Although faster and more expressive SMT solvers have been developed (Nieuwenhuis and Oliveras 2005; Dutertre and de Moura 2006), I will describe Ario because the class of problems that it solves is the closest to the hybrid problems that I am considering in my research. The basic algorithm behind Ario is the same basic algorithm used by the faster SMT solvers; the difference is that the faster solvers employ more sophisticated special-purpose constraint propagation techniques.

Ario is designed to solve Mixed Logical/Integer Linear Programming (MLILP) problems. An MLILP is a Boolean combination of literals, where a literal is either an atom or its negation, and an atom is either a Boolean variable or a Unit-Two-Variable-Per-Inequality (UTVPI) constraint. A UTVPI constraint is a constraint of the form $ax + by \leq d$, where $a, b \in \{-1, 0\}$ and $d \in \Re$. A UTVPI constraint is similar to a TD constraint, but slightly more expressive, in that TD constraints can only bound the difference between two timepoints, while a UTVPI constraint can also bound their sum.

In Ario, the UTVPI constraints of the MLILP are replaced by Boolean indicator variables—if a Boolean indicator variable B represents a UTVPI constraint c, and if B is assigned to TRUE, then it implies that c is active. In other words, Ario transforms the problem into a meta-level SAT representation. In Ario, the space of Boolean assignments is searched with an efficient SAT solver called Pueblo. To test the consistency of the active UTVPI constraints that are implied by the Boolean assignment, Ario employs a specialized solver that computes the transitive closure of the linear inequalities, much like the APSP algorithms described in Section 2.4.1.

The developers of Ario compared it against two other solvers on a set of combinational circuit benchmarks, and it performed very well, generally one to two orders of magnitude better than the other solvers, including Xpress-MP (an OR tool) and the Stanford Validity Checker (a verification tool). Although Ario is able to represent both finite-domain and temporal constraints as well as their interactions, the algorithm treats the different types of constraints equally. When I present hybrid techniques in Chapter 4, I will show why this approach is not always the best.

2.6. Dynamic Constraints

Change is the final component of the scheduling problems that I will consider in my research. A dynamic problem is one that changes over time, and is generally represented as a sequence of static problems in which each problem is a modification of the one before. Each modification is either a restriction (which adds or tightens constraints, possibly reducing the number of solutions) or a relaxation (which removes or weakens constraints, possibly increasing the number of solutions).

When solving a dynamic problem, it is important to consider not only efficiency—the speed with which a new solution is found after a change—but also stability—the degree of similarity between subsequent solutions. Previous research has pointed out the importance of stability (e.g., Verfaillie and Schiex 1994; Clement and Durfee 1998; Bartold and Durfee 2003; Gallagher, Zimmerman, and Smith 2006). A lack of stability between subsequent solution schedules could increase the communication costs to update the schedule, the computation costs incurred by agents reacting to the new schedule, and the frustration experienced by human users interacting with the system.

In the office example, the team meeting is originally scheduled for 2:00 in the conference room, but the schedule must change when Donna calls Bob to tell him that she and Evan need an extra half hour to prepare their presentation. The earliest time that Donna and Evan are both available is 2:00, so the team meeting is pushed back to 3:00, and the location must be changed to the team room in the engineers' building.

Alternatively, the team meeting could have been pushed back all the way to 4:00. Evan is unavailable after 4:00, so Donna would need to attend the meeting in his place. This alternative solution requires a larger time change (shifting the meeting time by two hours instead of just one) and requires an attendance change (Donna instead of Evan); hence, the option to meet at 3:00 is more similar to the original plan to meet at 2:00 than is the alternative option to meet at 4:00. One would expect that the 3:00 option might be preferable because it would not require the employees to adjust the times of other activities in their schedules as much, and it would not require Donna to mentally prepare for the meeting when she had expected that Evan would be presenting.

Before presenting methods to improve efficiency and stability, it is necessary to discuss a representation for problems with constraints that change over time. The

Dynamic CSP is a representation that models a finite-domain CSP with constraints that change over time.

Definition 2.16. A *Dynamic CSP (DCSP)* (Dechter and Dechter 1988) is a pair $\langle P_0, C\rangle$, where P_0 is a static CSP (the initial CSP of the sequence), and C is a sequence of changes. If C contains n changes, then the *solution* to the DCSP is a sequence of static CSP solutions $S = s_0, s_0, \ldots, s_n$, where s_i is the solution to problem P_i, and each CSP P_i is created by modifying $P_{i\text{-}1}$ according to change c_i, for $1 \leq i \leq n$.

The changes of a DCSP can be modeled entirely as changes within the set of constraints. Hence, it is only necessary to define two types of atomic DCSP changes:

1) **Tighten constraint.** One or more partial assignments that previously satisfied a constraint now violate it.
2) **Loosen constraint.** One or more partial assignments that previously violated a constraint now satisfy it.

A change to the domain of a variable can be modeled as a change to the unary constraint on that variable, and a change to the set of variables can be modeled either by tightening a constraint on a previously unconstrained variable or by completely loosening the constraints on a previously constrained variable.

Algorithms for solving the DCSP can be split into two categories: solution reuse and reasoning reuse.

Solution reuse techniques start with the solution assignment of the previous problem and attempt to modify it to create a solution for the current problem. Such a technique can improve both efficiency and stability if it is able to find a solution for the current problem after making only a small number of changes to the solution of the previous problem. In general, solution reuse techniques are variations of local search methods, so many of these techniques are incomplete (e.g., Selman, Levesque, and Mitchel 1992; Minton, *et al*. 1992; Morris 1993), although some complete solution reuse techniques have been developed (e.g., Verfaillie and Schiex 1994; Roos, Ran, and van den Herik 2000).

Reasoning reuse techniques (e.g., Schiex and Verfaillie 1993; van Hentenryck and Provost 1991) record information learned while solving one problem in the sequence and apply it when solving the next. A reasoning reuse algorithm can improve both efficiency and stability by trying to repeat the good decisions and avoid the bad decisions made while solving the previous problem. These techniques are specifically designed to work in the context of backtracking search.

Although some complete solution reuse techniques do exist, they are still based on local search methods, which prevent the use of many of the pruning techniques that improve efficiency in backtracking search. While solution reuse techniques are interesting in their own right, I will leave the study of their application to dynamic constraint-based scheduling problems for future work. On the other hand, reasoning reuse techniques are specifically designed to work in conjunction with backtracking search, so I will employ two reasoning reuse techniques in my research—nogood recording and oracles.

2.6.1. Nogood Recording

As discussed earlier, nogood recording (NGR) can improve efficiency when solving static finite-domain CSPs and static DTPs. Schiex and Verfaillie (1993) showed that NGR can also improve efficiency when solving finite-domain DCSPs. Basically, some of the nogoods that are recorded while solving one problem can be stored and applied while solving the next problem. Schiex and Verfaillie point out two properties of DCSPs that make it possible to decide which nogoods still apply after a CSP is changed.

Property 2.1. If a partial assignment *A* cannot be extended to a solution for CSP *P*, then *A* cannot be extended to a solution for any CSP that is a restriction of *P*.

Property 2.2. If a partial assignment *A* cannot be extended to a solution for CSP *P* because the constraints over the justification *J*, and if *P'* is a relaxation of *P*, but none of the constraints over *J* are relaxed, then *A* cannot be extended to a solution for *P'*.

The first property tells us that if a nogood is recorded while solving one problem, and if the next problem is a restriction, then that same nogood can still be safely applied without inadvertently pruning any solutions. This is true because a restriction can only remove solutions from the solution set of a problem. Since the nogood's assignment had no solution extensions before the restriction, it will still have no solution extensions after.

The second property tells us that if a nogood is recorded while solving one problem, and if the next problem is a relaxation, then that nogood can still be safely applied as long as the relaxation did not affect the constraints on the nogood's justification. Recall that, for nogood ⟨A, J⟩, the assignment A cannot be extended to a solution because of the constraints on J. If the constraints on J have not been relaxed (even though some other constraints were), then those constraints still prevent A from being extended to a solution.

2.6.2. Oracles

Oracles (van Hentenryck and Provost 1991) have also been applied to finite-domain DCSPs. During backtracking search, all partial assignments that were tested before the first solution was found must have been pruned. An oracle records the path to the first solution so the next search can try to repeat it, bypassing the pruned portion of the search space.

Definition 2.17. An oracle is a pair ⟨A, O⟩, where A is the solution assignment of the previous problem in the sequence, and O is the order in which variables were chosen to find this solution.

In the best case, the solution to the previous problem is still a solution to the current problem, so following an oracle will lead the search algorithm directly to a solution without backtracking. If a dead end is encountered while following an oracle, then the solver backtracks and discards the oracle, relying on heuristics to choose variables and values from then on. Even if the oracle does not lead directly to a solution, it can still improve efficiency by guiding the search through a consistent assignment of at least at some of the variables near the root of the search tree. An oracle can also improve stability by reusing part of the previous solution assignment. In Chapter 3, I will test the effectiveness of oracles and nogood recording when solving dynamic scheduling problems.

Chapter 3

Dynamic Disjunctive Temporal Problems

3.1. Introduction

In order to manage a dynamic scheduling problem, it is necessary to handle changes to the temporal constraints. Despite prior successful approaches to developing dynamic techniques for finite-domain CSPs that change over time (van Hentenryck and Provost 1991; Schiex and Verfaillie 1993), no prior work has addressed the Dynamic DTP.

In this chapter, I formally define the Dynamic Disjunctive Temporal Problem (DDTP). As a real-world example, I describe the complex, dynamic scheduling problem of air traffic control. I then describe several techniques for improving efficiency while solving a DDTP, including nogood recording, oracles, and a novel technique that I call justification testing, which is based on the idea that all variables are not equal. Finally, I discuss the concept of temporal stability, describing a family of stability metrics and showing how the techniques introduced to increase efficiency can also increase stability. I compare the techniques presented in this chapter on both randomly generated problems and on real-world problems based on actual air traffic control recordings.

The Dynamic Disjunctive Temporal Problem can be defined as follows:

Definition 3.1. A *Dynamic Disjunctive Temporal Problem (DDTP)* is a pair $\langle P_0, C\rangle$, where P_0 is a static DTP (the initial DTP of the sequence), and C is a sequence of change sets (defined below). If C contains n change sets, then the solution to the DDTP is a sequence of static DTP solutions $S = s_0, s_1, \ldots, s_n$, where s_i is the solution to problem P_i, and each DTP P_i is created by modifying P_{i-1} according to change set c_i, for $1 \le i \le n$.

A *change set* is a finite set of atomic DDTP changes. As with the Dynamic CSP (discussed earlier in Chapter 2), all of the changes in a DDTP can be expressed as changes to the constraints. The difference is that the disjunctive temporal constraints of a DTP have a specific structure, so different types of changes can have different effects from one another, leading to six different types of atomic changes in a DDTP. As in a DCSP, the change types can be partitioned into restrictions and relaxations. Restrictions (1-3 below) can only remove solutions from the solution set, while relaxations (4-6 below) can only add solutions to the solution set.

1) **Tighten bound.** The bound b of a TD constraint $x - y \le b$ is decreased.
2) **Remove disjunct.** A disjunct is removed from a disjunctive temporal constraint.
3) **Add constraint.** A disjunctive temporal constraint is added to the DTP.
4) **Loosen bound.** The bound b of a TD constraint $x - y \le b$ is increased.
5) **Add disjunct.** A disjunct is added to a disjunctive temporal constraint.
6) **Remove constraint.** A disjunctive temporal constraint is removed from the DTP.

As I present the different dynamic techniques throughout this chapter, I will describe them as if the sequence of change sets *C* contains only a single set of changes. This simplification is consistent with previous research on dynamic CSPs, such as (Dechter and Dechter 1988; Verfaillie and Schiex 1994). With only a single set of changes, the sequence of DTPs only contains two problems—P_0, which I will refer to as the *initial problem*, and P_1, which I will refer to as the *modified problem*. If the initial problem is consistent, I refer to the DDTP as *initially consistent*, and if the initial problem is inconsistent, I refer to the DDTP as *initially inconsistent*. The techniques presented here are not limited to dynamic sequences of only two problems—rather, they can be applied between any two consecutive problems at any point in the sequence.

3.2. Air Traffic Control Domain

The problem domain of air traffic control is complex, dynamic, and extremely important. Air traffic controllers must regulate and optimize traffic flow while reacting to unexpected events and maintaining safety. Air traffic control is so important that it has been studied for decades (Sokkappa and Steinbacher 1977; Petre 1991). Before the 1970s, the purpose of air traffic control was simply to maintain a safe distance between aircraft and respond to special situations such as weather and emergencies. Aircraft would fly to their destinations as quickly as possible, and then hold until a runway became available. While this method was simple, it was also very inefficient, wasting both time and fuel. After the energy crisis of the 1970s and the air traffic controller strike of 1981, it was evident that an automated system was necessary improve efficiency and controller workload.

In the 1990s, the Federal Aviation Administration (FAA) and the National Aeronautical and Space Agency (NASA) worked together to develop the Center-TRACON Automation System (CTAS) to assist air traffic controllers (Wong 2000). The Traffic Management Advisor (TMA) is the component of CTAS that assists the controllers with planning an approach route and schedule for incoming aircraft, and the heart of the TMA is the Dynamic Planner (DP).

For each aircraft, the DP must assign a runway and schedule a time to pass through each of a set of reference points along its approach path. As an aircraft approaches its destination, it first enters the airspace of the destination airport's Center. A Center is a large region of the country, generally several hundred miles across. The first reference point the aircraft crosses in the Center's airspace is the outer meter arc (see Figure 3.1). Next, the aircraft crosses into the airspace of the Terminal Radar Approach Control (TRACON), which is the area immediately surrounding the airport. The reference point at which the aircraft enters the TRACON airspace is called a meter fix, and meter fixes are grouped into gates. The aircraft then crosses the final approach fix before touching down on the runway. There is one final approach fix for each runway.

The DP must satisfy a number of constraints when scheduling incoming aircraft. These constraints can be partitioned into two categories, hard constraints and soft constraints. The hard constraints are based on the laws of nature or FAA regulations and are necessary to ensure safety. The soft constraints are added by the air traffic controllers and are meant to improve traffic flow and efficiency.

The first of the hard constraints is the estimated time of arrival. For each aircraft, the DP is given an estimated time of arrival (ETA) for each possible reference point. The

ETA is the minimum time required for an aircraft to reach a reference point assuming that no other aircraft interfere, and is calculated by other CTAS components based on flight plans, radar tracks, weather conditions, and physical models. At each reference point, aircraft must be scheduled in first come, first served order. That is, if the ETA of aircraft *A* is before the ETA of aircraft *B* for a particular reference point, then aircraft *A* must be scheduled to arrive at the reference point before aircraft *B*. Each aircraft also has a maximum time to travel between any two reference points without resorting to drastic measures (such as circling or flying a zigzag pattern).

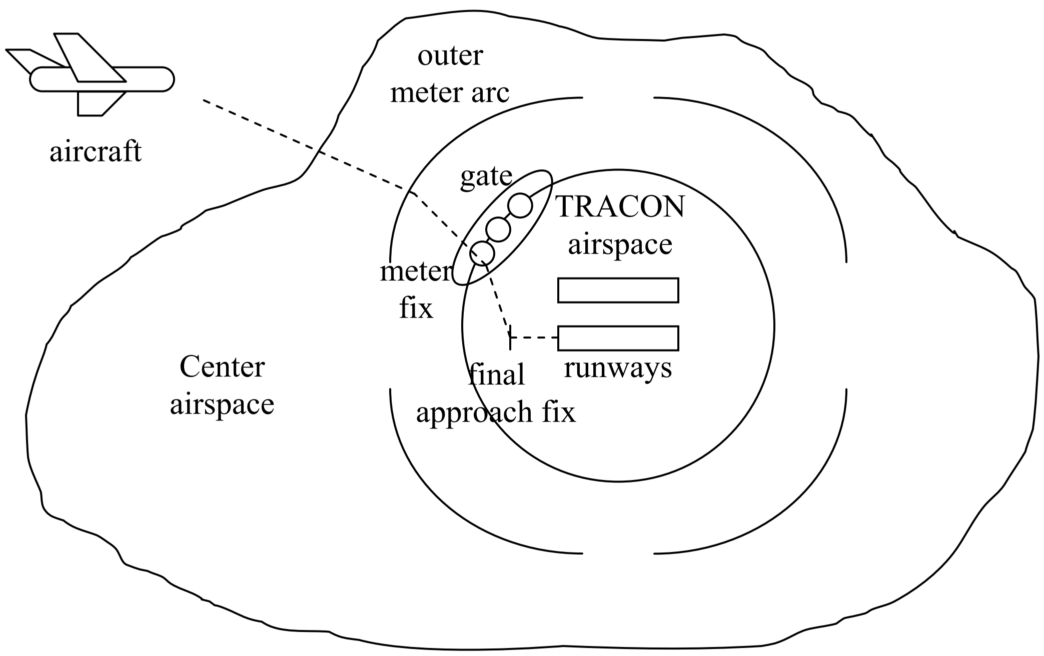

Figure 3.1. Organization of Center-TRACON airspace.

If two aircraft will pass through the same point in space, the second aircraft must follow by enough distance such that it is not affected by the turbulence of the first. At the meter fix, this distance is called miles-in-trail, and at the runway threshold, this distance is called wake vortex separation. Finally, after an aircraft lands, no other aircraft can land on the same runway until the first has been cleared. The minimum time required between two aircraft landing on the same runway is called runway occupancy time.

In addition to these hard constraints, the air traffic controllers have the option of applying several types of soft constraints in order to balance traffic flow. Generally, air traffic controllers operate under one of several configurations. Each configuration defines the subset of runways that are available for aircraft of each engine type to land on. The configuration is changed regularly to account for the usual changes in traffic patterns throughout the day. For each configuration, the air traffic controllers choose one of a set of possible flow parameter sets. Each flow parameter set acts as a set of additional constraints to control traffic flow. These constraints include acceptance rates, runway separations, and occupancy times. An acceptance rate limits the number of aircraft that can land at the airport during a one hour interval. By applying an acceptance rate, air traffic controllers can delay some aircraft so they land at a time that is slightly less busy. Runway separations and occupancy times are the same as the hard wake vortex separation and runway occupancy times, except that these separations can be increased to slow down traffic.

Given all of this information, the DP must assign a runway and a schedule for aircraft. The DP operates by first allocating runways, then scheduling a time for each aircraft to pass each reference point. Aircraft are assigned to runways in order of ETA. For each airport, a decision tree is used to choose a runway for each aircraft based on the current configuration and the engine type of the aircraft. Each aircraft is then assigned a time to its meter fix. The aircraft with the earliest ETA is scheduled to pass its meter fix at exactly its ETA. Then the aircraft with the second earliest ETA is scheduled. If both aircraft are passing through the same meter fix, then the second aircraft is delayed in order to meet the miles in trail constraint. The time at which an aircraft is scheduled to pass a particular reference point is called its scheduled time of arrival (STA). The STA of each aircraft at each reference point is equal to its ETA for that reference point plus any delay necessary to meet the scheduling constraints. After scheduling each aircraft to its meter fix, the ETAs to the final approach fixes and runway thresholds are updated. The process is then repeated in order to schedule a time of arrival for each aircraft at its final approach fix, and then repeated again to schedule a time of arrival for each aircraft at its runway threshold.

Note that at no time does the DP ever backtrack over a decision it has made. Generally, traffic is light enough that this approach is successful. It is possible, although rare, that an aircraft will be delayed so much that its scheduled time of arrival exceeds its maximum travel time between a pair of reference points. If this happens, the air traffic controllers are notified so they can modify the schedule manually. In other words, the DP applies a polynomial-time algorithm to an NP-complete problem, relying on humans to find a solution when the algorithm fails.

The air traffic control problem domain is not only complex, but also highly dynamic. The schedule must be updated every time a new aircraft enters the airspace, any aircraft passes a reference point, or an ETA is updated due to new radar tracks or atmospheric conditions, such as air temperature and wind speed. Additionally, the schedule must be updated whenever the air traffic controllers change the configuration or any of the flow parameters. Finally, the DP must respond to unexpected events, such as a thunderstorm or a runway closure.

In fact, the air traffic control domain is so dynamic and the schedule must be updated often enough that it can become a nuisance to air traffic controllers. Whenever any scheduling constraint is updated, the DP algorithm reacts by updating the runway allocation and schedule. This can lead to a thrashing behavior, with large sections of the schedule changing from one moment to the next. In other words, the schedule lacks stability over time.

In an attempt to reduce the impact of this instability, air traffic controllers can impose STA freezes and sequence freezes that lock aircraft into a schedule or sequence as they approach reference points. This approach, however, makes it impossible to optimize the schedule of frozen aircraft, and does nothing to improve the stability of the rest of the schedule. While the DP algorithm is fast enough to keep up with the changing environment of the air traffic control domain, it is not guaranteed to produce a solution every time, and it is not capable of optimizing the schedule to improve stability.

Because the air traffic control problem is NP-complete, there is no known complete polynomial-time algorithm for solving it. To guarantee that a solution is found whenever one exists, it is necessary to apply a more sophisticated algorithm, such as a depth-first search algorithm. In this chapter and the next, I will present a variety of techniques that can be applied to improve the performance of search algorithms when solving complex scheduling problems like those found in the air traffic control domain. I will compare many of these techniques using problems based on actual air traffic control recordings. I will begin here by presenting a set of techniques that can improve performance when solving DDTPs.

3.3. DDTP Techniques

The meta-SAT transformation of a DTP makes it possible to apply existing dynamic techniques such as nogood recording or oracles when solving a DDTP. In this section, I explore the effect of both these techniques on DDTP efficiency, along with a novel technique that I call justification testing.

3.3.1. Nogood Recording

Nogood recording is the process by which the solver records dead ends in the search tree so as to avoid them in the future. Nogood recording can be applied whether the initial problem is consistent or not; however, previous work has only considered the effectiveness of this technique when the initial problem is consistent (e.g., van Hentenryck and Provost 1991; Schiex and Verfaillie 1993). Although never stated explicitly, previous work on the subject seems to assume that the initial problem of a sequence is always consistent, and if the constraints are ever restricted to the point of inconsistency, then the most recent restriction is retracted. In many domains, however, these assumptions do not hold. For example, in a mixed-initiative setting, a user may add a constraint that causes an inconsistency. The user may then wish to relax a *different* constraint involved in the inconsistency, not necessarily the last constraint that was added. Because of this, I do not make these assumptions of consistency in my research.

3.3.2. Oracles

An oracle records the search path to the solution found for the initial problem, and then attempts to follow that same search path when solving the modified problem after a change to the constraints. The intuition is that the modified problem is still similar to the initial problem, so it should be possible to find a similar solution quickly by following the same search path. An oracle can only be applied when the initial problem is consistent; if the initial problem is inconsistent, then there is no search path to follow. In previous research (van Hentenryck and Provost 1991), oracles have only been applied to dynamic finite-domain constraint problems. How should the basic intuition of an oracle be interpreted when solving a dynamic temporal constraint problem?

One possible interpretation is to ignore the temporal aspect of the problem and apply the idea of the oracle to the meta-level problem. That is, when searching for a solution to the modified DTP, select the meta-level variables and values in the same order as the search path to the meta-level solution assignment of the initial DTP. Because this interpretation uses the meta-values to guide the search, I will refer to this as a *meta-value oracle*.

A second possible interpretation is that the meta-values should be chosen to guide the search not toward a similar meta-level solution, but instead directly toward a similar solution schedule. That is, when choosing a meta-level value, choose the one that causes the solution schedule of the modified problem to differ from the solution schedule of the initial problem the least. The amount by which two schedules differ can be measured using dynamic distance, as defined below:

Definition 3.2. Given the d-graphs D_i and D_{i+1} of component STPs from two consecutive DTPs in a DDTP, and given two timepoints, x and y, the *dynamic distance*, $dd[x, y]$, is the amount by which the induced bound on the difference $x - y$ has decreased from D_i to

D_{i+1}. That is, $dd[x, y] = D_i[x, y] - D_{i+1}[x, y]$, where $D_i[x, y]$ is the induced bound on the difference $x - y$ according to d-graph D_i.

At first pass, it might seem that this interpretation would lead to relatively similar performance compared to the first interpretation—if the temporal bounds have not changed much, then choosing the same meta-value from the initial solution should lead to a solution STP with a similar transitive closure. It is possible, however, that a small change in an explicit temporal constraint could result in large changes in the implied meta-level constraints. For example, tightening a single temporal bound could implicate that temporal difference constraint in many more negative cycles, adding a large number of implied meta-level constraints. Essentially, the meaningful information is contained in the temporal difference constraints, while the meta-values are just arbitrary labels. According to this intuition, it is more important to match the transitive closure of the temporal bounds (that is, the d-graph) than it is to match the meta-value choices of the initial search path. Because this interpretation uses the temporal bounds to guide the search, I will refer to this as a *temporal bounds oracle*.

Note that moving an event forward or backward in time does not map simply to an increase or decrease in a single induced bound. In general, if the time between two timepoints falls within an interval, and if that interval is moved forward or backward in time, but its size does not change, then one induced bound will increase while the opposite induced bound will decrease by the same amount.

For example, let the temporal reference *TR* represent 12:00pm, and let M_S represent the start of a meeting. If in the initial solution, the meeting must start between

2:00pm and 3:00pm, then $D_0[M_S, TR] = 180$ (the meeting must start before 3:00pm) and $D_0[TR, M_S] = -120$ (the meeting must start after 2:00pm). If the meeting must be postponed for one hour, then in the modified solution, $D_1[M_S, TR] = 240$ and $D_1[TR, M_S] = -180$, so $dd[M_S, TR] = -60$ and $dd[TR, M_S] = 60$. On the other hand, if the meeting must be moved up one hour, then in the modified solution, $D_1[M_S, TR] = 120$ and $D_1[TR, M_S] = -60$, so $dd[M_S, TR] = 60$ and $dd[TR, M_S] = -60$. In both cases, the meeting start time has been moved one hour, and in both cases, one dynamic distance is 60 and the other dynamic distance is -60. Using dynamic distance as a measure of the difference between two schedules, the *direction* of a change in the schedule does not matter, only the *degree* of the change matters.

3.3.3. Justification Testing

When the consistency assumptions made in previous work on nogood recording are discarded, it becomes evident that a niche is left open. Nogood recording can be applied whether the initial problem is consistent or not, while oracles can be applied only when the initial problem is consistent. There is no analogue to oracles that can only be applied when the initial problem is inconsistent. To fill this niche, I developed a new technique that I call justification testing. Whereas an oracle attempts to reproduce the search path of a consistent problem, justification testing attempts to reproduce the proof that a problem contains an inconsistency.

Recall that, if the initial problem is inconsistent, a solver can produce a justification. A justification is a subset of variables whose constraints contain an inconsistency. Assume that the initial problem is inconsistent, and its justification is the set of meta-level variables *J*. Let C_0 be the set of constraints over *J* in the initial problem, and let C_1 be the set of constraints over *J* in the modified problem. It is already known that C_0 constrains the variables of *J* so much so that there is no consistent assignment to them.

Justification testing capitalizes on the fact that these justification variables are different from the other variables. The intuition behind justification testing is that the constraints of C_1 are similar to the constraints of C_0, so it is likely that the variables of *J* are also tightly constrained by the constraints of C_1. Justification testing takes advantage of this intuition by moving the variables of *J* to the beginning of the variable ordering when solving the modified problem. The advantage here is that the justification provides a global perspective, suggesting that the variables of *J* are more tightly constrained than other variables in the problem, whereas standard variable selection heuristics take a more local perspective, comparing one individual variable to another. This helps to maximize propagation near the top of the search tree and minimize backtracking overall.

The variable selection method using justification testing is laid out in the **Select-Variable-JT** procedure of Figure 3.2. If any of the variables of the justification are still unassigned, then one of those variables is selected according a standard variable selection heuristic. In the case of my SAT-based implementation, this is the VSIDS heuristic. Only after all of the variables of the justification have been assigned can a variable that is not in the justification be selected.

```
Select-Variable-JT(unassigned variables $V$, justification $J$)
1   if ($J \cap V \neq \emptyset$) then
2       Select-Variable($J \cap V$)
3   else
4       Select-Variable($V$)
```

Figure 3.2. The **Select-Variable-JT** procedure.

3.4. DDTP Efficiency

I will compare the efficiency of these dynamic techniques in two phases. I will first compare nogood recording and oracles using initially consistent problems; I will not compare justification testing in this set of experiments because it can only be applied when the initial problem is inconsistent. Similarly, oracles can only be applied when the initial problem is consistent, so I will perform a second set of experiments to compare nogood recording and justification testing using initially inconsistent problems.

3.4.1. Testing Efficiency with Initially Consistent Randomly Generated Problems

To compare the efficiency of nogood recording and oracles, I created a test set of randomly generated initially consistent DDTPs. Each DDTP of this set contains 10, 20, 30, or 40 timepoints. The tightness of constraints of each initial problem is 20%, 40%, 60%, or 80% consistency (I will explain exactly what this means shortly). Finally, the number of changes between the initial problem and the modified problem is 2, 4, 6, or 8. I generated 20 random DDTPs for each combination of problem size, constraint tightness, and number of changes, resulting in a total of 1,280 randomly generated DDTPs in this set. By testing so many combinations of parameters, this test set attempts to cover a variety of situations that might be encountered in a real-world problem domain.

In the initial problem of each DDTP, each constraint is a disjunction of two temporal difference constraints (in many scheduling domains, the majority of disjunctive temporal constraints are a disjunction of two temporal difference constraints to prevent two intervals from overlapping). The timepoints of each temporal difference constraint are chosen randomly with uniform probability, and the bound of each temporal difference is a real number chosen randomly with uniform probability from the range [-100, 100]. A temporal horizon of 1,000 is imposed on these randomly generated problems, meaning that all timepoints must be scheduled within 1,000 units of time of one another.

I control the tightness of the constraints of the initial problem of a DDTP by controlling the number of disjunctive temporal constraints. Before generating the random DDTPs, I experimented with randomly generated static DTPs to find the relationship between the number of constraints and the percentage of DTPs that are consistent. For each problem size (10, 20, 30, and 40 timepoints), I varied the ratio of constraints to timepoints from 3.80 to 7.00 at increments of 0.05. For each combination of problem size and ratio of constraints, I tested the consistency of 1,000 randomly generated problems. Figure 3.3 shows the percentage of problems that were found to be consistent using each ratio of constraints. For each problem size, Table 3.1 lists the ratio of constraints to timepoints necessary to achieve the desired consistency percentages.

Using this information, I can control the tightness of constraints by controlling the number of constraints in a DDTP of a given size. If the tightness of the constraints of a DTP is 20% consistency, this means that the DTP has enough constraints that 20% of randomly generated DTPs of the same size and same number of constraints are consistent. Note that two DTPs of the same size might both be consistent, but if the first is at 20% consistency and the second is at 80% consistency, then the first is more tightly constrained than the second.

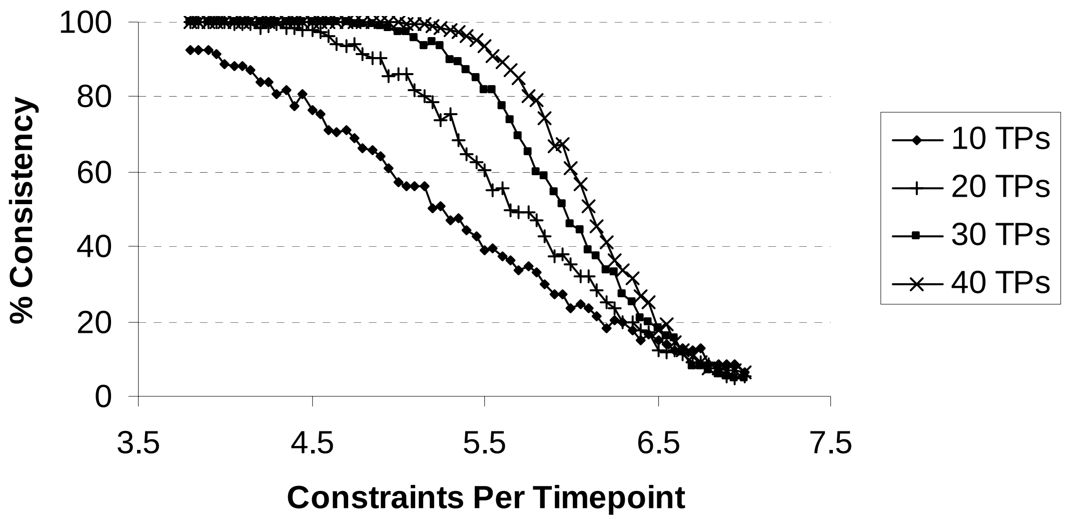

Figure 3.3. Percentage consistency of randomly generated DTPs as a function of the ratio of constraints to timepoints.

% Consistency	80%	60%	40%	20%
10 Timepoints	4.35	4.95	5.50	6.25
20 Timepoints	5.15	5.50	5.90	6.35
30 Timepoints	5.55	5.80	6.10	6.45
40 Timepoints	5.75	6.00	6.20	6.55

Table 3.1. Ratio of disjunctive temporal constraints to timepoints necessary to achieve percentage consistency of randomly generated DTPs.

Given the size and number of constraints for a particular DDTP, I would first randomly generate a static DTP and test its consistency. If it was inconsistent, I would discard it, randomly generate another static DTP, and test its consistency. This process repeated until a consistent static DTP was found to serve as the initial DTP in the dynamic problem.

Once the initial DTP was found, it was extended to become a dynamic problem by adding a set of changes. In each of the initially consistent DDTPs, each change of the change set is a restriction. If an initially consistent problem is relaxed, the resulting modified problem must also be consistent, so I do not test this case; I will test initially inconsistent problems that are then relaxed in later sections. The type of restriction for each change of an initially consistent DDTP is chosen randomly with uniform probability from the three restriction types listed in Definition 3.1. In a real-world problem, a single domain-level change could result in a mixture of simultaneous atomic DDTP changes, so the change sets of this randomly generated problem set contain a mixture of change types to reflect this possibility. Once the type of change is chosen, a change is generated randomly as follows:

1) **Tighten bound.** A temporal difference constraint is chosen randomly with uniform probability, and its bound is decreased by a random amount chosen with uniform probability in the range [0, 100].
2) **Remove disjunct.** A disjunctive temporal constraint with two or more disjuncts is chosen randomly with uniform probability, and one of its disjuncts is chosen randomly with uniform probability and removed.
3) **Add constraint.** A new disjunctive temporal constraint is generated in the same manner as in the initial problem and added.

In order to compare the dynamic techniques, I implemented them in a DDTP solver based on the Boolean satisfiability solver MiniSAT (Een and Sörensson 2003). MiniSAT is a very efficient, very compact solver that incorporates the most successful

SAT efficiency techniques. Even though MiniSAT is implemented in only 600 lines of C++, it is competitive with any of the most efficient SAT solvers, many of which are tens of thousands of lines of code.

I extended the MiniSAT code to be able to solve DDTPs based on a meta-SAT transformation. My extension of MiniSAT is representative of the most efficient constraint solvers that are capable of solving DTPs, such as BarcelogicTools (Nieuwenhuis and Oliveras 2005) or Yices (Dutertre and de Moura 2006), in that it converts the problem to a meta-level SAT representation and solves it using a depth-first search algorithm that periodically tests the consistency of the temporal constraints implied by the current partial assignment. The experiments were run in a Linux environment with 1GB of memory. Each DTP within the dynamic sequence was limited to 500 seconds before being terminated—if any problem ran beyond the time limit, then the run time reported for this problem is 500 seconds.

Using the set of 1,280 randomly generated initially consistent DDTPs, I compared six combinations of techniques. These combinations are as follows:

1) **Baseline.** Each DTP in the dynamic sequence is solved as if it were an unrelated static problem, without the use of nogood recording or oracles.
2) **NGR.** Nogoods that are learned while solving the initial problem are kept and applied when solving the modified problem; oracles are not used.
3) **O-MV.** A meta-value oracle records the search path to the solution of the initial problem and guides the search for a solution to the modified problem.
4) **NGR/O-MV.** Nogood recording and a meta-value oracle are used in combination.

5) **O-TB.** A temporal bounds oracle records the search path and solution STP of the initial problem and guides the search for a solution to the modified problem.
6) **NGR/O-TB.** Nogood recording and a temporal bounds oracle are used in combination.

Nogood recording can prevent the solver from exploring some dead-end branches of the search tree, and it has already been shown to be effective when applied to finite-domain DCSPs. I therefore hypothesize that nogood recording will also improve efficiency when applied to DDTPs:

Hypothesis 3.1. Nogood recording will significantly improve efficiency over the baseline algorithm when applied to initially consistent DDTPs.

To illustrate my analysis of this hypothesis, Figure 3.4 compares the run time required to solve the modified problem of each randomly generated initially consistent DDTP using both the baseline algorithm and nogood recording. Each plus sign plots the run time using the baseline algorithm, and each point plots the run time using nogood recording. The plus signs plotting the run time of the baseline algorithm are packed closely enough to one another that they look more like a thick black line from the lower left of the graph to the upper right. The problems are arranged along the horizontal axis in ascending order according to the run time using the baseline algorithm. The vertical axis measures run time in seconds on a logarithmic scale. Because the problems are arranged in ascending order by run time, and because run time is measured on a logarithmic scale, a vertical distance between two points toward the right side of the graph represents a larger difference in run time than the same vertical distance between two points toward the left side.

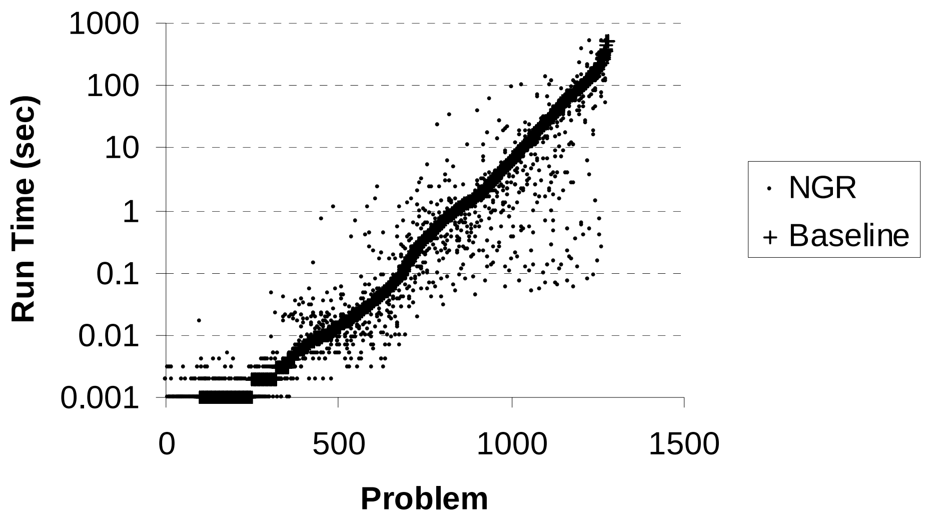

Figure 3.4. Run time of randomly generated initially consistent DDTPs using the baseline algorithm and nogood recording.

Because the problem set contains such a range over problem size and constraint tightness, the time required to solve some of the problems is several orders of magnitude greater than the time required to solve others. It is possible to overcome this variability between problems because the data is paired—that is, each problem can be tested with all combinations of techniques. In contrast, in a medical study, for example, only half of the subjects could be given the treatment while the other half would be given a placebo. A Student's paired t-test accounts for this by examining the difference in run time for each individual problem. Essentially, the differences between the pairs of measurements are normally distributed (Devore 1995), so a Student's paired t-test considers the variability in the average of the differences, rather than the variability in the difference between the averages. Although it may be difficult to see in Figure 3.4, a Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.1 is correct for this set of randomly generated DDTPs—nogood recording significantly improves efficiency over the baseline algorithm. This result confirms the fact that nogood recording is effective not only for DCSPs, but also for DDTPs.

Oracles have also proven effective when applied to finite-domain DCSPs, so I also hypothesize that they will improve efficiency when applied to DDTPs:

Hypothesis 3.2. Meta-value oracles and temporal bounds oracles will both significantly improve efficiency over the baseline algorithm when applied to initially consistent DDTPs.

To test Hypothesis 3.2, Figure 3.5 compares the run time of each DDTP using the baseline algorithm against the run time using meta-value oracles, and Figure 3.6 compares the run time using the baseline algorithm against the run time using temporal bounds oracles. In both of these figures, as in Figure 3.4, each plus sign plots the run time using the baseline algorithm, and each point plots the run time using oracles. The problems are arranged along the horizontal axis in ascending order according to the run time using the baseline algorithm. For both types of oracles, a Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.2 is correct for this set of randomly generated DDTPs—meta-value oracles and temporal bounds oracles both improve efficiency over the baseline algorithm.

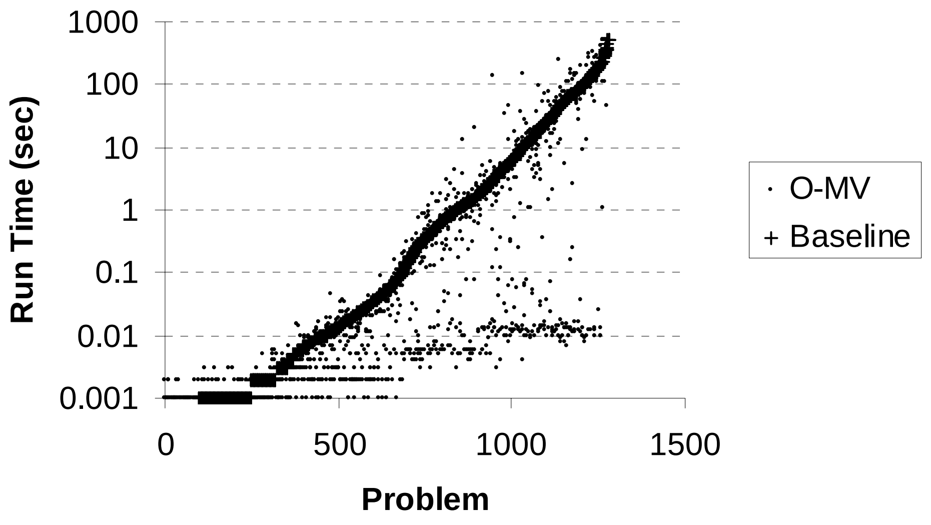

Figure 3.5. Run time of randomly generated initially consistent DDTPs using the baseline algorithm and meta-value oracles.

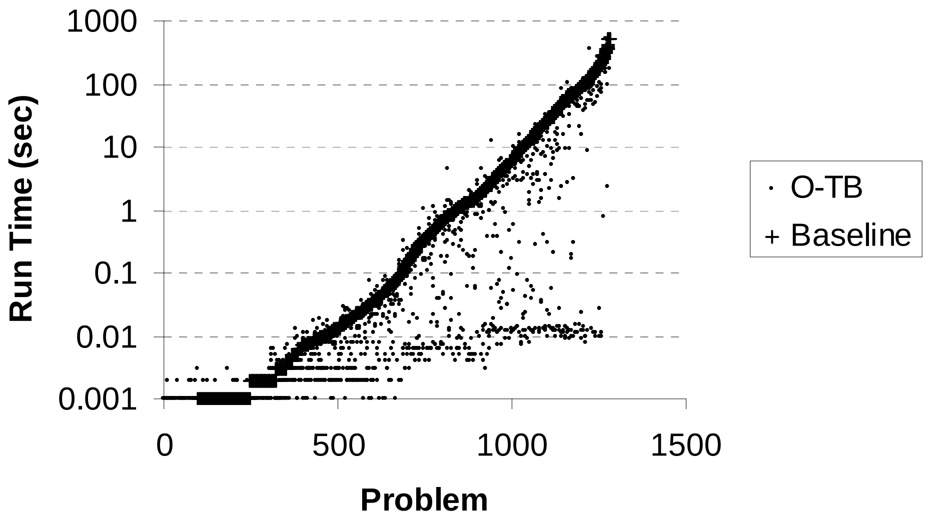

Figure 3.6. Run time of randomly generated initially consistent DDTPs using the baseline algorithm and temporal bounds oracles.

It is interesting to consider which type of oracle is more effective, meta-value oracles or temporal bounds oracles. As pointed out earlier, a small change in a temporal constraint could result in large changes in the implied meta-level constraints. If this is the case, then following a meta-level oracle could lead the search toward a very different part of the search space, possibly away from solutions to the modified DTP. A temporal bounds oracle, on the other hand, continually pushes the search toward a similar solution schedule. I therefore hypothesize that temporal bounds oracles will outperform meta-value oracles.

Hypothesis 3.3. Temporal bounds oracles will improve efficiency significantly more than meta-value oracles improve efficiency when applied to initially consistent DDTPs.

Figure 3.7 tests this hypothesis by comparing the run time of the randomly generated initially consistent DDTPs using meta-value oracles to the run time using temporal bounds oracles. Each plus sign plots the run time using meta-value oracles, each point plots the run time using temporal bounds oracles, and the problems are arranged along the horizontal axis in ascending order according to their run time using meta-value oracles. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.3 is correct for this randomly generated test set—temporal bounds oracles are significantly more efficient than meta-value oracles.

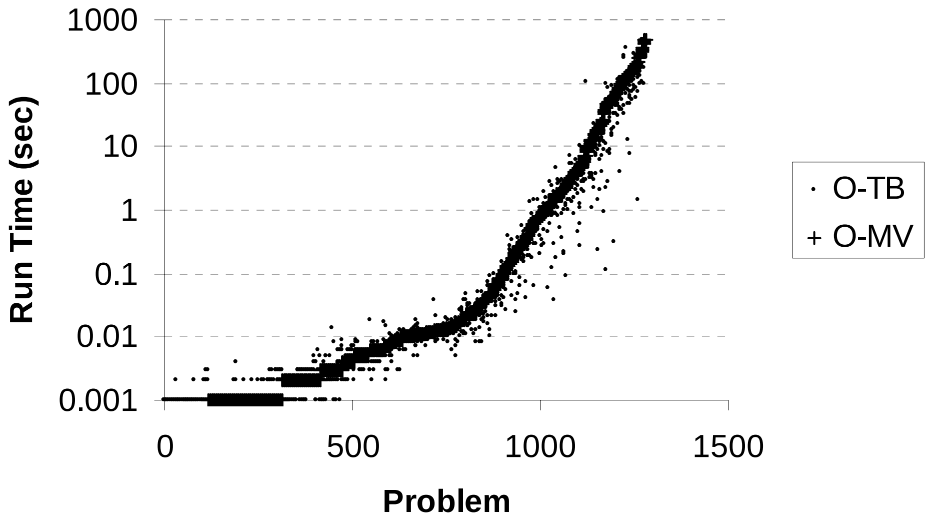

Figure 3.7. Run time of randomly generated initially consistent DDTPs using meta-value oracles and temporal bounds oracles.

It is also interesting to consider the relative performance of oracles versus nogood recording. While nogood recording can help to avoid some dead-ends, if an oracle is completely successful, it can help to avoid all dead-ends. I hypothesize that both types of oracles are more effective than nogood recording.

Hypothesis 3.4. Meta-value oracles and temporal bounds oracles will both significantly improve efficiency over nogood recording when applied to initially consistent DDTPs.

To test Hypothesis 3.4, Figure 3.8 compares the run time of each DDTP using nogood recording against the run time using meta-value oracles, and Figure 3.9 compares the run time using nogood recording against the run time using temporal bounds oracles. In both figures, each plus sign plots the run time using nogood recording, each point plots the run time using oracles, and the problems are arranged along the horizontal axis in ascending order according to the run time using nogood recording. For this set of problems, Student's paired t-tests confirm with 99.5% confidence that Hypothesis 3.4 is partially correct—temporal bounds oracles significantly improve efficiency over nogood recording, but meta-value oracles do not.

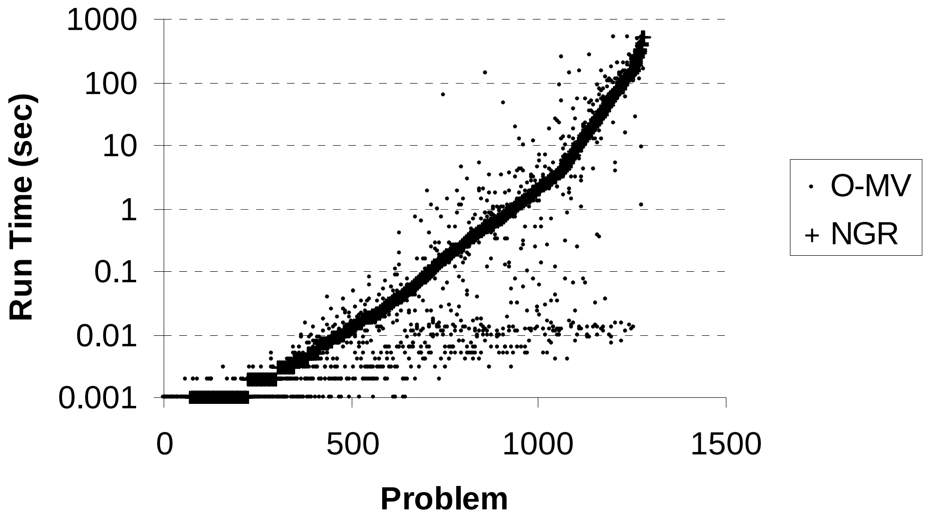

Figure 3.8. Run time of randomly generated initially consistent DDTPs using nogood recording and meta-value oracles.

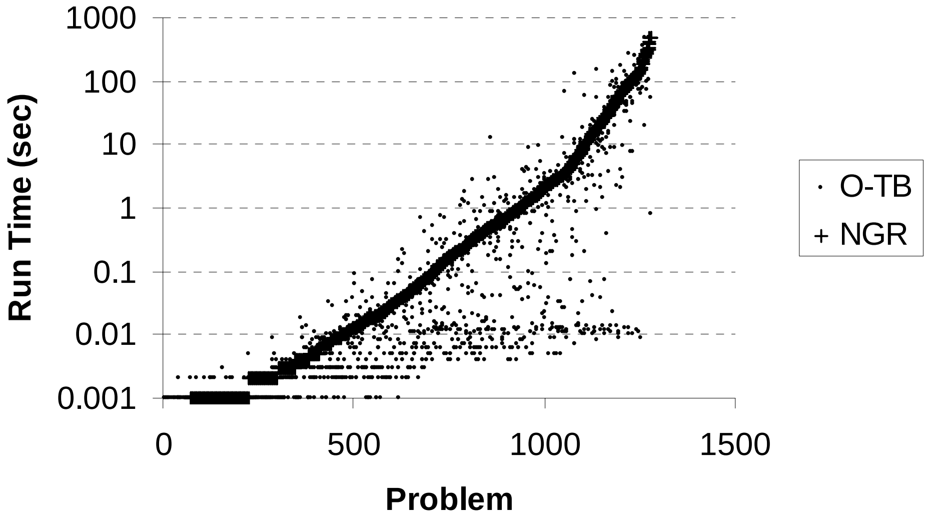

Figure 3.9. Run time of randomly generated initially consistent DDTPs using nogood recording and temporal bounds oracles.

Finally, there is no reason why nogood recording and oracles cannot be used together, so we should consider the effects combining them. If the oracle is completely successful and a solution is found without backtracking, then nogood recording will have no effect on performance. On the other hand, if it is necessary to backtrack, nogood recording could improve efficiency by avoiding dead-ends. I hypothesize that the efficiency of oracles can be improved by combining them with nogood recording.

Hypothesis 3.5. The combination of nogood recording and oracles (either meta-value oracles or temporal bounds oracles) will significantly improve efficiency over oracles alone when applied to initially consistent DDTPs.

To test Hypothesis 3.5, Figure 3.10 compares the run time using meta-value oracles and nogood recording in combination against the run time using meta-value oracles alone. Similarly, Figure 3.11 compares the run time using temporal bounds oracles and nogood recording in combination against the run time using temporal bounds oracles alone. In both figures, each plus sign plots the run time using oracles alone, each point plots the run time using the combination of oracles and nogood recording, and the problems are arranged along the horizontal axis in ascending order according to the run time using just oracles. For both types of oracles, a Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.5 is correct for this set of randomly generated DDTPs—the performance of meta-value oracles and temporal bounds oracles can be improved by combining them with nogood recording.

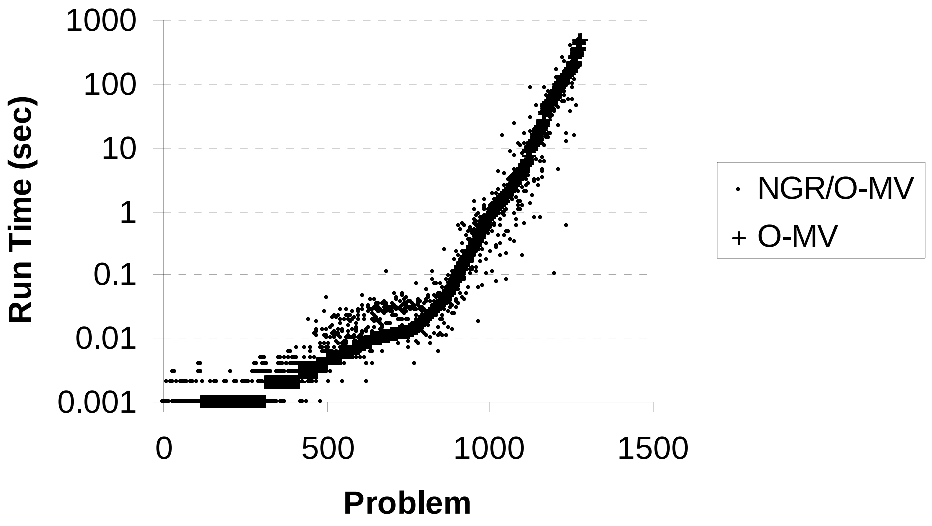

Figure 3.10. Run time of randomly generated initially consistent DDTPs using meta-value oracles alone and in combination with nogood recording.

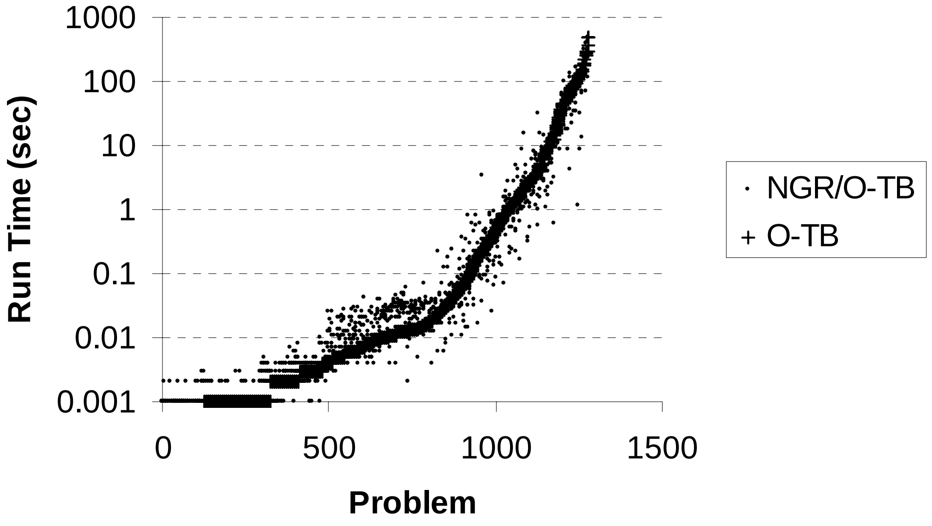

Figure 3.11. Run time of randomly generated initially consistent DDTPs using temporal bounds oracles alone and in combination with nogood recording.

Because the comparisons depicted in Figures 3.4 through 3.11 do not provide a clear indication of the relative efficiency of one combination of dynamic techniques compared to another, Figure 3.12 reports the average improvement (in percent) of each technique over the baseline run time. For example, if a technique has a 50% average run time improvement, then it runs two times as fast as the baseline algorithm on average, and if a technique has a 90% average run time improvement, then it runs ten times as fast as the baseline algorithm on average. This figure corresponds to the Student's paired t-test, which measures the average and variability of the differences between two techniques. In Figure 3.12, the height of each bar represents the arithmetic mean of the differences, and the error bars represent the standard error.

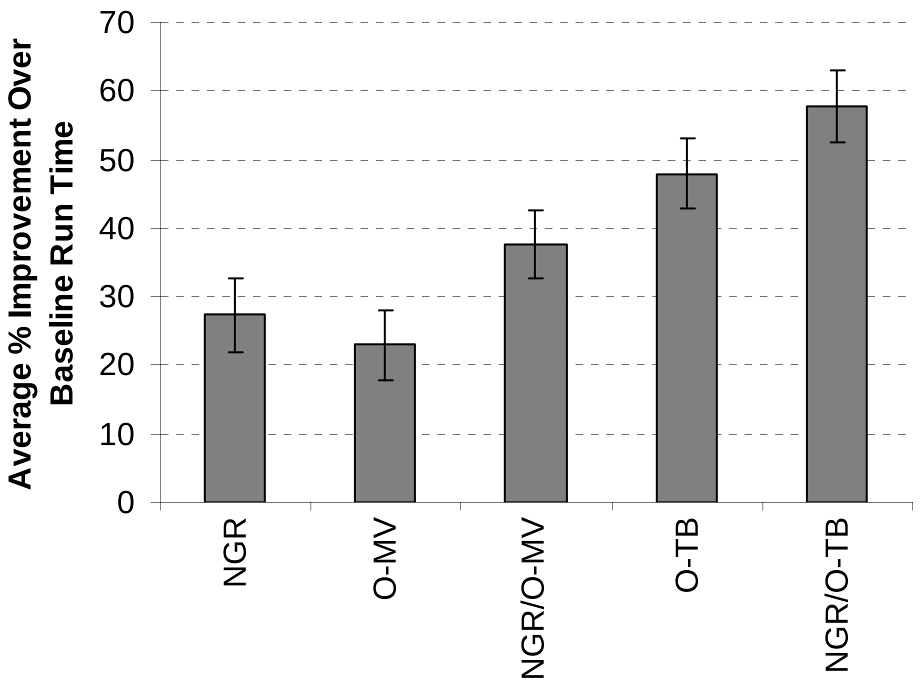

Figure 3.12. Average percent improvement over baseline run time using dynamic techniques alone and in combination to solve randomly generated initially consistent DDTPs.

The temporal bounds interpretation of an oracle that I devised was the most efficient of the three dynamic techniques alone, which improved run time by 48.0% over the baseline algorithm on average. Without this interpretation, the best combination of techniques would have been nogood recording and meta-value oracles, averaging a

37.5% run time improvement over the baseline algorithm. With my interpretation, the best combination of techniques is nogood recording and temporal bounds oracles, which improved run time by 57.7% over the baseline algorithm on average.

The dynamic techniques presented here operate on the premise that each problem in a dynamic sequence is similar to its predecessor. The effectiveness of each dynamic technique should therefore depend on the number of changes between the initial problem and the modified problem within each DDTP, as proposed in Hypothesis 3.6:

Hypothesis 3.6. The effectiveness of any combination of nogood recording, meta-value oracles, and/or temporal bounds oracles at improving efficiency will significantly decrease as the number of changes between the initial and modified problems of initially consistent DDTPs increases.

Figure 3.13 tests Hypothesis 3.6 by partitioning the randomly generated DDTPs into four sets according to the number of changes in each change set. Essentially, Figure 3.13 breaks each bar of Figure 3.12 into four bars, one for each different change set size. Figure 3.13 clearly shows that Hypothesis 3.6 is correct for this set of randomly generated initially consistent DDTPs—as the number of changes between the initial problem and the modified problem increases, the effectiveness of dynamic techniques at improving efficiency significantly decreases. In all but a few cases, the difference between the effectiveness of the techniques is not statistically significant when the number of changes differs by only 2, but if the number of changes differs by 4 or more, then the difference in the effectiveness of the techniques is statistically significant with

99.5% confidence.

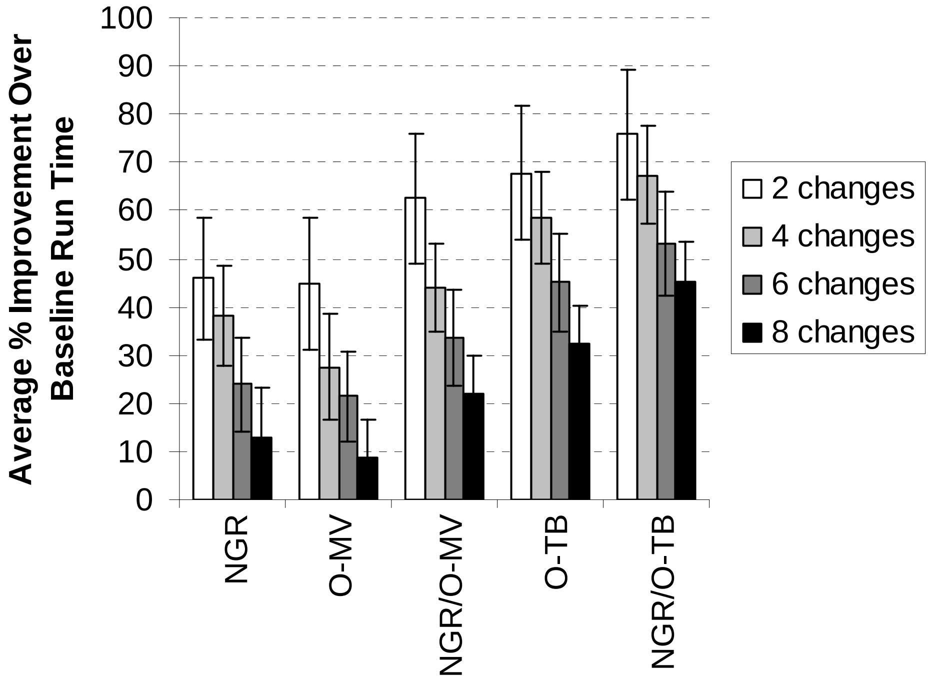

Figure 3.13. Average percent improvement over baseline run time using dynamic techniques to solve randomly generated initially consistent DDTPs, partitioned by the number of changes in each change set.

3.4.2. Testing Efficiency with Initially Consistent Air Traffic Control Problems

We have just seen how nogood recording and oracles compare when applied to randomly generated problems, but how will they compare when applied to structured, real-world problems? To answer this question, I created a second test set based on actual air traffic control recordings. In order to collect this data, I contacted Michelle Eshow and Gregory Wong at the NASA Ames Research Center, two of the developers of the CTAS Dynamic Planner, which is currently used to schedule incoming aircraft at major airports across the country. They were able to provide me with a wealth of domain knowledge along with a set of live air traffic control recordings. These recordings tracked all incoming aircraft arriving at the Dallas/Fort Worth International Airport for twenty-four hours a day over a period of seven days, from February 6th to February 12th, 2007. This raw data totaled over 4GB worth of text files.

To generate DDTPs based on this raw data, I first extracted a set of snapshots of the constraints. Each snapshot captures the complete set of all constraints on all incoming aircraft at a single moment in time. To find all of the constraints at a single moment, I traced the changes to the constraints every 30 minutes, and then copied all of the constraints that were active at that time to a separate file. The snapshots are separated by 30 minutes so they would represent sufficiently different problems. The original data files were divided by day, so I discarded all snapshots before 2:00am to avoid recording any partial constraint sets that were missing constraints that were added before midnight the day before. The majority of these snapshots contained constraints for 25 to 45 aircraft, so I also discarded any particularly small snapshots with fewer than 15 aircraft. This resulted in a set of 248 separate snapshots of air traffic control constraints.

To generate a DDTP, a snapshot was selected randomly, and all of the constraints that contained finite-domain variables were discarded. To complete the experiments in a reasonable amount of time, if the snapshot contained constraints for more than 31 aircraft, then the constraints of all but 31 randomly selected aircraft were also discarded. One of the remaining aircraft is randomly selected, separated from the rest, and set aside; this aircraft will be used to generate the change set, as will be described shortly. As I have pointed out previously, the air traffic control problems are so under-constrained that they can be solved with little to no backtracking. Removing all of the non-temporal constraints along with the constraints of some of the aircraft only exacerbates this issue.

To overcome this issue, I augmented each snapshot with simulated temporal constraints that impose ordering requirements between the runway arrival times of pairs of aircraft. Two aircraft are randomly selected, an ordering between them is chosen randomly, and a temporal difference constraint is created that requires the first aircraft to land before the second. If R_A is the runway arrival time of aircraft A, R_B is the runway arrival time of aircraft B, and R_A must land before R_B, then the temporal difference constraint that imposes this ordering is $R_A - R_B \leq 0$. This process is repeated to create a second temporal difference constraint that orders the runway arrival times of two independently selected aircraft, and these two constraints are disjoined to create a single disjunctive temporal constraint. The resulting disjunctive temporal constraint requires that the schedule satisfy at least one of the two ordering constraints between the pairs of aircraft.

It is important to keep in mind that the point of experimenting with real-world data is to compare the different techniques when applied to problems with structure. Hence, as long as the simulated constraints also have structure, it does not matter whether or not they seem to make sense in the context of an air traffic control problem. The simulated temporal constraints that I added to the problems are structured in several ways. First, the simulated constraints only restrict the times aircraft can land relative to one another; they do not restrict, for example, the time that one aircraft can pass its outer meter arc relative to the time that another aircraft can pass its final approach fix. Second, the simulated constraints are all ordering constraints, so the temporal bound of every simulated temporal difference constraint is 0. Relatively speaking, these simulated constraints are much more structured than the temporal constraints of the randomly generated problems, in which timepoints are chosen randomly from the complete set and bounds are chosen randomly from a range of values.

To control the tightness of the constraints of the DTPs based on air traffic control data, I performed a set of experiments to identify the number of simulated constraints per aircraft that should be added to achieve 20, 40, 60, and 80 percent consistency. These experiments mimicked those performed for the randomly generated problems. Given the combination of original and simulated temporal constraints, a static DTP is generated to represent the problem. In the air traffic control problem, it is necessary to schedule the time at which each aircraft will pass through each of four reference points, so the DTP contains four timepoints for each aircraft. With up to 30 aircraft represented in the constraint set, and with one timepoint to represent the temporal reference, the DTP could contain up to 121 timepoints. Time is measured in seconds, so I imposed a temporal horizon of 10,800 to force all aircraft in the airspace to land within three hours. After the DTP is generated, its consistency is tested; if it is inconsistent, then it is discarded, a new set of simulated constraints is generated, and the resulting DTP is tested again. This process repeats until a consistent DTP is found.

After finding a consistent initial DTP for the problem, I extended the static DTP to create a DDTP by adding a change set of restrictions (again, if an initially consistent problem is relaxed, the resulting modified problem must also be consistent, so I do not test this case). Rather than adding a set of randomly generated restrictions, the change set simulates a new aircraft entering the airspace. This is accomplished by adding all of the constraints of the randomly selected aircraft that was separated and set aside earlier. To maintain the number of simulated temporal ordering constraints per aircraft after this new aircraft has been added, several more simulated constraints are added with it. These simulated constraints are generated exactly as those of the initial DTP, except that one of the timepoints of one of the disjuncts is chosen randomly and replaced by the timepoint representing the runway arrival time of the new aircraft. This ensures that all of the new simulated constraints affect the new aircraft.

For each of the four consistency levels, I generated 100 initially consistent augmented air traffic control DDTPs, for a total of 400 problems in this set. By solving the problems of this set with each of the six combinations of nogood recording and oracles, I repeated the experiments that were performed using the randomly generated DDTPs in Section 3.4.1. Instead of graphing the relationship between each pair of combinations of dynamic techniques, I will only present the average percent improvement of each technique over the baseline run time in Figure 3.14.

By performing a series of Student's paired t-tests, I found that most of the hypotheses of Section 3.4.1 were once again confirmed with 99.5% confidence. There are, however, two notable exceptions. First, Hypothesis 3.3 suggested that temporal bounds oracles would outperform meta-value oracles. While this was true for the randomly generated problems, the opposite is true for the augmented air traffic control problems. Second, Hypothesis 3.4 suggested that meta-value oracles and temporal bounds oracles would both outperform nogood recording. For the air traffic control problems, meta-value oracles were more efficient than nogood recording, but temporal bounds oracles were not.

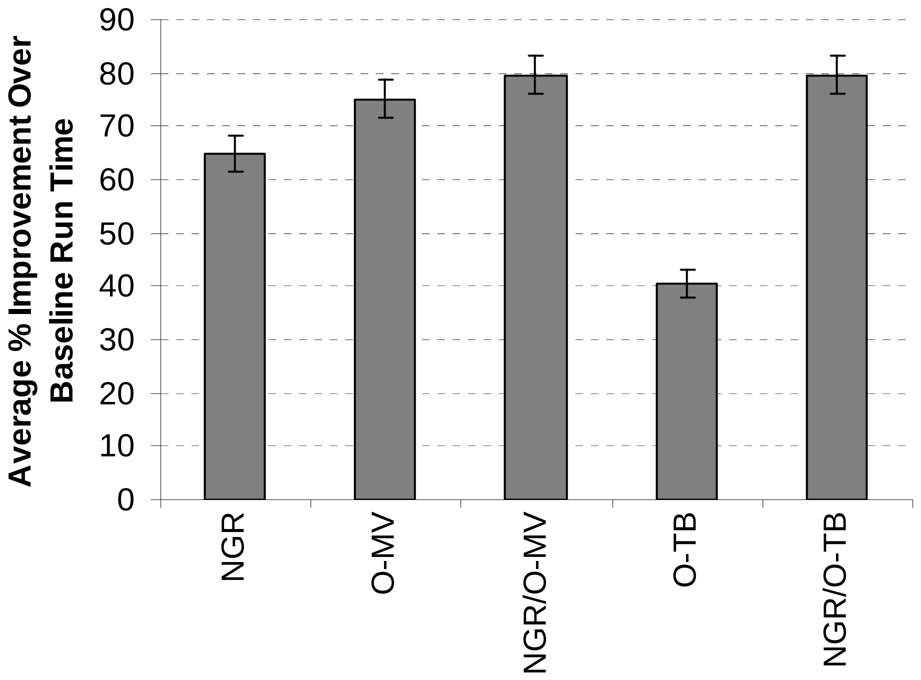

Figure 3.14. Average percent improvement over baseline run time using dynamic techniques alone and in combination to solve initially consistent air traffic control DDTPs.

Both of these discrepancies can be explained by the structure of the simulated constraints. The simulated temporal constraints are all *ordering* constraints, meaning that the temporal bounds are all 0. Basically, these constraints force the solver to find an acceptable partial ordering among all of the aircraft. The structure of these constraints has two effects. First, they allow for the quick discovery of small, and therefore powerful, nogoods. This increases the effectiveness of nogood recording; by comparing Figure 3.14 to Figure 3.12, it is easy to see that nogood recording was much more effective when applied to the air traffic control problems. Second, because it is possible to represent ordering constraints as meta-level finite-domain constraints, the relevant information of these simulated temporal constraints is stored in the meta-level constraints, not in the temporal bounds. This means meta-value oracles should be more effective than temporal bounds oracles when applied to air traffic control problems, and Figure 3.14 shows that this is indeed the case.

Interestingly, the average improvement of the combination of nogood recording and meta-value oracles is almost identical to the average improvement of the combination of nogood recording and temporal bounds oracles, despite the fact that the average improvement using meta-value oracles is over twice that of temporal bounds oracles. Although it is not completely clear why this happens, it is possible that a limit exists on the possible efficiency gains of the combination of oracles and nogood recording when applied to these augmented air traffic control problems, and that this limit has been reached.

Figure 3.14 clearly shows that nogood recording, meta-value oracles, and temporal bounds oracles can all significantly improve efficiency when applied to structured, real-world problems. For this problem set, meta-value oracles proved to be the most efficient of the dynamic techniques alone, improving run time by an average of 75.2% over the baseline algorithm. By combining meta-value oracles with nogood recording, the combination of techniques performed even better, improving run time by an average of 79.7% over the baseline algorithm. Compared to the results using the randomly generated problems, the dynamic techniques were able to improve efficiency significantly more when applied to structured air traffic control problems.

3.4.3. Testing Efficiency with Initially Inconsistent Randomly Generated Problems

To test the effectiveness of nogood recording and justification testing, I created a set of randomly generated initially inconsistent DDTPs. Like the previous set of randomly generated DDTPs, each DDTP of this set contains 10, 20, 30, or 40 timepoints, the tightness of constraints of each initial problem is 20%, 40%, 60%, or 80% consistency, and the number of changes between the initial problem and the modified problem is 2, 4, 6, or 8. Once again, I generated 20 random DDTPs for each combination of problem size, constraint tightness, and number of changes, resulting in a total of 1,280 randomly generated DDTPs in this set.

The process of generating each initially inconsistent DDTP is almost exactly the same as the process of generating the initially consistent DDTPs. The first difference is that static DTPs are generated, tested for consistency, and discarded until one is found that is inconsistent. The second difference is that each change of the change set is a relaxation; I do not test initially inconsistent problems with restrictions because the modified problem must always be inconsistent in this case. The type of relaxation is chosen randomly with uniform probability from the three types of DDTP relaxations.

1) **Loosen bound.** A temporal difference constraint is chosen randomly with uniform probability, and its bound is increased by a random amount chosen with uniform probability in the range [0, 100].
2) **Add disjunct.** A disjunctive temporal constraint is chosen randomly with uniform probability, and a new temporal difference constraint is generated in the same manner as in the initial problem and added to the disjunctive constraint.
3) **Remove constraint.** A disjunctive temporal constraint is chosen randomly with uniform probability and removed.

Using this set of 1,280 randomly generated initially inconsistent DDTPs, I compared four combinations of techniques. These combinations are as follows:

1) **Baseline.** Each DTP in the dynamic sequence is solved as if it were an unrelated static problem, without the use of nogood recording or justification testing. This same algorithm was used to test initially consistent DDTPs.
2) **NGR.** Nogoods that are learned while solving the initial problem are kept and applied when solving the modified problem; justification testing is not used. This same algorithm was used to test initially consistent DDTPs. Section 2.6.1 describes how to determine which nogoods can be applied after a problem is relaxed.
3) **JT.** Justification testing is used to move the variables of the justification of the initial problem to the beginning of the search order when solving the modified problem.
4) **NGR/JT.** Nogood recording and justification testing are used in combination.

Nogood recording proved to be effective when applied to initially consistent DDTPs. Nogood recording helps the solver avoid dead-end branches of the search tree; this process should improve efficiency whether the initial DTP is consistent or not, so I predict that nogood recording will once again outperform the baseline algorithm.

Hypothesis 3.7. Nogood recording will significantly improve efficiency over the baseline algorithm when applied to initially inconsistent DDTPs.

To test Hypothesis 3.7, Figure 3.15 compares the run time of each initially inconsistent randomly generated DDTP. Each plus sign plots the run time of using the baseline algorithm, and each point plots the run time using nogood recording. The problems are arranged in ascending order along the horizontal axis according to their run time using the baseline algorithm. As Figure 3.15 shows, the run time of these two algorithms is very similar. Although nogood recording is slightly faster on average, a Student's paired t-test cannot prove that the differences between these run times are statistically significant with any reasonable degree of confidence. For this set of initially inconsistent randomly generated DDTPs, Hypothesis 3.7 cannot be confirmed—nogood recording does not significantly improve efficiency over the baseline algorithm.

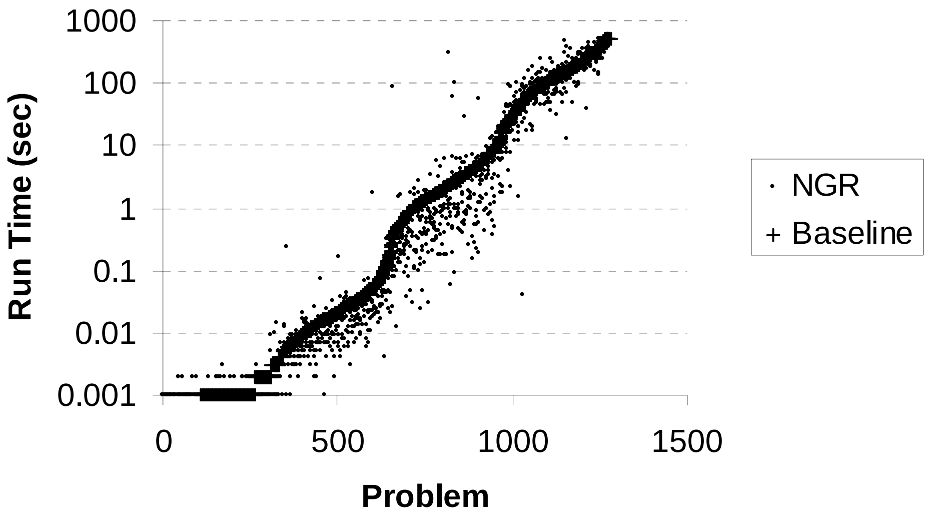

Figure 3.15. Run time of randomly generated initially inconsistent DDTPs using the baseline algorithm and nogood recording.

Considering that nogood recording significantly improves efficiency when applied to initially consistent DDTPs, this result is somewhat surprising. At present, I cannot offer a satisfactory explanation for this result. It is possible that the relaxations in the change sets affected a large portion of the recorded nogoods, so they had to be discarded and could have no effect on the run time of the modified problem. With such a small number of relaxations relative to the number of recorded nogoods, this explanation seems unlikely. It should be noted that nogood recording was originally designed with initially consistent problems in mind, and this is the first time nogood recording has ever been tested on problems that are initially inconsistent.

Unlike nogood recording, justification testing is specifically designed for use with initially inconsistent dynamic problems, so I predict that it will improve performance over the baseline algorithm.

Hypothesis 3.8. Justification testing will significantly improve efficiency over the baseline algorithm when applied to initially inconsistent DDTPs.

Figure 3.16 tests this hypothesis by comparing the run time of each initially inconsistent randomly generated DDTP. Each plus sign plots the run time of using the baseline algorithm, and each point plots the run time using justification testing. The problems are arranged in ascending order along the horizontal axis according to their run time using the baseline algorithm. At first glance, Figure 3.16 might not look much different from Figure 3.15. A closer inspection reveals that, in Figure 3.15, the bulk of problems for which nogood recording outperformed the baseline algorithm took below 10 seconds to solve; in Figure 3.16, the bulk of the problems for which justification testing outperformed the baseline algorithm took well over 10 seconds to solve. Because of the logarithmic scale on the vertical axis, this means the effects of justification testing are actually quite a bit larger than the effects of nogood recording. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.8 is correct for this set of randomly generated DDTPs—justification testing significantly improves performance compared to the baseline algorithm.

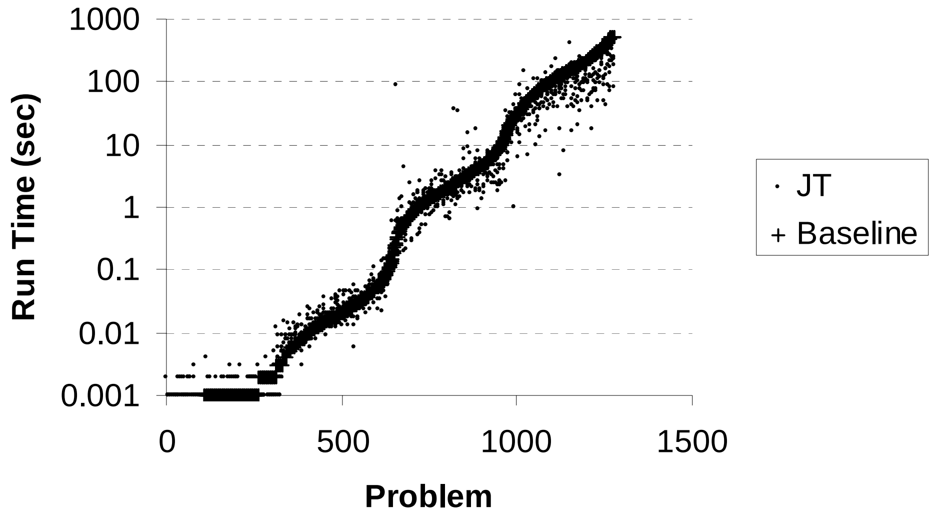

Figure 3.16. Run time of randomly generated initially inconsistent DDTPs using the baseline algorithm and justification testing.

Just as nogood recording improved performance when combined with oracles, it might also improve performance when combined with justification testing. If the variables of the justification really are more tightly constrained than the other variables as justification testing suggests, then a relatively large number of the nogoods recorded in the initial problem should constrain variables of the justification. Since justification testing moves these variables to the beginning of the search order, these nogoods can prune even more values while making assignments near the top of the search tree, so their effectiveness could be enhanced.

Hypothesis 3.9. The combination of nogood recording and justification testing will significantly improve efficiency over either nogood recording or justification testing alone when applied to initially consistent DDTPs.

To test this hypothesis, Figures 3.17 and 3.18 compare the run time of each initially inconsistent randomly generated DDTP using nogood recording and justification testing in combination against the run time using each technique alone. In both figures, each point plots the run time using both techniques together. In Figure 3.17, the problems are arranged in ascending order according to their run time using nogood recording alone, and these run times are plotted with plus signs. In Figure 3.18, the problems are arranged in ascending order according to their run time using justification testing alone, and these run times are plotted with plus signs. In both cases, a Student's paired t-test confirms with 99.5% confidence that, for this set of randomly generated DDTPs, Hypothesis 3.9 is correct—nogood recording and justification testing together are significantly more efficient than either technique alone. The fact that nogood recording improves performance when combined with justification testing but not by itself suggests that the reasoning behind Hypothesis 3.9 is correct.

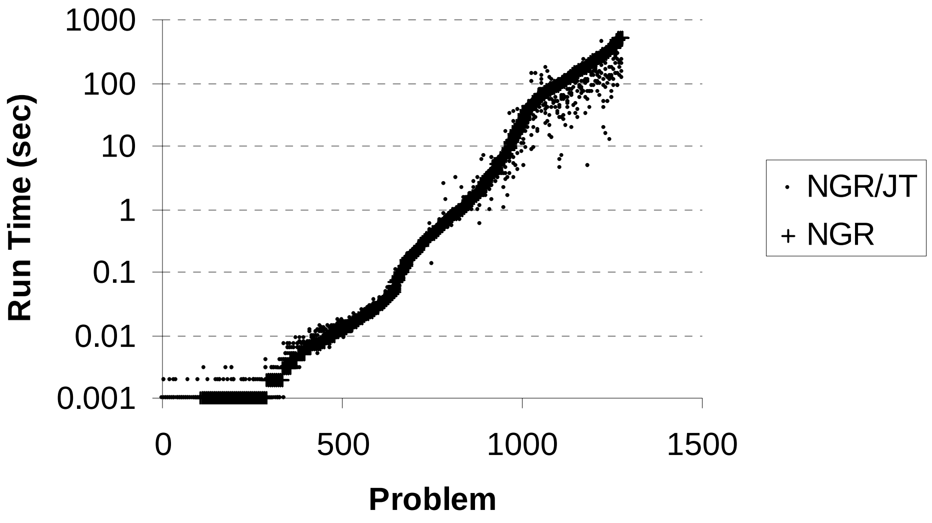

Figure 3.17. Run time of randomly generated initially inconsistent DDTPs using nogood recording alone and in combination with justification testing.

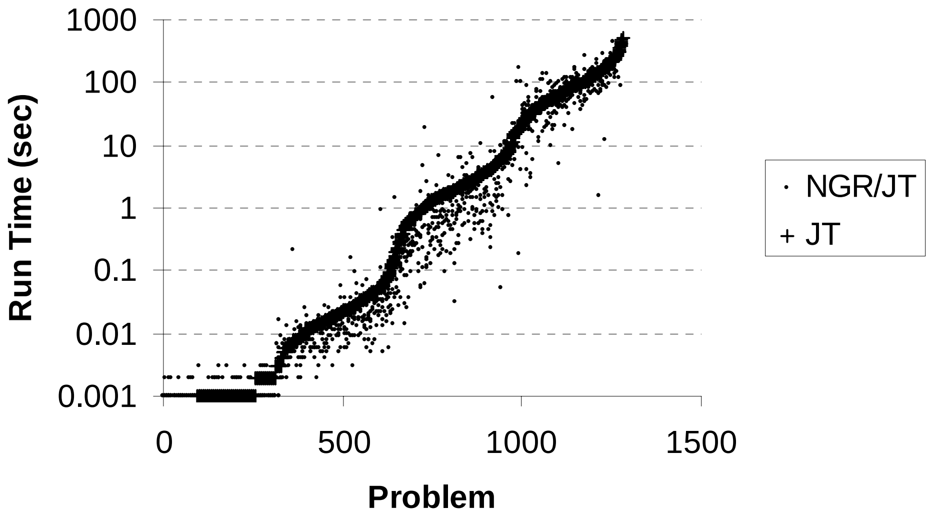

Figure 3.18. Run time of randomly generated initially inconsistent DDTPs using justification testing alone and in combination with nogood recording.

Figure 3.19 lends some perspective to the relative performance of these techniques alone and in combination by presenting their average percent improvement over the baseline run time. Nogood recording only improved run time by 3.4% on average; as pointed out, this difference was not statistically significant. Before my introduction of justification testing, nogood recording was the only technique available that might improve efficiency when solving initially inconsistent dynamic constraint problems. Justification testing improved run time by an average of 38.4% relative to the baseline. For this problem set, justification testing was an order of magnitude more effective at improving efficiency than nogood recording was. Finally, by combining nogood recording with justification testing, run time was reduced by 44.5% on average over the baseline. The fact that nogood recording did not significantly reduce run time over the baseline algorithm while it did significantly reduce run time over justification testing when the two techniques were used in combination indicates the existence of some form of positive interaction between nogood recording and justification testing, as suggested earlier when describing the intuition behind Hypothesis 3.9.

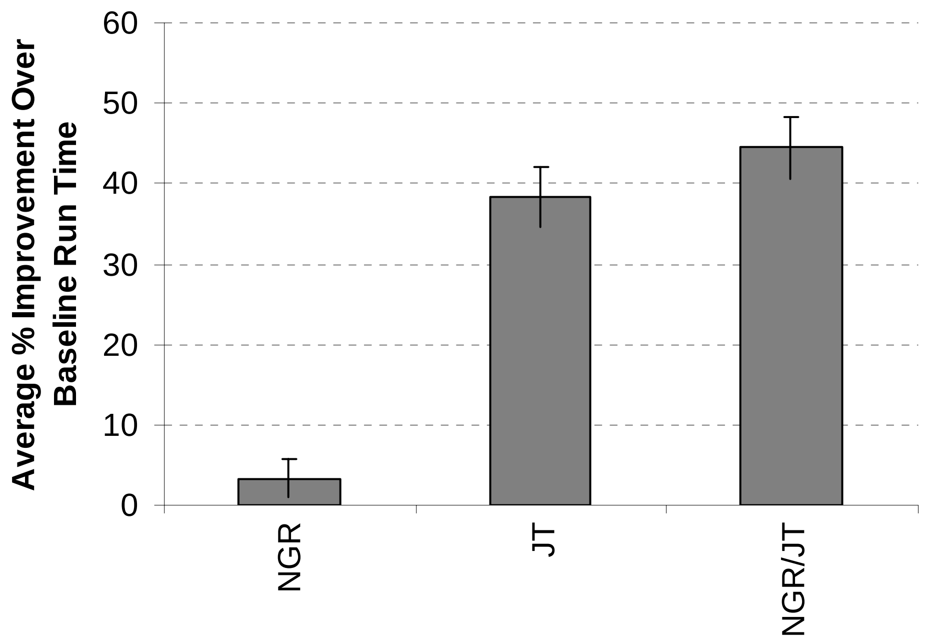

Figure 3.19. Average percent improvement over baseline run time using dynamic techniques alone and in combination to solve randomly generated initially inconsistent DDTPs.

Nogood recording and justification testing both operate on the premise that consecutive problems in a dynamic sequence are similar to each other. Hypothesis 3.10 suggests that the effectiveness of these techniques will decrease as the number of changes between the problems of a dynamic sequence increases:

Hypothesis 3.10. The effectiveness of nogood recording and justification testing, alone or in combination, at improving efficiency will significantly decrease as the number of changes between the initial and modified problems of initially inconsistent DDTPs increases.

Figure 3.20 tests this hypothesis by splitting each bar of Figure 3.19 into four separate bars, one for each change set size in this set of problems. As can be seen in Figure 3.20, the trend suggested by Hypothesis 3.10 holds true for nogood recording, but not for justification testing (alone or in combination with nogood recording). Although the results from this data set do not show it, it still seems reasonable that with enough changes in a single change set, the effectiveness of justification testing should decrease, as with the other dynamic techniques.

3.4.4. Testing Efficiency with Initially Inconsistent Air Traffic Control Problems

Once again, it is important to consider how these dynamic techniques will affect performance when applied to structured, real-world problems. Generating a set of initially inconsistent air traffic control problems is much like generating the set of initially consistent problems. First, a snapshot of constraints is selected randomly. All of the constraints that contain any finite-domain variables are removed, as are the constraints of all but 30 of the aircraft. Simulated temporal ordering constraints are added to achieve the desired percent consistency, and the resulting DTP is tested for consistency. If the DTP is consistent, then the process repeats until an inconsistent DTP is found.

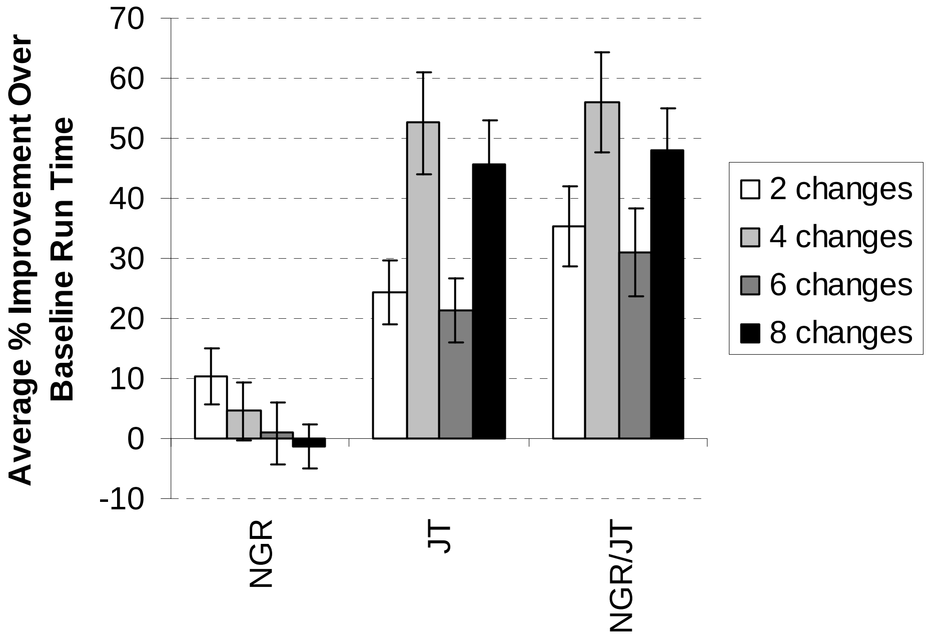

Figure 3.20. Average percent improvement over baseline run time using dynamic techniques to solve randomly generated initially inconsistent DDTPs, partitioned by the number of changes in each change set.

To create a change set that relaxes the problem, I simulate one of the aircraft leaving the airspace. That is, one aircraft is randomly selected, and the removal of each constraint associated with that aircraft is added to the change set. The resulting problem is an initially inconsistent DDTP. Using this process, I generated 100 problems at 20, 40, 60, and 80 percent consistency, for a total of 400 initially inconsistent augmented air traffic control DDTPs.

As in Section 3.4.2, I will skip the presentation of each pair-wise comparison of techniques and present the average percent improvements over the baseline run time, which are shown in Figure 3.21. As is clearly evident in this figure, the relative performance of nogood recording and justification testing is very different when they are applied to air traffic control problems compared to when they are applied to randomly generated problems. For randomly generated problems, justification testing significantly improved performance while nogood recording did not. For air traffic control problems, the opposite is true—nogood recording significantly improved performance while justification testing did not. Additionally, the average percent run time improvement using nogood recording and justification testing falls squarely in between the average improvement using either technique alone, indicating that there is no positive interaction like we saw for the randomly generated problems. For this set of air traffic control problems, the difference between run times for all pairs of techniques was statistically significant with 99.5% confidence according to a Student's paired t-test with the exception of the difference between the run times of the baseline algorithm and justification testing.

Why does nogood recording perform so much better on air traffic control problems while justification testing performs so much worse? As pointed out in Section 3.4.2, the simulated temporal constraints are all ordering constraints, which allow for the quick discovery of small and powerful nogoods; this explains why nogood recording performs better on the air traffic control problems. As for why justification testing performs worse, note that these initially inconsistent DDTPs are relaxed by removing all constraints related to a single aircraft. In the randomly generated problems, the DDTPs are relaxed by removing 2, 4, 6, or 8 randomly selected constraints, which is a much smaller percentage of the total number of constraints in the problem. Since removing all constraints related to an aircraft is a relative large change, the assumption that the variables of the justification are constrained more tightly than the other variables, which is central to the success of justification testing, is less likely to hold true.

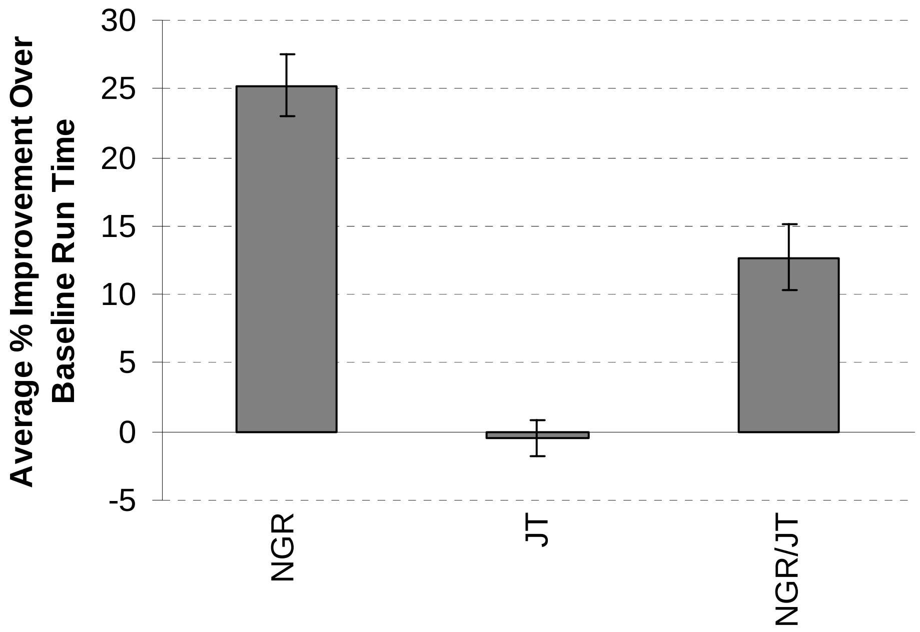

Figure 3.21. Average percent improvement over baseline run time using dynamic techniques alone and in combination to solve initially inconsistent air traffic control DDTPs.

Why do nogood recording and justification testing interact positively on randomly generated problems but not on air traffic control problems? In Section 3.4.3, I explained that nogood recording and justification testing might interact positively because the variables near the top of the search tree should be more tightly constrained than the other variables, and these constraints should be made explicit by the nogoods, thus maximizing pruning where it will be most effective. This positive interaction is dependent on the assumption that the variables of the justification are more tightly constrained than the other variables, which is apparently not the case for this set of augmented air traffic control problems.

3.5. DDTP Stability

While efficiency is certainly a central concern when producing a schedule, stability is also important in many problem domains. Stability is a measure of the similarity between the solutions of a dynamic sequence. There are several reasons why stability might be important in a particular application. In the air traffic control domain, for example, the controllers can quickly become frustrated by frequent dramatic changes to the schedule. The approach taken by the controllers is to impose STA freezes and sequence freezes that lock aircraft into a schedule or sequence as they approach reference points. This approach, however, makes it impossible to optimize the schedule of frozen aircraft, and does nothing to improve the stability of the rest of the schedule. In other domains, there may be a cost incurred that is relative to the amount of change between solution schedules. For example, the changes to the schedule might need to be communicated to a set of agents, or some additional processing might be required before the new schedule can be used.

A general definition of stability would be a measure of the total amount of similarity over all solutions in a sequence—that is, an aggregate measure of the similarity between *two or more solutions*. In keeping with similar research on stability (e.g., Dechter and Dechter 1988; Verfaillie and Schiex 1994; Fox, *et al.* 2006), I will present metrics of stability that measure the pair-wise similarity between *exactly two solutions*. This is the same simplification that was laid out at the end of Section 3.1. To demonstrate the difference between the general definition and my simplification, consider a sequence of three solutions. With only pair-wise measures of stability, the third solution will be compared to the second, but any similarity (or dissimilarity) between the third solution and the first solution will be ignored. An aggregate measure of temporal stability would compare third solution to both the second and the first. This might make it easier for a human user who to keep up with changes to the schedule, or it might allow an agent to reuse information learned while reasoning about the first schedule when it is later reasoning about the third. Although this is an interesting and potentially useful measure in its own right, it is beyond the scope of this thesis.

As just pointed out, the stability metrics I present will not measure a property of more than two solutions; additionally, they will not measure a property of less than two solutions. Robustness and flexibility are related notions that measure properties of a *single solution*. *Robustness* (Weld, Anderson, and Smith 1998; Bian, *et al*. 2005; Fox, Howey, and Long 2006) is a measure of the ability of a solution to resist the need for change in the face of uncertain future events. Robustness is related to stability in that a robust solution is unlikely to require modification even when the problem has changed, so it is likely to produce a stable sequence of solutions (if two solutions are identical, then the sequence of the two is maximally stable). *Flexibility* (Kramer and Smith 2003; Policella, *et al.* 2003; Policella, *et al.* 2004) is a measure of the number of solutions encapsulated within a solution set. A flexible solution leaves some details unspecified, so it actually defines a set of fully specified solutions. Flexibility is related to robustness in that if a solution is flexible, then there is less chance that a small change will invalidate all of the fully specified solutions within the set, so a flexible solution is generally also robust (and therefore stable). I will leave measures of schedule robustness, schedule flexibility, and aggregate temporal stability for future work.

Now that we have distinguished stability from similar concepts, how should stability be measured in the context of a dynamic DTP? In their original work on dynamic CSPs, Dechter and Dechter (1988) measure stability as the Hamming distance between two subsequent solution assignments, as do Verfaillie and Schiex (1994). Similarly, although not in the context of dynamic problems, Hamming distance has also been used to measure the similarity between multiple solutions to a CSP (Hebrard, *et al*. 2005) or a planning problem (Srivastava, *et al*. 2007). Fox, *et al.* (2006) use a counting method similar to Hamming distance when comparing the stability that results from replanning versus plan repair in a dynamic planning problem. Hamming distance makes sense in situations such as these when the variable domains are unordered. In such situations, there is no inherent measure of the similarity between values within the variable domains. In a map coloring problem, for example, there is no inherent reason to believe that the similarity between red and blue is any greater than the similarity between red and green. If a distance metric were defined between pairs of values, then Hamming distance would not take this distance metric into account.

One possible measure of temporal stability within a DDTP is the Hamming distance between the meta-level assignments of subsequent solutions. While this measure is simple and straightforward, it completely ignores the temporal aspect that is central to a DDTP: namely, in a DDTP, the variable domains are ordered, so there is an inherent measure of similarity between values. For example, moving a meeting from

1:00pm to 2:00pm is a smaller change than moving it to 3:00pm; this distinction would be lost to a temporal stability metric based on Hamming distance.

A second possible measure of temporal stability within a DDTP would be to generate and compare exact solutions to each DTP in the dynamic sequence. Recall from Section 2.4.2 that a solution to a DTP is a consistent component STP, whereas an *exact* solution to a DTP is an assignment of a time to each timepoint. As pointed out in that section, it is often preferable for a DTP solver to produce a solution STP rather than an exact solution because the solution STP offers greater flexibility (as defined above). While finding an exact solution to each DTP within a DDTP would require only a minimal (polynomial) amount of extra work, doing so solely to accommodate the stability measure would eliminate the flexibility offered by the solution STPs.

3.5.1. Temporal Stability Metrics

Rather than comparing meta-level solutions or exact solutions, a third approach to a temporal stability measure for DDTPs is to compare solution STPs directly. By comparing solution STPs, a measure of temporal stability can capture the temporal relationships that might be missed by comparing meta-level solutions while preserving the flexibility that would be lost by comparing exact solutions. To do this, the two solution STPs being compared should be put into a canonical form. The canonical form of an STP that I use is the transitive closure of its constraints represented as a d-graph. When represented as d-graphs, each solution of a DDTP has exactly the same number of constraints, and if two solutions represent the same set of induced constraints, then their d-graphs will be identical.

Using d-graphs as the canonical form of STPs, I have defined a family of temporal stability metrics. Since stability is a measure of the similarity between subsequent solutions, maximizing stability is equivalent to minimizing the difference between subsequent solutions. I therefore define this family of stability measures in terms of *temporal distance*. A temporal distance metric is a function that maps a pair of consecutive solution STPs in a DDTP to a single value, such that a lower value corresponds to greater stability. Recall from Definition 3.2 that if D_i and D_{i+1} are the d-graphs of consecutive solutions S_i and S_{i+1} in a DDTP, then $dd[x, y] = D_i[x, y] - D_{i+1}[x, y]$. Each temporal distance metric in the family I have defined takes on the following form:

$$\textbf{TemporalDistance}(S_i, S_{i+1}) = \oplus_{x, y \in T} \otimes(dd[x, y]) \qquad (3.1)$$

In Formula 3.1, $\otimes$ is a *local distance metric* that imposes a penalty for the amount of change on the induced bound between a single pair of timepoints, and $\oplus$ is a *global distance metric* that aggregates the local distance metrics into a single value.

A local distance metric $\otimes$ could be any function, but what functions make sense in the context of temporal stability? Given a DDTP, let T be the set of timepoints, and let S_0 be a solution STP to the initial DTP in the sequence. Let S_A and S_B be two possible solution STPs to the modified DTP. Let D_0, D_A, and D_B be the transitive closures of S_0, S_A, and S_B, respectively, represented as d-graphs. Let D_A and D_B be identical for all pairs of timepoints except (v, w)—that is, $D_A[v, w] \neq D_B[v, w]$. Depending on the relationships between D_0, D_A, and D_B, which solution schedule is preferred, S_A or S_B?

If the goal is to measure temporal distance, then the local distance metric should increase with the amount of change between the schedules. The simplest function would be a direct measure of the amount of change on the induced bound on a pair of timepoints from one schedule to the next:

$$\otimes(dd[x, y]) = dd[x, y] \tag{3.2}$$

Formula 3.2 is straightforward, but it is important to consider all of the implications when choosing a local distance metric for a particular application. According to Formula 3.2, if the bound on the difference between the timepoints increases, then the local distance value is negative. Depending on the global distance metric, a negative local distance value could decrease the overall temporal distance. Assume that the bound on $w - v$ has been relaxed in both S_A and S_B—that is, $D_A[w, v] > D_0[w, v]$ and $D_B[w, v] > D_0[w, v]$. Furthermore, assume that the bound on $w - v$ has been relaxed more in S_A—that is, $D_A[w, v] > D_B[w, v]$. In this case, S_A is preferred to S_B according to Formula 3.2, even though S_A is actually less similar to S_0.

In some applications, this is not the desired behavior of a local distance metric. Instead, it might be desired that the local distance metric increases with the dynamic distance, whether the dynamic distance is positive or negative. For example, if some post-processing optimization was performed based on the initial schedule, then the result might be suboptimal whether the induced bound was restricted or relaxed. In such a case, Formula 3.3 can be used:

$$\otimes(dd[x, y]) = |\, dd[x, y] \,| \qquad (3.3)$$

According to Formula 3.3, S_B is preferred to S_A in the above example because it is more similar to S_0.

In other applications, the desired behavior of the local distance metric might be to increase only when the bound on a pair of timepoints has been restricted. For example, the schedule S_0 might be used in a multi-agent context in which it must be integrated with the schedules of other agents. If the other schedules were produced based on the constraints of S_0, then as long as none of the constraints of S_0 are tightened, then none of the other schedules need to be updated. In this situation, the appropriate local distance metric would return a positive value when a temporal bound has been restricted and return 0 otherwise, as in Formula 3.4:

$$\otimes(dd[x, y]) = max(0, dd[x, y]) \qquad (3.4)$$

Of course, many other local distance metrics are possible. Several possibilities are listed here, including a weighted linear function (Formula 3.5), a polynomial function (Formula 3.6), an exponential function (Formula 3.7), and a logarithmic function (Formula 3.8); in each of these examples, *c* is a constant. In fact, there is no reason why a different local distance metric could not be applied to the bound of every different pair of timepoints in the DDTP. The local distance metric that is appropriate will depend on the domain. The key is to choose one that applies the appropriate penalty given the amount of change on a temporal bound between solutions.

$$\otimes(dd[x, y]) = c(dd[x, y]) \tag{3.5}$$

$$\otimes(dd[x, y]) = (dd[x, y])^{c} \tag{3.6}$$

$$\otimes(dd[x, y]) = c^{dd[x, y]} \tag{3.7}$$

$$\otimes(dd[x, y]) = \log_{c}(dd[x, y]) \tag{3.8}$$

Once a local distance metric has been chosen, it is still necessary to choose a global distance metric. A global distance metric aggregates all of the individual local distance values into a single value that describes the overall temporal distance between the initial schedule and the modified schedule. While many aggregate functions are possible, two stand out as sensible choices for global distance metrics in a wide variety of problem domains, the maximum (Formula 4.9) and the average (Formula 4.10):

$$\oplus_{x, y \in T} \otimes(dd[x, y]) = max_{x, y \in T} \otimes(dd[x, y]) \tag{3.9}$$

$$\oplus_{x, y \in T} \otimes(dd[x, y]) = avg_{x, y \in T} \otimes(dd[x, y]) \tag{3.10}$$

The maximum makes sense as a global distance metric when the goal is to minimize the largest value of the local distance metrics, which often has the effect of distributing changes more evenly over the entire schedule. The average makes sense as a global distance metric when the distribution of changes does not matter. As with the local distance metric, the global distance metric that is appropriate will depend on the domain.

3.5.2. Testing Initial Stability with Randomly Generated Problems

After selecting a local and global distance metric, it is possible to measure the temporal distance between two consecutive solution schedules. Using such a temporal distance metric, it is possible to compare the performance of dynamic techniques on the dimension of stability. Stability is only measurable when two consecutive DTPs of a dynamic sequence are both consistent. Justification testing cannot be applied if the initial problem is consistent, so I compare the performance of nogood recording, meta-value oracles, and temporal bounds oracles, as in Section 3.4.1. Based on these dynamic techniques, I propose the following hypotheses:

Hypothesis 3.11. Meta-value oracles will significantly improve stability when compared to the baseline algorithm. That is, the temporal distance between solutions to subsequent problems should be significantly less using O-MV than using Baseline. By following at least part of the initial search path, the modified solution schedule should have some similarity to the initial solution schedule.

Hypothesis 3.12. Temporal bounds oracles will significantly improve stability when compared to the baseline algorithm or meta-value oracles. That is, the temporal distance using O-TB should be significantly less than either O-MV or Baseline. A meta-value oracle chooses the same meta-values as in the initial solution, which may or may not correspond to a similar solution schedule; a temporal bounds oracle chooses the meta-values that minimize temporal distance, which should more directly lead to a similar solution schedule.

Hypothesis 3.13. Nogood recording will have no significant effect on stability, no matter which other dynamic techniques it is combined with. That is, there should be no significant difference in temporal distance between NGR and Baseline, NGR/O-MV and O-MV, or NGR/O-TB and O-TB. Nogood recording only guides the search away from branches of the search tree that are known to be dead ends; it does not guide the search toward any particular solution.

I compared the stability performance of these combinations of techniques using the same 1,280 initially consistent randomly generated DDTPs that were used to compare efficiency in Section 3.4.1. From this problem set, the modified problem is consistent in only 846 of the DDTPs; stability cannot be measured for the other problems, so they are ignored. For the purposes of comparison, it is necessary to define a temporal distance metric. As an example, I apply Formula 3.4 as a local distance metric and Formula 3.9 as a global distance metric:

$$\mathbf{TemporalDistance}(S_i, S_{i+1}) = max_{x,\, y \in T}\ (\ max(0, dd[x, y])\ ) \qquad (3.11)$$

To illustrate my analysis of Hypothesis 3.11, Figure 3.22 compares the temporal distance between the initial and modified solution of each randomly generated DDTP using the baseline algorithm and using meta-value oracles. Each point on the horizontal axis represents a single problem, sorted in order from the smallest to largest temporal distance using the baseline algorithm. The vertical axis reports the temporal distance of each DDTP as measured by Formula 3.11. Each plus sign plots the temporal distance using the baseline algorithm, forming a sort of wavy backbone from the lower left to the upper right of the graph (because the problems are sorted according to the baseline algorithm). I discuss reasons for the wavy shape of this backbone below. Each dot plots the temporal distance using meta-value oracles. For the majority of problems, using a meta-value oracle produces a solution with lower temporal distance than does the baseline algorithm. A Student's paired t-test confirms with 99.5% confidence that this difference is statistically significant. This shows that Hypothesis 3.11 is correct for this set of randomly generated DDTPs—meta-value oracles significantly improve stability when compared to the baseline algorithm.

Recall that each DDTP of this randomly generated set has a temporal horizon of 1000, meaning that the induced bound between any pair of timepoints cannot exceed this limit. Furthermore, no induced bound can be tightened below -1000, as this would create a negative cycle. The largest possible change in induced bounds between subsequent solutions is a restriction from 1000 to -1000, which means that the largest possible temporal distance between subsequent solution schedules in this set of randomly generated DDTPs is 2000. In general, the maximum possible temporal distance as measured by Formula 3.11 is twice the temporal horizon.

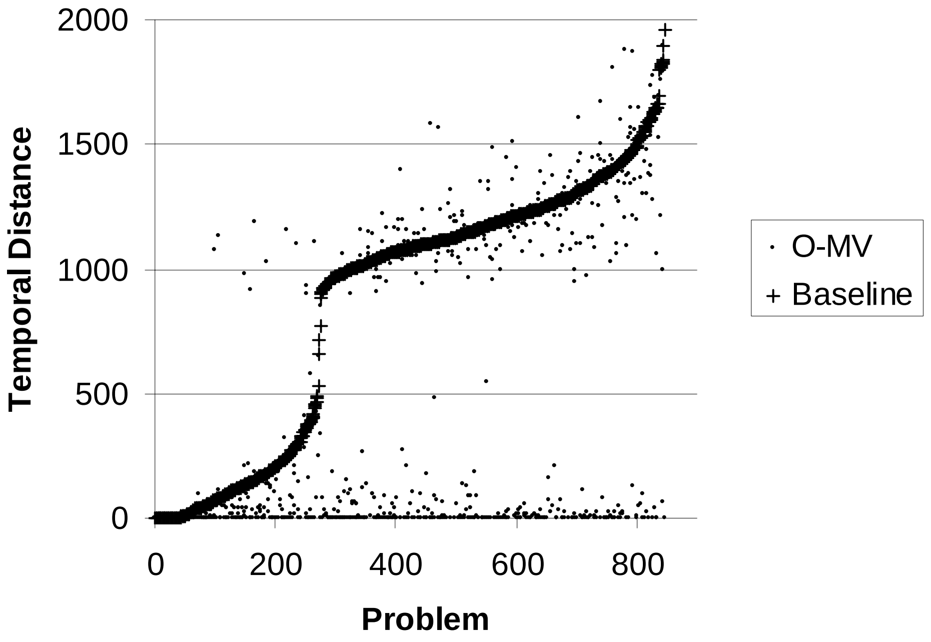

Figure 3.22. Temporal distance between DDTP solutions using the baseline algorithm and meta-value oracles.

In Figure 3.22, note that the temporal distance of many of the problems falls between 0 and 500 and between 1000 and 1500, whether using the baseline algorithm or meta-value oracles. In this problem set, it is common for the initial solution schedule to contain several pairs of timepoints that are unconstrained. For each of these unconstrained pairs, the induced bound is 1000 because that is the temporal horizon. If one of these pairs then becomes constrained in the modified solution schedule, then the temporal distance between the two solutions must be greater than or equal to the amount by which this induced bound has been restricted. If several of these pairs become constrained in the modified solution, then it is likely that at least one of the induced bounds has dropped below 0, meaning that the temporal distance is slightly above 1000. On the other hand, if none of these pairs of timepoints that are unconstrained in the initial solution become constrained in the modified solution, then the temporal distance will be slightly above 0. It is for this reason that, in general, many of the temporal distances as measured by Formula 3.11 will fall either slightly above 0 or slightly above the temporal horizon. This leads to the wavy shape of the baseline temporal distances observed in Figure 3.22.

According to Hypothesis 3.12, using temporal bounds oracles will result in greater stability than using either the baseline algorithm or meta-value oracles. Figure 3.23 shows the results of my tests of the first part of this hypothesis, comparing the temporal distance of each DDTP using the baseline algorithm and temporal bounds oracles. Figure 3.24 shows the tests for the second part of the hypothesis, comparing meta-value oracles and temporal bounds oracles. Student's paired t-tests confirm that Hypothesis 3.12 is correct with 99.5% confidence for these randomly generated DDTPs—temporal bounds oracles produce solutions with greater stability than either the baseline algorithm or meta-value oracles.

Hypothesis 3.13 suggests that nogood recording will have no significant effect on stability, no matter which other dynamic techniques it is combined with. Figure 3.25 compares the temporal distance of each DDTP using the baseline algorithm and nogood recording, Figure 3.26 compares the temporal distance of meta-value oracles alone and in combination with nogood recording, and Figure 3.27 compares the temporal distance of temporal bounds oracles alone and in combination with nogood recording. As can be seen in each of these figures, the effect of adding nogood recording is relatively random—it reduces temporal distance for some problems and increases it for others. Student's paired t-tests confirm with 99.5% confidence that the overall effect of nogood recording is statistically insignificant in each of these cases. Hypothesis 3.13 is thus confirmed for this set of randomly generated DDTPs—nogood recording has no significant effect on stability, no matter which other dynamic techniques it is combined with.

Figures 3.22 through 3.27 do not show the scale to which each dynamic technique improves stability, so Figure 3.28 shows the average percent improvement of each combination of dynamic techniques over the temporal distance using the baseline algorithm. As before, the error bars represent standard error. Compared to the baseline, meta-value oracles reduced temporal distance by an average of 45.5%, and temporal bounds oracles. This means that using oracles improves stability considerably by this measure. Temporal bounds oracles improve stability more than meta-value oracles do, but the difference is not quite so dramatic. As expected, the effect of combining nogood recording with any of the other dynamic techniques is negligible.

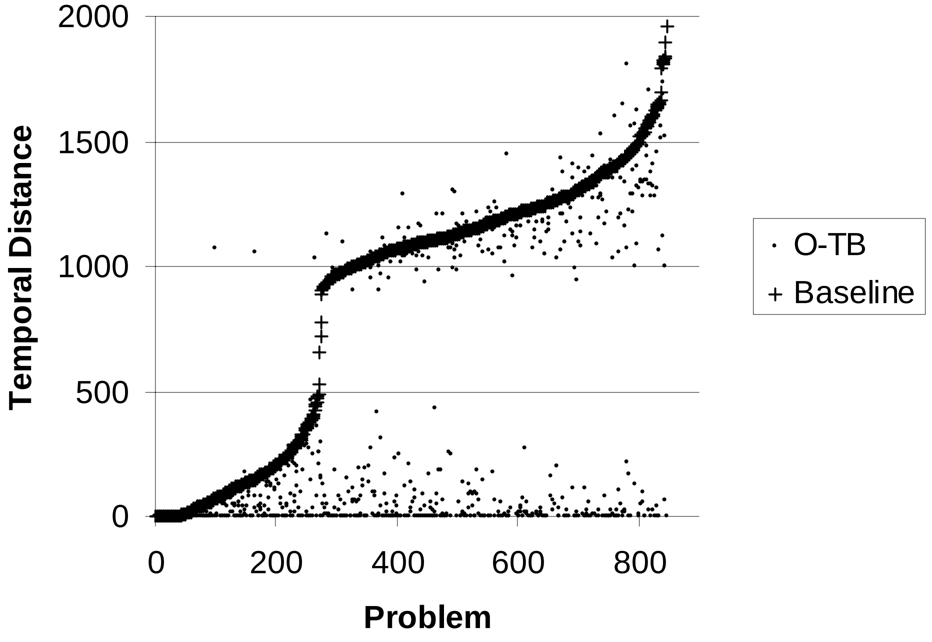

Figure 3.23. Temporal distance between DDTP solutions using the baseline algorithm and temporal bounds oracles.

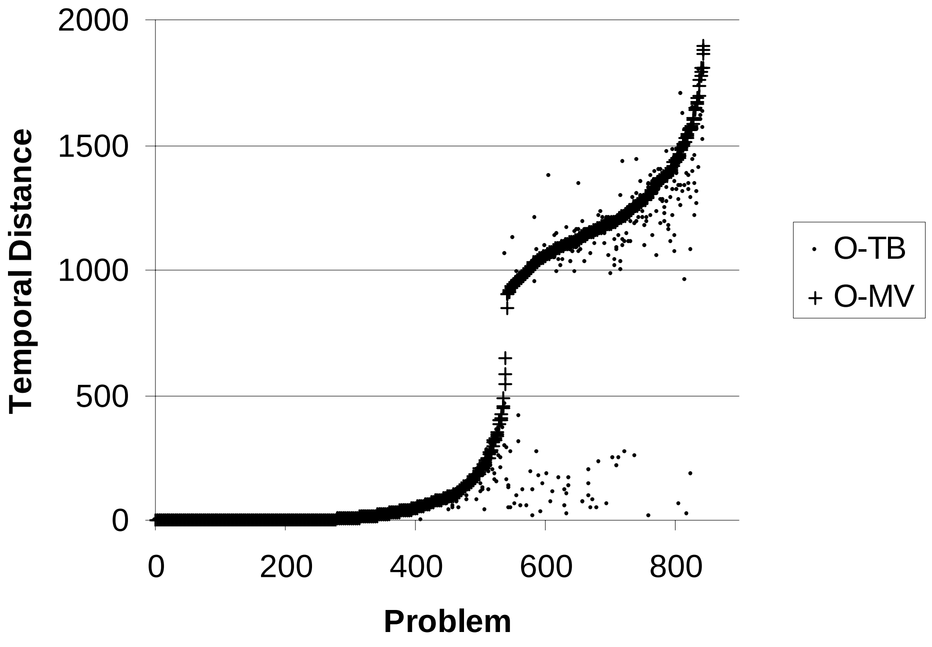

Figure 3.24. Temporal distance between DDTP solutions using meta-value oracles and temporal bounds oracles.

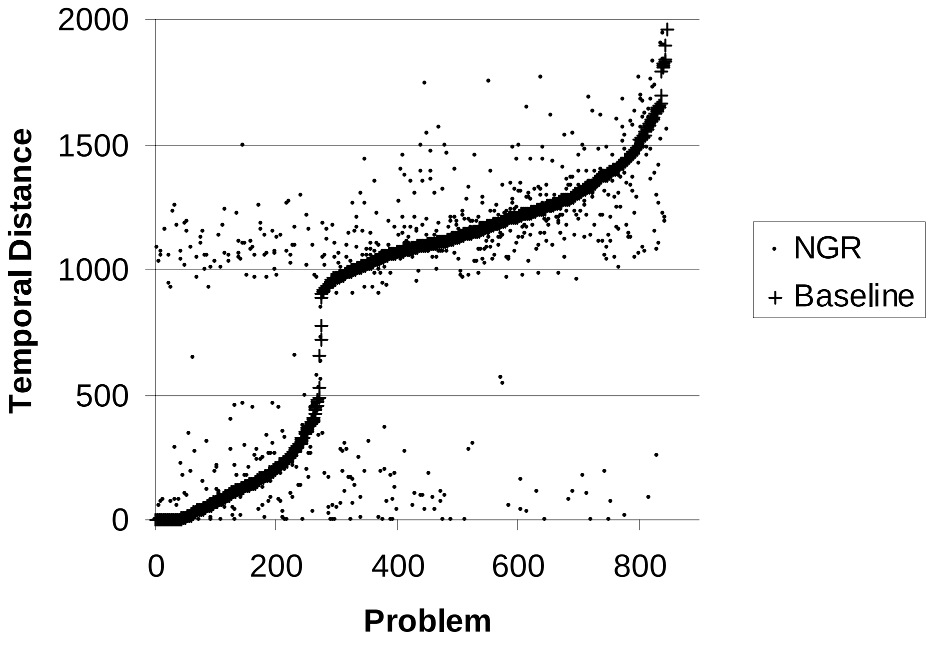

Figure 3.25. Temporal distance between DDTP solutions using the baseline algorithm and nogood recording.

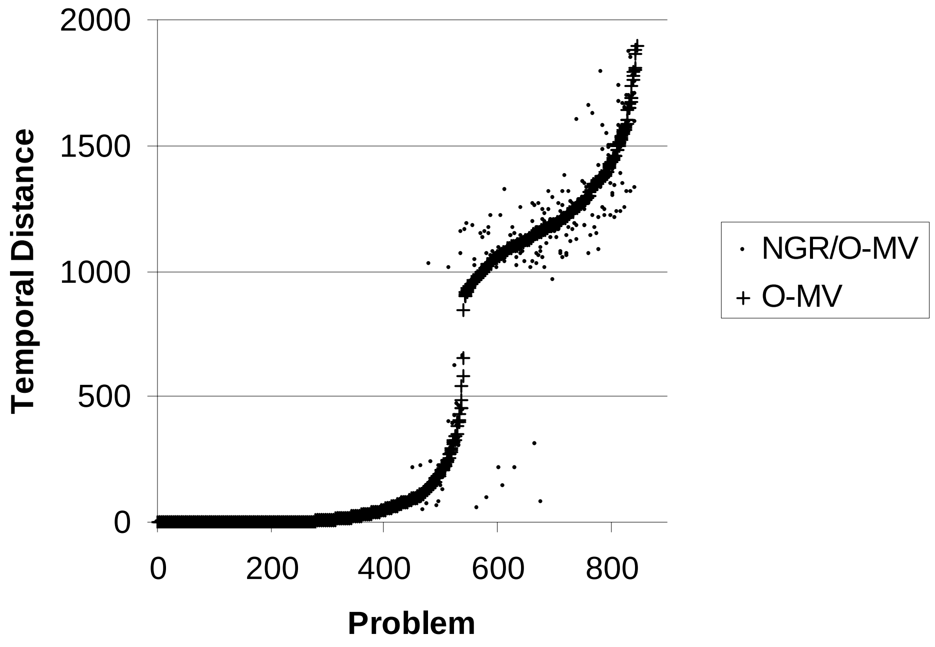

Figure 3.26. Temporal distance between DDTP solutions using meta-value oracles and the combination of meta-value oracles and nogood recording.

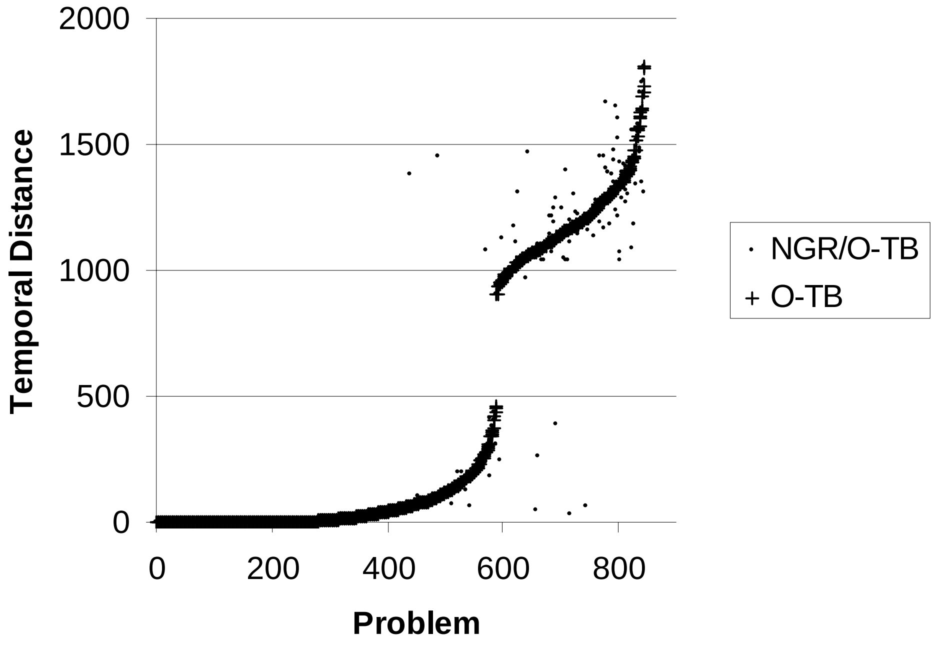

Figure 3.27. Temporal distance between DDTP solutions using temporal bounds oracles and the combination of temporal bounds oracles and nogood recording.

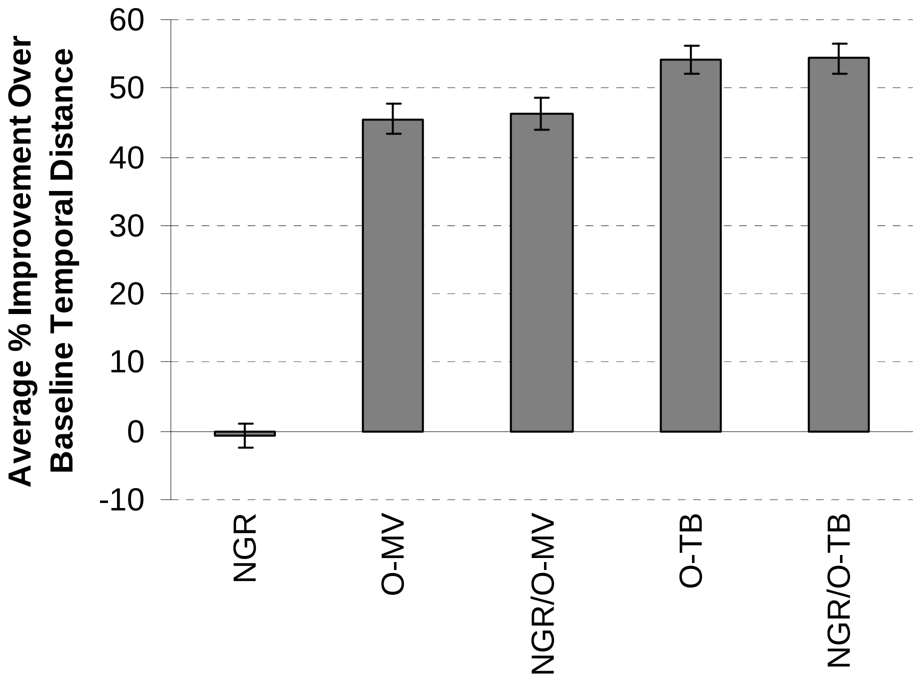

Figure 3.28. Average percent improvement over baseline temporal distance using combinations of dynamic techniques when solving randomly generated initially consistent DDTPs.

In Section 3.4.1, we saw that as the number of changes between the initial and modified problems increases, the ability of dynamic techniques to improve efficiency in initially consistent DDTPs decreases. Meta-value oracles and temporal bounds oracles are effective at reducing temporal distance because they guide the solver toward a similar solution, and these techniques operate on the premise that each problem in a dynamic sequence is similar to its predecessor. It is therefore reasonable to believe that the ability of oracles to reduce temporal distance should decrease as the number of changes between the initial and modified problems increases. As just shown above, nogood recording has no discernable effect on temporal distance, so it seems reasonable to believe that the number of changes in the change set should have no effect on the ability of nogood recording to reduce temporal distance.

Hypothesis 3.14. The effectiveness of meta-value oracles and temporal bounds oracles at reducing temporal distance will significantly decrease as the number of changes between the initial and modified problems of initially consistent DDTPs increases. The number of changes between the initial and modified problems will have no effect on the ability of nogood recording to reduce temporal distance.

To test this hypothesis, Figure 3.29 partitions the randomly generated DDTPs into four sets according to the number of changes in each change set. According to Student's paired t-tests, for each technique except nogood recording, the amount by which the average percent improvement over the baseline temporal distance decreases when the number of changes increases by 2 is statistically significant with 99.5% confidence. Hypothesis 3.14 is therefore confirmed—the effectiveness of oracles at reducing temporal distance decreases as the number of changes per change set increases, but the effectiveness of nogood recording does not.

3.5.3. Testing Stability Optimization with Randomly Generated Problems

Measuring the temporal distance between the initial solution and the modified solution is just one way to compare the effect of each combination of dynamic techniques on stability. A second measurement is the amount of time required to find a solution with optimal stability. As pointed out in Section 2.2.4, the standard backtracking search algorithm can be converted into an optimization algorithm by simply continuing to search for better solutions until the search space is exhausted.

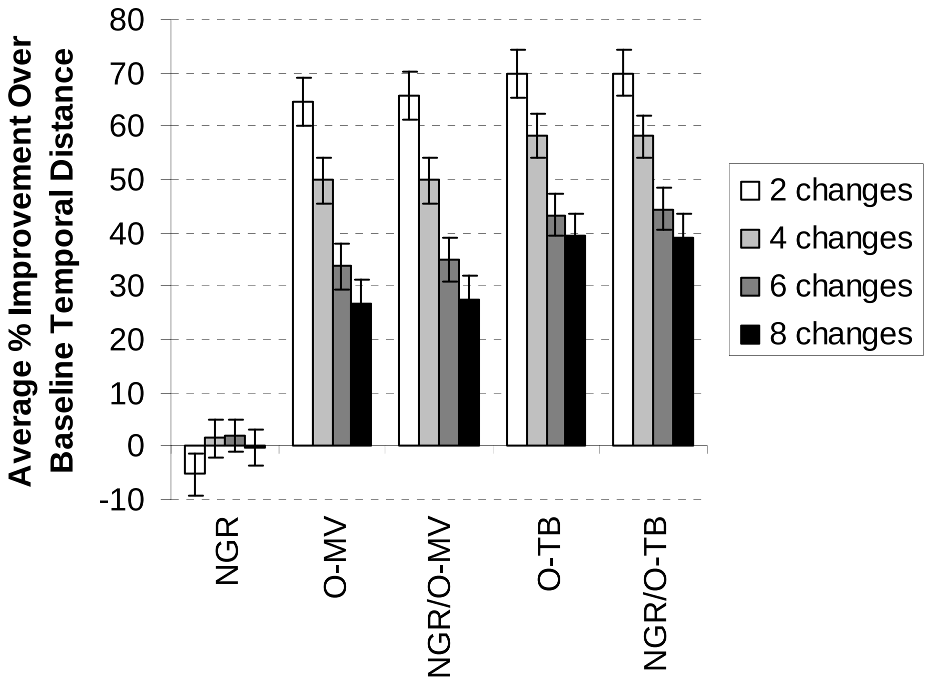

Figure 3.29. Average percent improvement over baseline temporal distance using dynamic techniques to solve randomly generated initially consistent DDTPs, partitioned by the number of changes in each change set.

Depending on the choice of local and global distance metrics that are applied, it is possible that the resulting measure of temporal distance will be monotonically non-decreasing relative to the depth of the search tree while searching for a solution to the modified problem. Recall from Section 2.2.4 that a preference function is monotonically non-decreasing iff for any node *n* in the search tree, the preference values of all nodes in the subtree rooted at *n* are no lower than the preference value at *n*.

A temporal distance metric ranks the possible solutions of the modified DTP according to their similarity to the solution of the initial DTP, so a temporal distance metric is a form of preference function. A temporal distance metric is monotonically non-decreasing if, after extending the meta-level assignment, it is impossible for the temporal distance to decrease. For example, any combination of the local distance metrics defined in Formulas (3.3) or (3.4) and the global distance metrics defined in Formulas (3.9) and (3.10) is monotonically non-decreasing. If the temporal distance metric is monotonically non-decreasing, then the efficiency of the stability optimization algorithm can be greatly increased using branch and bound and valued nogoods to prune parts of the search space.

Given an implementation of this stability optimization algorithm with branch and bound and valued nogoods, each combination of dynamic techniques can be compared on the basis of the time required to find a solution of minimum temporal distance. Based on this measurement, how should we expect these dynamic techniques to perform? I will test the following hypotheses:

Hypothesis 3.15. Meta-value oracles will lead to an optimally stable solution faster than the baseline algorithm. The first solution found using meta-value oracles has, on average, a lower temporal distance than the first solution found using the baseline algorithm. Starting with a lower temporal distance will allow branch and bound to prune more nodes from the search tree faster, thus finding an optimal solution in less time.

Hypothesis 3.16. Temporal bounds oracles will lead to an optimally stable solution faster than either meta-value oracles or the baseline algorithm. The first solution using temporal bounds oracles has, on average, a lower temporal distance than the first solution using either meta-value oracles or the baseline algorithm, which should increase the effectiveness of branch and bound even more.

Hypothesis 3.17. Nogood recording will improve stability optimization performance when combined with any other dynamic technique. Nogood recording does not, on average, cause the first solution found to the modified problem to have a lower temporal distance, so it does not enhance branch and bound. Instead, just as nogood recording prunes some nodes of the search tree before the first solution is found, it will continue to do so after. Note that "nogood recording" here refers to the process of keeping nogoods from the initial DTP and applying them when solving the modified DTP; this is unrelated to the use of valued nogoods when solving the modified DTP.

In order to compare the optimization performance of each combination of dynamic techniques, I implemented branch and bound and valued nogoods in the same meta-SAT solver used in all of the previous experiments. The temporal distance metric in Formula 3.11 is monotonically non-decreasing, so it is possible to apply both of these optimization techniques. When using an oracle, after the first solution is found for the modified problem, the oracle is ignored for the remainder of the search. For each of the 846 randomly generated DDTPs with consistent initial and modified problems, I ran the solver until it either found an optimal solution or exceeded the 500-second time limit.

Figure 3.30 tests Hypothesis 3.15 by comparing the stability optimization time required by the baseline algorithm and meta-variable oracles. Each point on the graph plots the optimization time using meta-value oracles, and each plus sign plots the optimization time using the baseline algorithm. Problems are sorted along the horizontal axis in ascending order according to the optimization time required by the baseline algorithm. The vertical axis measures the optimization time in seconds on a logarithmic scale. For the majority of problems, the optimization time required using meta-value oracles is significantly less than the baseline algorithm. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 3.15 is correct for this problem set—meta-value oracles optimize for stability faster than the baseline algorithm.

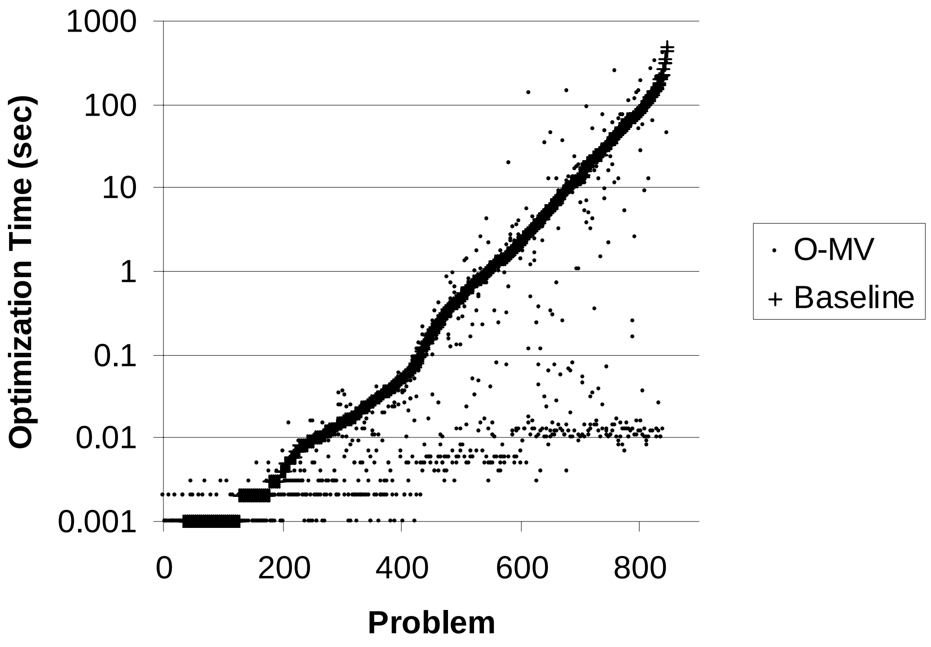

Figure 3.30. Time to optimize for stability in randomly generated DDTPs using meta-value oracles and the baseline algorithm.

Hypothesis 3.16 conjectures that temporal bounds oracles will optimize for stability faster than either meta-value oracles or the baseline algorithm. Figures 3.31 and 3.32 therefore compare the optimization time using temporal bounds oracles to the optimization time using the baseline algorithm and meta-value oracles, respectively. In Figure 3.31, problems are sorted according to the optimization time using the baseline algorithm, and in Figure 3.32, problems are sorted according to the optimization time using meta-value oracles. In both cases, a Student's paired t-test confirms Hypothesis 3.16 with 99.5% confidence for this set of randomly generated DDTPs.

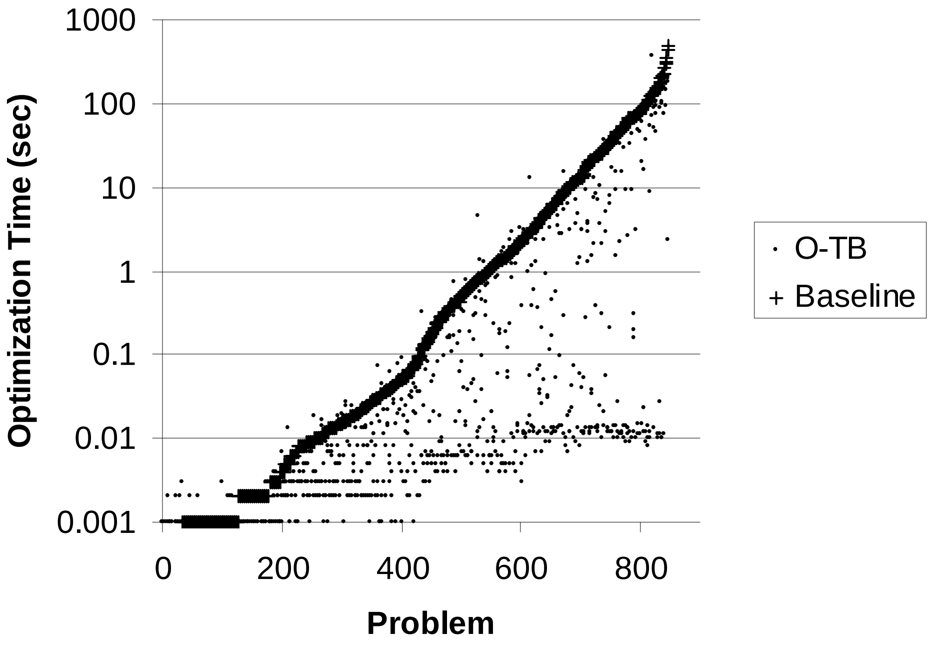

Figure 3.31. Time to optimize for stability in randomly generated DDTPs using temporal bounds oracles and the baseline algorithm.

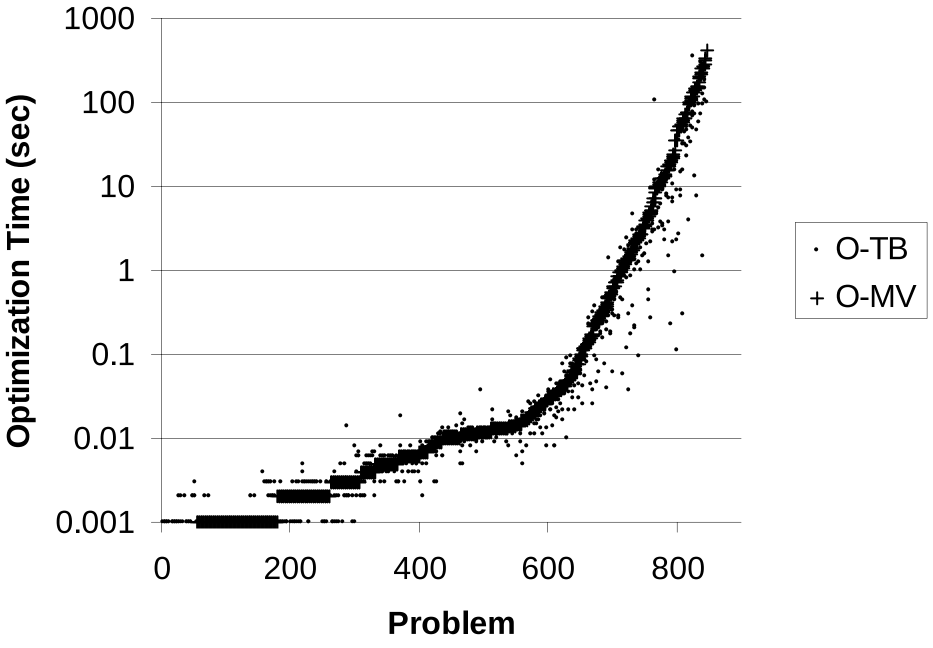

Figure 3.32. Time to optimize for stability in randomly generated DDTPs using temporal bounds oracles and meta-value oracles.

Finally, according to Hypothesis 3.17, nogood recording will improve stability optimization performance when combined with any other dynamic technique. Figure 3.33 shows that for a majority of the problems, nogood recording optimizes for stability faster than the baseline algorithm, and a Student's paired t-test confirms with 99.5% confidence that this difference is statistically significant. Figures 3.34 and 3.35 compare the optimization performance of meta-value oracles and temporal bounds oracles, respectively, with and without nogood recording. In both of these cases, the difference in stability optimization time is not statistically significant according to a Student's paired t-test. Combined, these results confirm that Hypothesis 3.17 is only partially correct. Nogood recording improves stability optimization performance compared to the baseline algorithm, but does not significantly improve performance when combined with either meta-value or temporal bounds oracles.

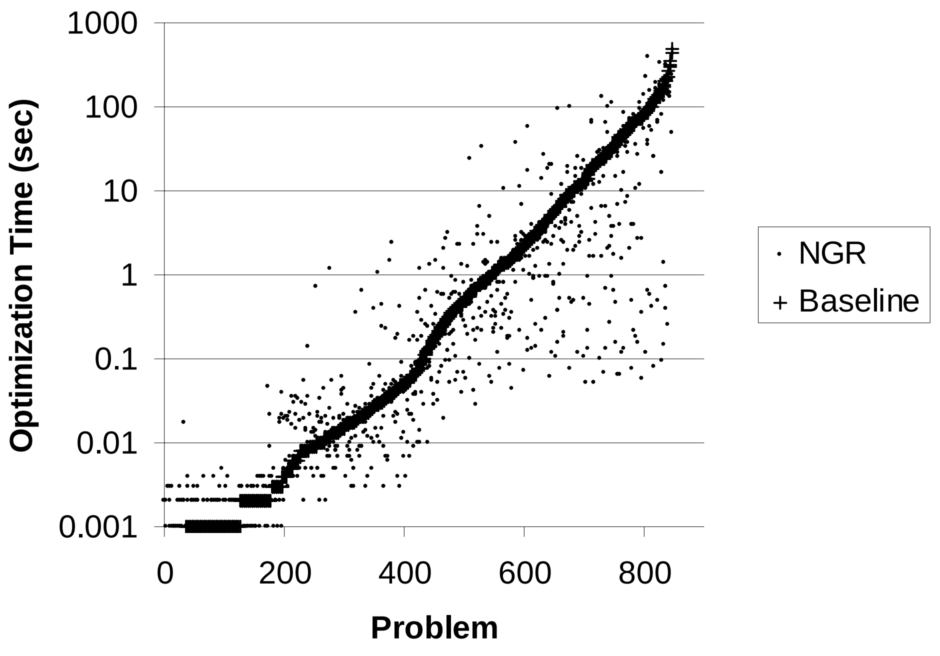

Figure 3.33. Time to optimize for stability in randomly generated DDTPs using nogood recording and the baseline algorithm.

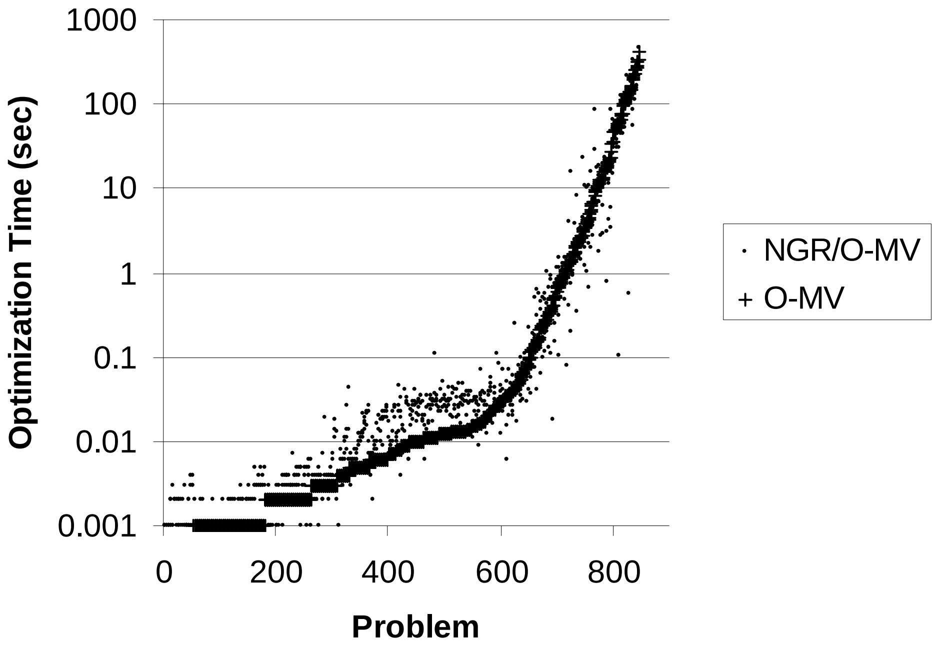

Figure 3.34. Time to optimize for stability in randomly generated DDTPs using meta-value oracles and the combination of meta-value oracles and nogood recording.

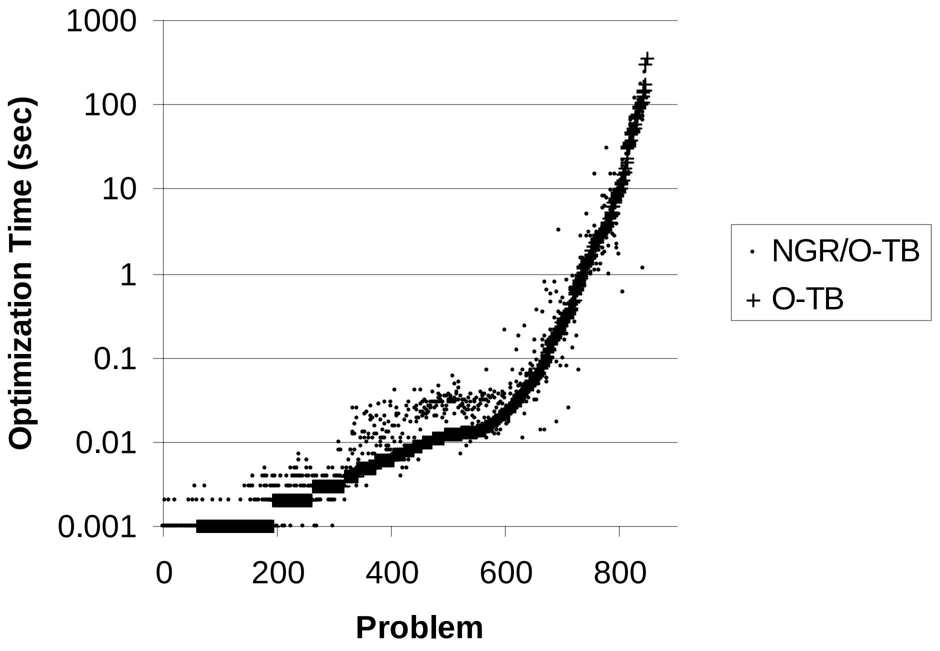

Figure 3.35. Time to optimize for stability in randomly generated DDTPs using temporal bounds oracles and the combination of temporal bounds oracles and nogood recording.

Why is it that nogood recording turns out not to improve stability optimization performance when combined with oracles? Nogood recording operates by pruning nodes from the search tree that cannot be extended to a solution. Branch and bound operates by pruning nodes that cannot be extended to a solution *that is any better than the best solution found so far*. This means that branch and bound can preempt nogood recording, pruning a branch before it has reached the point where it cannot be extended to a solution. Branch and bound is most effective when a solution has already been found with a low temporal distance. As evidenced by Figures 3.22, 3.23, and 3.28, following an oracle generally leads to a solution with much lower temporal distance than the baseline algorithm does. Thus, oracles (whether meta-value oracles or temporal bounds oracles) increase the effectiveness of branch and bound, which in turn decreases the effectiveness of nogood recording.

To lend some perspective as to the relative optimization performance of each combination of dynamic techniques, Figure 3.36 reports the average percent improvement of each technique over the baseline optimization run time. Meta-value oracles proved quite effective, reducing the optimization time by an average of 42.7% of the baseline, while temporal bounds oracles are considerably more effective, reducing the optimization time by an average of 71.2%. Nogood recording is also very effective relative to the baseline algorithm, reducing optimization time by an average of 36.0%. On the other hand, nogood recording is not very effective when combined with oracles. As pointed out earlier, the additional improvement made by combining nogood recording with either meta-value oracles or temporal bounds oracles is not statistically significant.

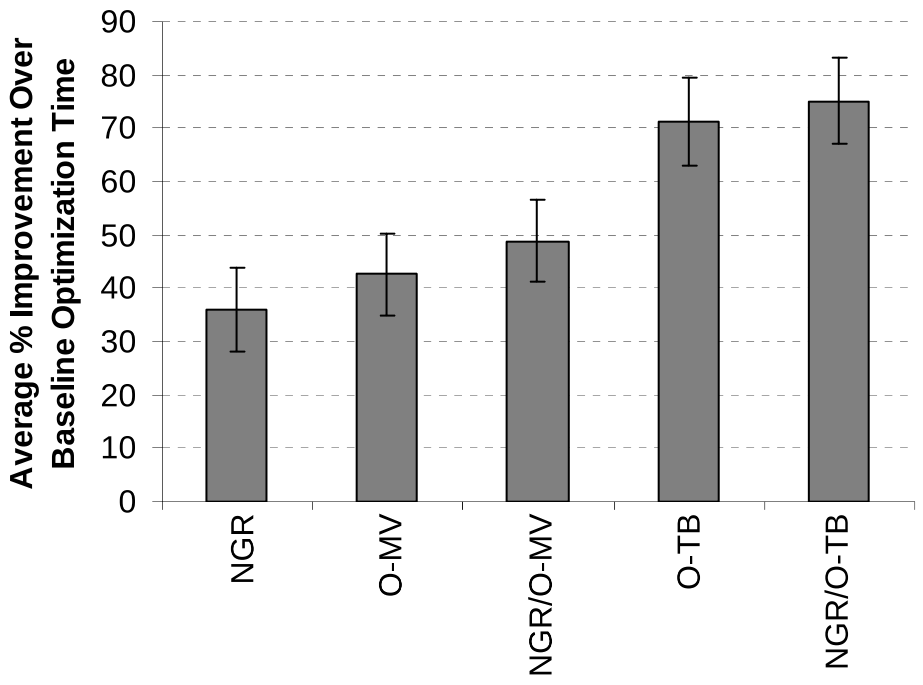

Figure 3.36. Average percent improvement over baseline stability optimization time using combinations of dynamic techniques when solving randomly generated initially consistent DDTPs.

In Section 3.5.2, it was shown that as the number of changes in the change set increases, the effectiveness of meta-value oracles and temporal bounds oracles at reducing temporal distance decreases, while nogood recording remains consistently ineffective. As just shown above, nogood recording does significantly reduce the run time necessary to find an optimally stable solution. It is therefore reasonable to hypothesize that, for any combination of these techniques, the amount by which the optimization time improves will decrease as the number of changes increases.

Hypothesis 3.18. The effectiveness of any combination of nogood recording, meta-value oracles, and/or temporal bounds oracles at improving the run time necessary to find an optimally stable solution will significantly decrease as the number of changes between the initial and modified problems of initially consistent DDTPs increases.

Figure 3.37 tests this hypothesis by partitioning the problems according the number of changes in each changes set. As can be seen in this figure, the effectiveness of each combination of techniques tends to drop off as the number of changes increases. The exact number of changes that must be added before statistical significance can be proven varies from one combination to the next, but for all combinations, if the number of changes increases by 6, then the effectiveness of the combination is reduced by a statistically significant amount with 99.5% confidence according to Student's paired t-tests, proving that Hypothesis 3.18 is correct.

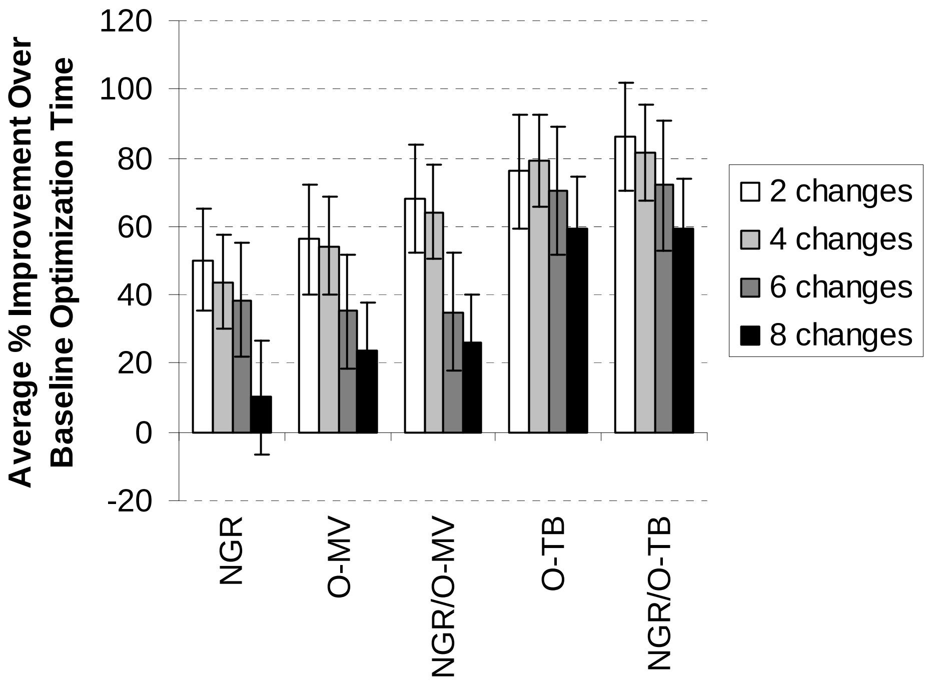

Figure 3.37. Average percent improvement over baseline stability optimization time using combinations of dynamic techniques when solving randomly generated initially consistent DDTPs, partitioned by the number of changes in each change set.

3.5.4. Testing Stability with Air Traffic Control Problems

To further compare the stability performance of nogood recording and oracles, I repeated the experiments of Sections 3.5.2 and 3.5.3, only this time using the initially consistent air traffic control problems from Section 3.4.2. In this problem set, the modified DTP of all 400 problems is consistent, so it was possible to measure temporal distance for each problem in the set. As with the initially consistent randomly generated DDTP experiments, I optimized the temporal distance metric of Formula 3.11, repeated here:

$$\textbf{TemporalDistance}(S_i, S_{i+1}) = max_{x, y \in T} \left(max(0, dd[x, y]) \right) \qquad (3.11)$$

Unfortunately, the problems of this test set were not conducive to demonstrating any difference between the dynamic techniques; for all augmented air traffic control problems, the first solution found by each of the four combinations of dynamic techniques was optimally stable. If we compare the temporal distance of the first solution found using each combination of techniques, these values are all equal because they are all optimal. If we compare the run time necessary to find an optimal solution, these run times will be exactly the same as those reported in Section 3.4.2 because the first solution is always optimal.

How could it be that every combination of dynamic techniques finds an optimal solution the first time every time? The actual temporal difference of the optimal solution of almost all of these problems was exactly equal to the temporal horizon of 10,800, and for the other problems, it was just slightly more.

This particular behavior results from the structure of the simulated temporal constraints. Each simulated temporal constraint represents a choice of ordering constraints between pairs of runway arrival times. If the runway times of two aircraft have no relationship to each other in the solution of the initial DTP, the difference between these timepoints is only bounded by the temporal horizon, which is 10,800 in these problems. When the problem is modified by adding a new aircraft, several new simulated ordering constraints are added with it. If the solution of the modified problem imposes an ordering constraint between the runway arrival times of two aircraft that were unrelated in the initial DTP, then the bound on that pair of timepoints has decreased from 10,800 to 0. In this case, the temporal distance between the two solutions must be at least 10,800, which is exactly what was observed.

This should not be taken to mean that stability is a useless measure in the air traffic control domain. Instead, this demonstrates the importance of choosing a temporal stability metric carefully. For these experiments, stability was measured using Formula 3.11, which measures the maximum amount by which the induced bound on any pair of timepoints has decreased relative to the initial solution schedule. While this metric might make sense in other domains, it may not be a good fit for the air traffic control domain. For example, the amount of time between two aircraft passing through their meter fixes is of little interest in this domain. The times at which two aircraft pass through their respective meter fixes might be unrelated in the initial solution (with a temporal bound equal to the horizon of 10,800), but might be ordered in the modified solution (with a temporal bound of 0). In this case, the value of the local stability metric would be 10,800, which could possibly dominate the global stability value.

A more appropriate temporal stability metric might measure the amount by which each aircraft's landing time is delayed. This can be accomplished using the same local stability metric from Formula 3.11, but modifying the global stability metric by weighting each bound $RT(A) - TR$ with 1 (where $RT(A)$ is the time at which aircraft A passes its runway threshold) and weighting all of the other bounds with 0. The resulting temporal stability metric would report the maximum amount of time by which any aircraft's landing time is delayed. Alternatively, one might replace each weight of 1 with a weight equal to the number of passengers on the aircraft. By applying an average rather than a maximum as the global stability metric, the resulting temporal stability metric would report the average amount of time by which each *passenger's* landing time is delayed. Although this temporal stability metric is more sophisticated and a better fit with the air traffic control domain, it still fits within the family of metrics defined in Section 3.5.1.

When the local stability metrics are averaged, as suggested in the last temporal stability metric above, it is not as likely that a change in a single bound would dominate the value of the global stability metric, so it is not as likely that the first solution found is optimally stable. Instead, one would expect solution quality to increase over time, leaving room for the different dynamic techniques to lead to measurably different stability performance. I will leave the empirical comparison using more sophisticated temporal metrics such as this for future work.

3.6. Conclusions

In this chapter, I defined the Dynamic Disjunctive Temporal Problem, which is capable of representing scheduling problems with constraints that change over time. I presented two possible interpretations of oracles that could be applied to DDTPs, meta-value oracles and temporal bounds oracles. I tested nogood recording and oracles from the DCSP literature and found that they are also effective when applied to DDTPs. I identified the need for a new technique to improve efficiency for initially inconsistent DDTPs, and I developed justification testing to fill this need. On randomly generated DDTPs, justification testing proved to be more than ten times as effective as nogood recording. I also tested these techniques on DDTPs derived from actual air traffic control recordings, and I found that the structure of the problems can have a large effect on the relative performance of the techniques.

To measure the similarity between the solution schedules of a DDTP, I developed a family of temporal stability metrics that can be applied in a variety of scheduling domains. I described several sophisticated temporal stability metrics from within this family that could be applied to the air traffic control domain. I tested the stability performance of nogood recording and oracles on randomly generated DDTPs, and I found that temporal bounds oracles were the most effective whether stability was measured in terms of the temporal distance of the first solution or in terms of the time required to find an optimal solution. Given two possible interpretations of oracles, the new technique of justification testing that can improve efficiency if the previous problem was inconsistent, and a family of temporal stability metrics, future developers now have a choice of a variety of dynamic scheduling techniques that can be used in many different situations.

Chapter 4

Hybrid Scheduling Problems

4.1. Introduction

It is often the case that a scheduling problem is not purely temporal in nature—in addition to deciding *when* events should occur, it may also be necessary to decide *how* they occur. From a computational point of view, this presents an interesting challenge. On one hand, the domain of each timepoint is continuous and infinite, while on the other hand, the domain of each decision variable is typically discrete and finite. Despite these differences, these two types of variables can interact with one another, so it is necessary to develop representations and methods that can handle both.

In this chapter, I formally define the Hybrid Scheduling Problem (HSP) as a representation of scheduling problems that contain both temporal and finite-domain variables. I present several techniques for improving efficiency when solving an HSP, including detecting local inconsistencies, focusing search, and choosing the most-constrained subset. Each of these techniques exploits the inherent inequities between the different types of variables. I compare the performance of each of these techniques when solving randomly generated HSPs as well as real-world problems based on the air traffic control recordings described in Chapter 3.

4.1.1. Hybrid Scheduling Problem Basics

A Hybrid Scheduling Problem can be defined as follows:

Definition 4.1. A *Hybrid Scheduling Problem (HSP)* is a tuple $\langle V, D, C\rangle$. The set of variables V can be partitioned into two sets: 1) V_F, the set of finite-domain variables with domains D, and 2) V_T, the set of temporal variables with real domains. The set of constraints C can be partitioned into three sets: 1) C_F, the set of finite-domain constraints defined over V_F, 2) C_T, the set of disjunctive temporal constraints defined over V_T, and 3) C_H, the set of hybrid constraints defined over V_F and V_T.

Definition 4.2. A *hybrid constraint* is the disjunction of a finite-domain constraint and a disjunctive temporal constraint.

The HSP has almost the same expressive power as the MLILP with logical Boolean and linear UTVPI constraints (Sheini and Sakallah 2005). In fact, any instance of MLILP with logical Boolean and linear UTVPI constraints can be expressed as an HSP as long as the UTVPI constraints of that instance can be expressed as TD constraints. Recall from Section 2.5.2 that the MLILP is a Boolean combination of literals in which each literal is an atom or its negation, and each atom is either a Boolean variable or a UTVPI constraint. If the MLILP is represented in CNF, the clauses can be partitioned into three sets. Each clause of the first set only contains atoms that are Boolean variables; this set of clauses can be represented as a set of finite-domain CSP constraints. Each clause of the second set only contains atoms that are UTVPI constraints; assuming that each UTVPI constraint can be expressed as a TD constraint, each clause of this set has exactly the form of a disjunctive temporal constraint. Each clause of the third set contains atoms that are both Boolean variables and UTVPI constraints. For each clause in this last set, the literals can be partitioned into a set with only Boolean variables and a set with only UTVPI constraints. The set of literals with Boolean variables can be represented as a finite-domain CSP constraint, and the set of literals with UTVPI constraints can be represented as a disjunctive temporal constraint (again, assuming that each UTVPI constraint can be expressed as a TD constraint). Since these two sets of literals are connected by a disjunction, each clause of this last set can be represented as a hybrid constraint.

As stated in Definition 4.1, the variables of an HSP can be partitioned into two subsets—the finite-domain variables V_F, and the temporal variables V_T. Based on these subsets of variables, an HSP separates naturally into two subproblems—a finite-domain CSP $\langle V_F, D, C_F\rangle$, which I will refer to as the finite-domain subproblem, and a DTP $\langle V_T, C_T\rangle$, which I will refer to as the temporal subproblem. These subproblems are related to each other through the hybrid constraints of C_H. For each hybrid constraint, an HSP solver must decide whether to add an additional constraint to the finite-domain subproblem or the temporal subproblem.

While it would be possible to think of the hybrid constraints as defining a third subproblem, this subproblem would still contain both finite-domain and temporal variables, and so would itself be another HSP. The techniques that I will present in this chapter are based on subproblems that do not mix variable types, so the present work will only focus on the finite-domain and temporal subproblems. These techniques could be extended to handle hybrid subproblems or subproblems of other variable types; I will discuss these ideas in later sections on future work.

Separating an HSP into subproblems helps to show that an HSP is an NP-complete problem. An HSP is obviously NP-hard because it contains a finite-domain CSP and a DTP as subproblems, and these are both NP-hard problems. An HSP is in NP because a complete assignment can be verified as a solution in polynomial time. This can be accomplished by first verifying that the finite-domain variable assignments satisfy all of the finite-domain constraints in polynomial time, just as it is in a finite-domain CSP, and then verifying that the temporal variable assignments satisfy all of the disjunctive temporal constraints in polynomial time, just as it is in a DTP. Finally, it is possible to verify that each hybrid constraint is satisfied in polynomial time by verifying that either the finite-domain component or the temporal component of the hybrid constraint is satisfied.

The structure of an HSP offers many different approaches to solving it. As with the DTP and many classes of SMT, one way to solve an HSP is to transform the problem into a meta-level problem and apply well-known finite-domain search techniques. While meta-CSP and meta-SAT transformations are both possible, current state of the art solvers employ the meta-SAT transformation. To transform an HSP into a meta-SAT representation, first let each finite-domain variable be represented as a set of Boolean variables and clauses. Next, transform each finite-domain constraint into a set of Boolean clauses. Let each finite-domain constraint that is part of a hybrid constraint be represented as a conjunction of Boolean clauses. Next, replace each TD constraint with a Boolean indicator variable, as is done in the meta-SAT transformation of a DTP. At this point, the HSP is represented as a logical combination of Boolean variables, so the transformation can be completed by converting the problem to CNF.

After the transformation, the clauses can easily be partitioned according to the type of constraints they represent in the original problem. Any clause that contains all indicator variables represents a temporal constraint, any clause that contains no indicator variables represents a finite-domain constraint, and any clause that contains some variables that are indicator variables and some that are not represents a hybrid constraint. This simple distinction makes it possible to separate the meta-SAT representation of an HSP into a finite-domain SAT subproblem and a temporal DTP subproblem in time that is linear in the total number of literals in all meta-SAT clauses.

The meta-SAT approach to solving an HSP offers several advantages. First, it allows for the use of all of the most advanced efficiency techniques developed in the Boolean SAT literature, such as unit propagation, two watched literals per clause, and the VSIDS variable selection heuristic. Second, it allows a solver to ignore the distinction between the Boolean indicator variables that represent TD constraints and the Boolean variables that represent finite-domain variables and constraints. Ignoring this distinction, a standard SAT solver can be extended in a simple and elegant manner to solve HSPs. Namely, before returning a solution, the TD constraints that correspond to the meta-level assignment of Boolean indicator variables must be tested for temporal consistency. This is common practice in both DTP and SMT solvers. I will refer to this approach as the baseline algorithm. Based on the practice of ignoring the distinction between variables of different types, previous research has focused on how to improve efficiency when testing temporal consistency (Dutertre and de Moura 2006), how often to test temporal consistency (Nieuwenhuis and Oliveras 2005), or how best to extract meta-level information as a result of testing temporal consistency (Sheini and Sakallah 2005).

4.1.2. HSP Problem Space

Rather than ignore the distinction between finite-domain and temporal meta-level variables, I have developed a set of techniques that exploit this distinction to improve efficiency when solving HSPs. Figure 4.1 depicts one possible representation of the problem space of HSPs. In this figure, the horizontal axis represents the tightness of the temporal constraints, with loose constraints to the left and tight constraints to the right. The vertical axis represents the tightness of the finite-domain constraints, with loose constraints to the bottom and tight constraints to the top. The gray area to the right (respectively, top) of the graph represents HSPs in which the temporal (respectively, finite-domain) constraints are so tight that the temporal (respectively, finite-domain) subproblem contains an inconsistency.

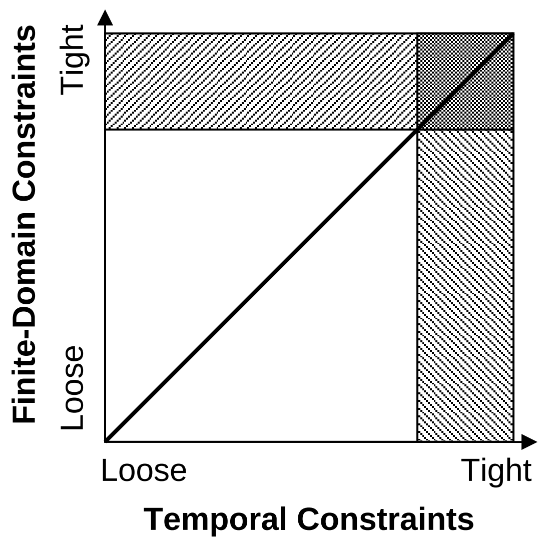

Figure 4.1. Problem space of Hybrid Scheduling Problems separated by tightness of temporal and finite-domain constraints.

Based on this representation of the HSP problem space, the key insight to improving efficiency is that in many HSPs, temporal and finite-domain variables are not constrained equally. In Figure 4.1, only problems that fall along the main diagonal contain variables of different types that are constrained equally; in problems that fall below the main diagonal, the temporal constraints are tighter, and in problems that fall above the main diagonal, the finite-domain constraints are tighter. Based on this insight, I have developed three techniques to improve efficiency when solving HSPs: 1) detecting local inconsistencies, which breaks the problem into temporal and finite-domain subproblems, 2) focusing search, which increases propagation by assigning variables of one type before the other, and 3) choosing the most-constrained subset, which estimates the solution density of the subproblems and moves the variables of the more tightly constrained subproblem to the beginning of the search ordering.

4.2. Detecting Local Inconsistencies

For some scheduling problems, the temporal or finite-domain constraints are so tight that one of the subproblems contains an inconsistency. I will refer to an inconsistency that exists entirely within either the temporal constraints or the finite-domain constraints as a local inconsistency:

Definition 4.2. A *local inconsistency* is an inconsistent subset of constraints that is contained entirely within either the temporal constraints C_T or finite-domain constraints C_F of an HSP. That is, a local inconsistency is an inconsistent subset of either C_T or C_F; a local inconsistency cannot contain any hybrid constraints of C_H.

Detecting local inconsistencies (DLI) is a simple process. First, break the HSP into separate finite-domain and temporal subproblems. As pointed out earlier, breaking an HSP into subproblems can be accomplished in linear time. Then solve each subproblem independently—if a local inconsistency is detected within either subproblem, then that local inconsistency serves as a justification for the complete HSP. In this case, there is no need to solve the complete HSP, thus decreasing run time. If, on the other hand, no local inconsistency is detected, then it is necessary to solve the complete HSP, and the time spent solving the subproblems has been wasted. In Figure 4.1, detecting local inconsistencies identifies problems that fall in the gray areas.

I will test the following hypothesis regarding the detection of local inconsistencies:

Hypothesis 4.1. Detecting local inconsistencies will significantly improve efficiency when solving HSPs.

4.2.1. Testing DLI with Randomly Generated Problems

To test Hypothesis 4.1, I generated a set of random HSPs. Each HSP in this set contains 20, 40, or 60 Boolean variables and 10, 20, or 30 timepoints. Each temporal constraint is a disjunction of two TD constraints, and each TD constraint is generated by randomly selecting two timepoints with uniform probability and assigning a temporal bound chosen from the range [-100, 100] with uniform probability. Each finite-domain constraint is a disjunction of three literals, as in 3-SAT. Each literal is generated by choosing a Boolean variable randomly with uniform probability, and then negating it with a probability of 0.5.

As in Section 3.4.1, I control the tightness of the finite-domain and temporal constraints with the number of constraints of each type. For both the temporal and the finite-domain subproblems, I added enough constraints to achieve 10%, 30%, 50%, 70%, and 90% consistency. For the temporal constraints, I was able to use the results of the DTP consistency experiments of Section 3.4.1. For each size of temporal subproblem, Table 4.1 lists the ratio of temporal constraints to timepoints necessary to achieve the consistency percentages used in these randomly generated HSPs.

% Consistency	**90%**	**70%**	**50%**	**30%**	**10%**
10 Timepoints	3.95	4.70	5.25	5.85	6.80
20 Timepoints	4.85	5.35	5.65	6.15	6.70
30 Timepoints	5.30	5.70	5.95	6.25	6.80

Table 4.1. Ratio of temporal constraints to timepoints necessary to achieve percentage consistency of randomly generated DTPs.

For the finite-domain subproblems, I ran a similar set of experiments on Boolean SAT problems. In these SAT experiments, each problem contains 20, 40, or 60 Boolean variables and three literals per clause; thus, these SAT problems are representative of the SAT subproblems of the randomly generated HSPs. For each problem size, I varied the ratio of constraints to variables from 3.50 to 6.00 at increments of 0.05. For each combination of problem size and ratio of constraints, I tested the consistency of 1,000 randomly generated problems. The results of these experiments are displayed in Figure 4.2. For each size of finite-domain subproblem, Table 4.2 lists the ratio of clauses to

Boolean variables necessary to achieve the consistency percentages of the randomly generated HSPs in this section.

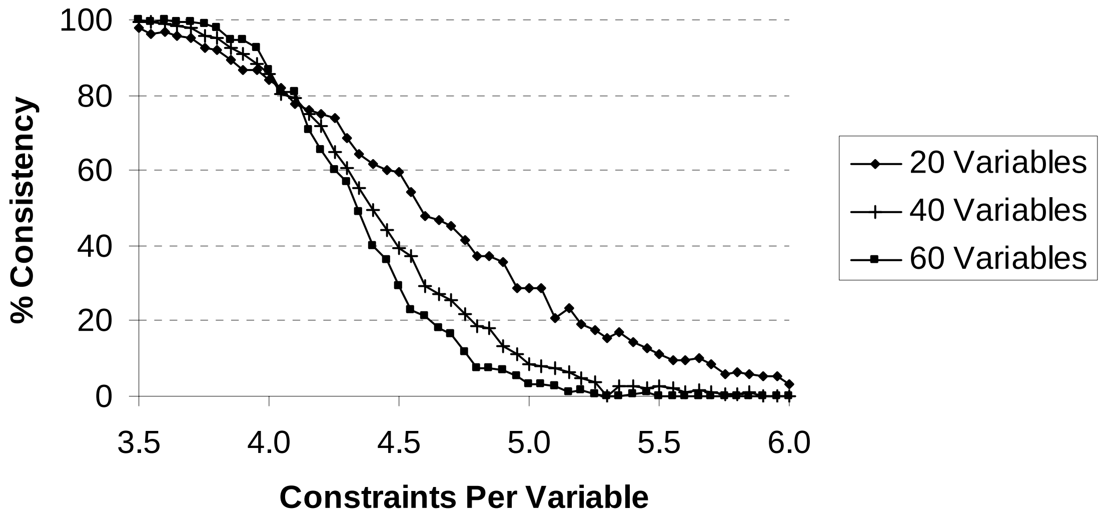

Figure 4.2. Percentage consistency of randomly generated SAT problems as a function of the ratio of constraints to Boolean variables.

% Consistency	**90%**	**70%**	**50%**	**30%**	**10%**
20 Variables	3.85	4.25	4.55	4.95	5.60
40 Variables	3.90	4.20	4.40	4.60	4.95
60 Variables	3.95	4.15	4.35	4.50	4.75

Table 4.2. Ratio of clauses to Boolean variables necessary to achieve percentage consistency of randomly generated SAT problems.

In addition to the temporal and finite-domain constraints, each randomly generated HSP also contains a set of hybrid constraints. Each hybrid constraint is the disjunction of two TD constraints and three Boolean literals. The number of hybrid constraints of each HSP of this problem set is 0.1 times the number of temporal constraints and finite-domain constraints combined. By varying the size and percent consistency of each subproblem, this set of randomly generated HSPs covers a large portion of the problem space.

For each combination of number of timepoints, number of Boolean variables, temporal consistency, and finite-domain consistency, I randomly generated 10 HSPs, resulting in a total of 2,250 problems in this set. I solved each HSP using the baseline algorithm, which solves the HSP directly, and the DLI algorithm, which solves the subproblems to detect local inconsistencies and only solves the complete HSP when necessary.

Figure 4.3 tests Hypothesis 4.1 by comparing the run time of each randomly generated HSP using the baseline algorithm and the DLI algorithm. The vertical axis reports run time, measured on a logarithmic scale. Each plus sign plots the run time using the baseline algorithm, and each dot plots the run time using the detection of local inconsistencies. Problems are arranged along the horizontal axis in ascending order according to the run time of the baseline algorithm, so the baseline run times form a backbone that curves from the lower left to the upper right of the graph. The combination of the logarithmic scale and the fact that problems are ordered by run time means that differences in run time on the right of the graph are larger than differences on the left of the graph. Although it may not be easy to see in this figure, a Student's paired t-test confirms with 99.5% confidence that Hypothesis 4.1 is correct—detecting local inconsistencies significantly improves efficiency for this set of randomly generated HSPs.

Because of the tightness of the temporal and finite-domain constraints of the problems in this set, one should expect 50% of the problems to contain a local inconsistency in the temporal subproblem, and 50% of the problems to contain a local inconsistency in the finite-domain subproblem. The temporal and finite-domain constraints are generated independently, so one should expect 75% of the problems to contain a local inconsistency in either subproblem. In fact, 1,701 of the randomly generated HSPs contain a local inconsistency, which is 75.6% of the 2,250 problems in this set.

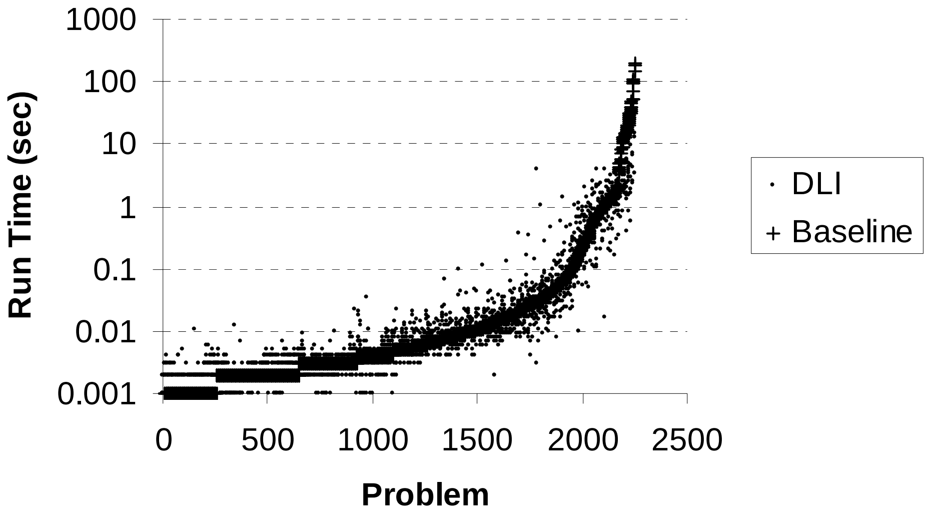

Figure 4.3. Run time of randomly generated HSPs using the baseline algorithm and detecting local inconsistencies.

With so many of the problems containing a local inconsistency, one would expect DLI to greatly outperform the baseline algorithm. In order to better understand the relative effectiveness of DLI, Figure 4.4 shows that detecting local inconsistencies reduced run time by 63.8% of the baseline run time on average. Once again, the error bars indicate standard error.

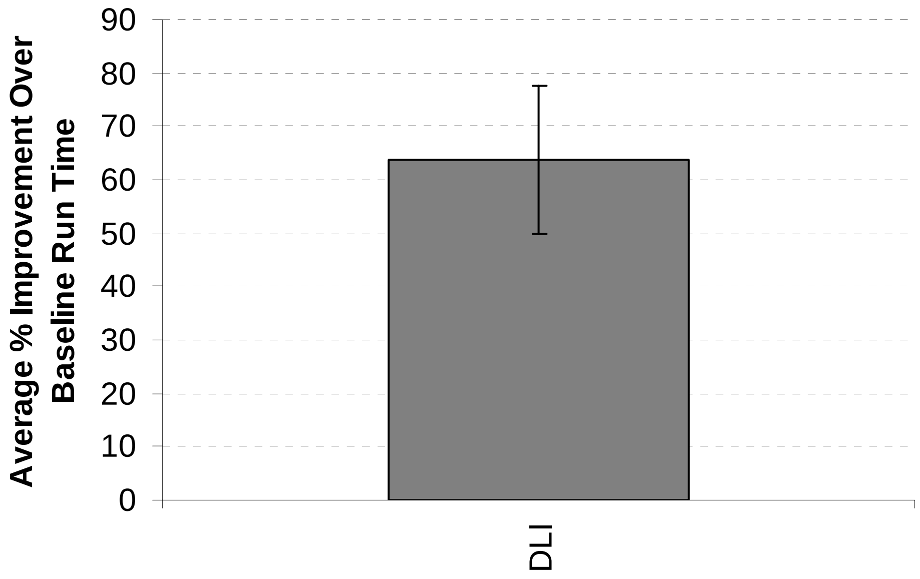

Figure 4.4. Average percent improvement over baseline run time using detecting local inconsistencies when solving randomly generated HSPs.

Figure 4.5 provides some perspective on the amount of time spent solving each subproblem versus the amount of time spent solving the complete HSPs. Each column of Figure 4.4 is split into three sections: the black section represents the average time spent solving the finite-domain subproblem, the light gray section represents the average time spent solving the temporal subproblem, and the dark gray section represents the average time spent solving the complete HSP. The baseline algorithm solves the complete HSP directly, so no time is spent solving the subproblems. Using DLI, the average time spent solving the finite-domain subproblem was 0.0018 seconds per problem, which is too small to be visible in this figure. The average time spent solving the temporal subproblem was 0.16 seconds per problem, which is 37.3% of the total average DLI run time. The remainder of the average DLI run time is comprised of the 0.27 seconds per problem to solve the complete HSP when no local inconsistency is detected, which is 62.3% of the total average DLI run time. With 24.4% of the HSPs in this set containing no local inconsistency, it is not surprising that the average run time to solve the complete HSP after testing for local inconsistencies is 22.6% of the baseline average run time.

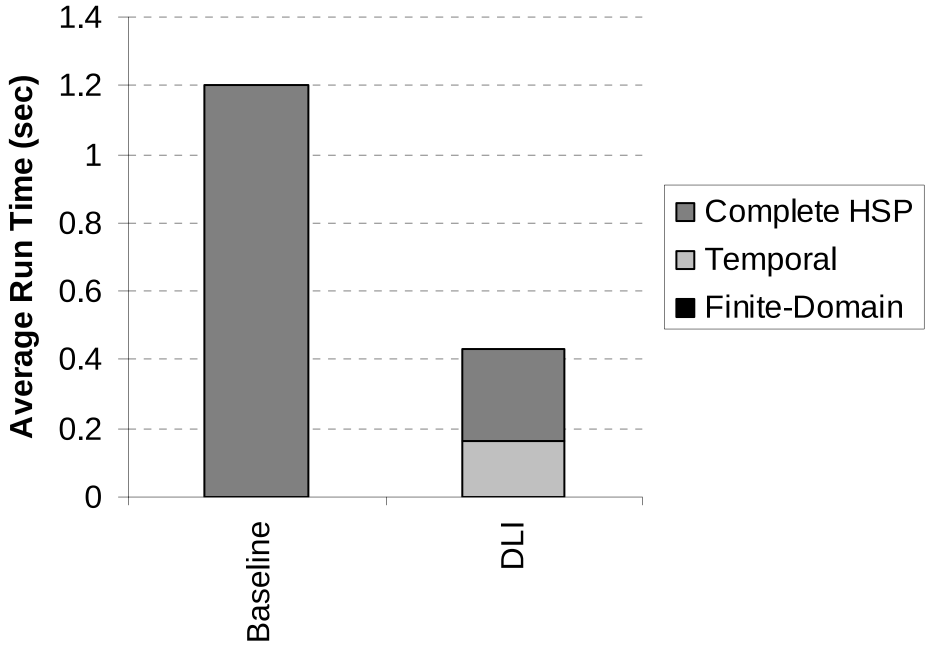

Figure 4.5. Average run time of randomly generated HSPs using the baseline algorithm and detecting local inconsistencies.

Although detecting local inconsistencies significantly improves efficiency for randomly generated HSPs, why should one expect this technique to improve efficiency when applied to real-world problems? Obviously, the degree to which DLI improves efficiency depends on the percentage of problems that actually contain a local inconsistency. Note, however, that each subproblem of an HSP contains fewer variables than the complete HSP does, and so their search spaces are exponentially smaller. This means that the time spent solving the subproblems of an HSP is only a small fraction of the time necessary to solve the complete problem. For example, solving the subproblems of the randomly generated HSPs required on average only 13.6% of the time to solve the complete HSPs. If a local inconsistency is detected, then a relatively large amount of time has been saved, while if no local inconsistency is detected, then only a relatively small amount of time has been wasted. This means that, for any given problem domain, only a small fraction of problems need to contain a local inconsistency for DLI to save time on average.

4.2.2. Testing DLI with Air Traffic Control Problems

The effectiveness of DLI is a function of the frequency with which local inconsistencies appear in the problems of a given domain. As pointed out previously, the problems of the air traffic control domain are all consistent, so they contain no local inconsistencies. While it would be possible to add simulated constraints as was done in Chapter 3, any resulting local inconsistencies would be completely artificial, so I will not run DLI experiments based on the air traffic control problems. I will, however, use the air traffic control problems to test the next hybrid technique: focusing search.

4.3. Focusing Search

Assuming that no local inconsistencies are detected, it is necessary to solve the complete HSP. In the baseline algorithm, variables are chosen according to a standard variable ordering heuristic. Because this heuristic is operating on the meta-level problem, it ignores the variable type when making a selection. While this approach simplifies the algorithm, it misses several opportunities to improve efficiency.

One opportunity is to improve the meta-level variable ordering heuristic by incorporating a more global perspective into the variable selection process. Recall that the basic idea behind most variable ordering heuristics, including VSIDS, is to choose the most-constrained variable, thus keeping the search tree as narrow as possible near the root and reducing the overall size of the search space. Most-constrained variable heuristics differ in how they measure the tightness of the constraints of a variable, but in general they measure the amount of interaction with other unassigned variables.

This approach is somewhat myopic in that it compares the tightness of constraints between individual variables. Consider a CSP in which the variables have been partitioned into two sets, *A* and *B*. Let *A* contain a variable v_A such that v_A is more tightly constrained than any other variable of either *A* or *B*. Let the remaining variables of *A* be very loosely constrained, and let the variables of *B* be constrained such that, on average, the variables of *B* are more tightly constrained than the constraints of *A*. Assume that there is no constraint interaction between v_A and the variables *B*. In this situation, a most-constrained variable heuristic would select v_A as the next variable to be assigned. After assigning a value to v_A, the most-constrained variable heuristic would then start selecting variables from *B* because those variables are more tightly constrained than the remaining variables of *A*. If an inconsistency that includes v_A is discovered later while assigning the variables of *A*, then it will be necessary to backtrack over all of the variables of *B* to try a different value for v_A. In this case, selecting v_A before the variables of *B* has done nothing to keep the search tree narrow near the root.

A better choice in this situation would be to assign all of the variables of *B*, then v_A, and then the remaining variables of *A*. A most-constrained variable heuristic cannot see this opportunity because it does not consider the larger context of subsets of variables. The same basic idea behind the most-constrained variable heuristic can be extended to subsets. In a *most-constrained subset (MCS) heuristic*, the variables of a

CSP are partitioned into subsets *a priori*, and these subsets are placed in descending order according to an aggregate measure of the tightness of their constraints. During search, a variable is chosen in two phases. First, the most-constrained subset *S* that contains any unassigned variables is chosen according to the subset ordering. Then a variable *v* of *S* is chosen according to a most-constrained variable heuristic. Once a value is assigned to *v*, the effects of that assignment are propagated to all other unassigned variables. If unit propagation is employed, then any variable whose domain is reduced to a single value is immediately assigned to that value, no matter which subset it belongs to.

When applied to the meta-level representation of an HSP, the most-constrained variable heuristic suffers from another drawback. The meta-level variables represent different types of object-level variables and constraints, so standard most-constrained variable heuristics that do not account for this might have difficulty comparing the tightness of constraints on different types of meta-level variables accurately. The relationship between the number of constraints on a finite-domain variable and how constrained it is might be operating on a different scale than the relationship between the number of constraints on a temporal variable and how constrained it is. This means that using a standard most-constrained variable heuristic to compare finite-domain and temporal variables would be like comparing apples to oranges.

In order to apply the most-constrained subset heuristic to HSPs, it is necessary to partition the variables into subsets. The meta-level variables of an HSP separate naturally into two subsets—the temporal indicator variables and the finite-domain variables. Note that, as shown in Figure 4.1, even if both subproblems are consistent, it is still unlikely that the variables of each type are constrained equally. According to the most-constrained subset heuristic, efficiency can be improved by assigning the temporal variables first for any HSP that falls below the diagonal line and assigning the finite-domain variables first for any HSP that falls above it. Predicting whether a particular HSP falls above or below this line is an interesting and difficult problem that I explore later in Section 4.4.

For any particular problem domain, it is unlikely that the number of problems in which the finite-domain constraints are tighter is exactly equal to the number of problems in which the temporal constraints are tighter. This means that it is not necessary to predict the most-constrained subset for each individual HSP. For a particular problem domain, it should be possible to improve efficiency on average by focusing search on the subset of variables that is more tightly constrained in the majority of problems. If we believe that most of the problems in the domain fall above the main diagonal of Figure 4.1, then we should focus the search on the finite-domain variables; if we believe that most of the problems in the domain fall below the main diagonal, then we should the focus search on the temporal variables.

In effect, focusing search operates on the assumption that one variable type will be more tightly constrained in a majority of the problems that the solver will be given. It is not the responsibility of the search algorithm to predict which variable type will usually be more tightly constrained; that determination must be made externally, most likely by a domain expert. For most problems in which the search is focused on the more tightly constrained variable type, run time should decrease. On the other hand, for most problems in which the search is focused on the less tightly constrained variable type, run time should increase. As long as the prediction as to which variable type will be more tightly constrained in the majority of problems is correct, efficiency should improve on average. Whether or not this prediction is correct, one should expect some of the problems to run faster while others run slower. The net result could be increased run time variability or increased worst-case run time. As stated in Chapter 1, I will measure efficiency as the average reduction in run time, so I will merely note that increased variability and worst-case run time could be potential drawbacks of focusing search.

In addition to correctly choosing the most-constrained subset of variables in a majority of problems, focusing search may improve efficiency another way as well—by avoiding thrashing. Any solution assignment for the complete HSP must contain solution assignments to the temporal and finite-domain subproblems as subsets. That is, along the way to solving the complete HSP, a solver must implicitly solve both of the subproblems. The variables of the two subproblems can only interact through the hybrid constraints, and if there are relatively few hybrid constraints relative to the number of finite-domain and temporal constraints, then assigning a finite-domain variable is unlikely to propagate to many temporal variables, and vice versa. In the context of HSPs, "thrashing" means frequently switching back and forth between assignments to finite-domain variables and assignments to temporal variables. By focusing search, the solver will solve one of the subproblems, and then extend that assignment to a solution to the complete problem. In this manner, focusing search might help to improve efficiency by avoiding this thrashing behavior. If this is true, focusing search should improve efficiency whether or not the search is focused on the most-constrained subset of variables.

The efficiency gains of focusing search by avoiding thrashing should be most apparent when there are relatively few hybrid constraints compared to the number of finite-domain and temporal constraints. The ratio of hybrid constraints to finite-domain and temporal constraints will vary from domain to domain and problem to problem. The number of hybrid constraints of each randomly generated HSP of my test set is only 0.1 times the number of finite-domain and temporal constraints. If focusing search does improve efficiency by avoiding thrashing, this fact should be evident in problems with such a relatively small number of hybrid constraints.

To assess the effects of focusing search when solving HSPs, I compare three algorithms. The baseline algorithm selects variables according to the most-constrained variable heuristic, which ignores variable type when making its choice. The FS(*F*) algorithm focuses search on the finite-domain variables by selecting variables with the most-constrained subset heuristic, placing the finite-domain variables in the first subset and the temporal variables in the second. Conversely, the FS(*T*) algorithm focuses search on the temporal variables by selecting variables with the most-constrained subset heuristic, placing the temporal variables in the first subset and the finite-domain variables in the second. I test the following hypotheses to compare the efficiency of these three algorithms:

Hypothesis 4.2. Focusing search on the finite-domain variables first with the FS(*F*) algorithm will significantly improve efficiency relative to the baseline algorithm.

Hypothesis 4.3. Focusing search on the temporal variables first with the FS(*T*) algorithm will significantly improve efficiency relative to the baseline algorithm.

4.3.1. Testing FS with Randomly Generated Problems

Focusing search could affect efficiency in two ways: either by providing the variable selection heuristic with a more global perspective of the constraints, or by reducing thrashing between variables of different types. After running each of the three algorithms on a set of HSPs, how can one determine which effect is the cause of any efficiency gains? If the efficiency gain is due to the variable selection heuristic, one would expect to see one of either FS(*F*) or FS(*T*) to run faster than the baseline algorithm and the other to run slower. This is because only one type of variable can constitute the most-constrained subset in a majority of the problems. In this case, either Hypothesis 4.2 or 4.3 will be confirmed, but not both. If, on the other hand, the efficiency gain is due to a reduction in thrashing, one would expect to see both FS(*F*) and FS(*T*) to run faster than the baseline algorithm. In this case, Hypotheses 4.2 and 4.3 will both be confirmed.

Detecting local inconsistencies can quickly identify HSPs that fall in the gray areas of Figure 4.1, so I compared the FS(*F*) and FS(*T*) algorithms against the baseline algorithm with a second set of 2,250 randomly generated HSPs that fall entirely within the white area of Figure 4.1. The problems in this second set are generated in the same manner as the problems in the first, except that if a randomly generated problem contains a local inconsistency, it is discarded and replaced by a new one. As before, a time limit of 500 seconds is imposed when solving each problem.

To test Hypothesis 4.2, Figure 4.6 compares the run time of the baseline algorithm against the run time of the FS(*F*) algorithm based on this set of randomly generated HSPs. In Figure 4.6, the problems are arranged along the horizontal axis in ascending order according the run time of the baseline algorithm, and the vertical axis shows the run time of each algorithm being compared. Each plus sign plots the run time using the baseline algorithm, and each dot plots the run time using the FS(*F*) algorithm. Although focusing search on the finite-domain variables reduced run time for some of the problems, it actually increased run time for the majority of problems. A Student's paired t-test proves that Hypothesis 4.2 is *incorrect* with 99.5% confidence—for this set of randomly generated HSPs, focusing search on the finite-domain variables does not improve efficiency, but actually hurts efficiency significantly.

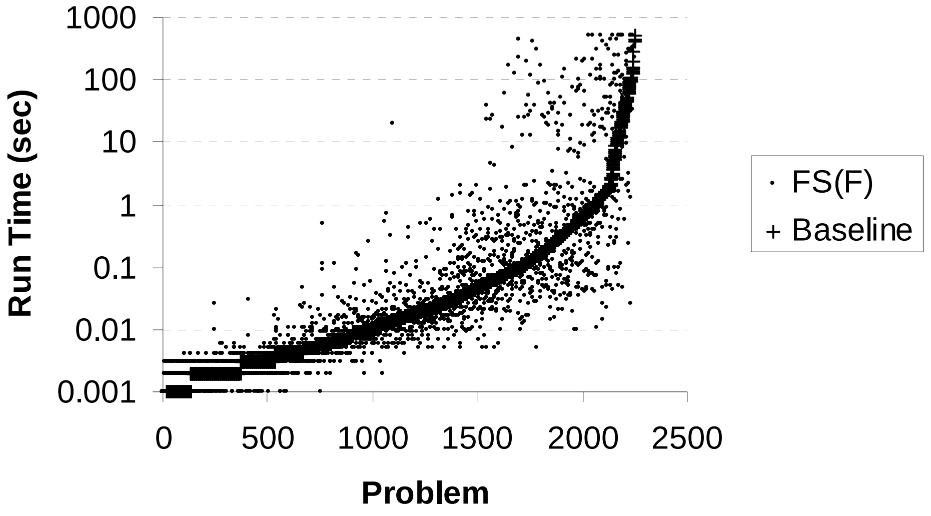

Figure 4.6. Run time of randomly generated HSPs using the baseline algorithm and focusing search on finite-domain variables.

To test Hypothesis 4.3, Figure 4.7 compares the run time of the baseline algorithm against the run time of the FS(*T*) algorithm. As before, the problems are arranged along the horizontal axis in ascending order according the run time of the baseline algorithm, and the vertical axis shows the run time of each algorithm being compared. Each plus sign plots the run time using the baseline algorithm, and each dot plots the run time using the FS(*T*) algorithm. Comparing Figure 4.7 to Figure 4.6, it should be evident that the performance of the FS(*T*) algorithm was much closer to the performance of the baseline algorithm than was that of the FS(*F*) algorithm. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 4.3 is actually correct—focusing search on the temporal variables significantly improves performance.

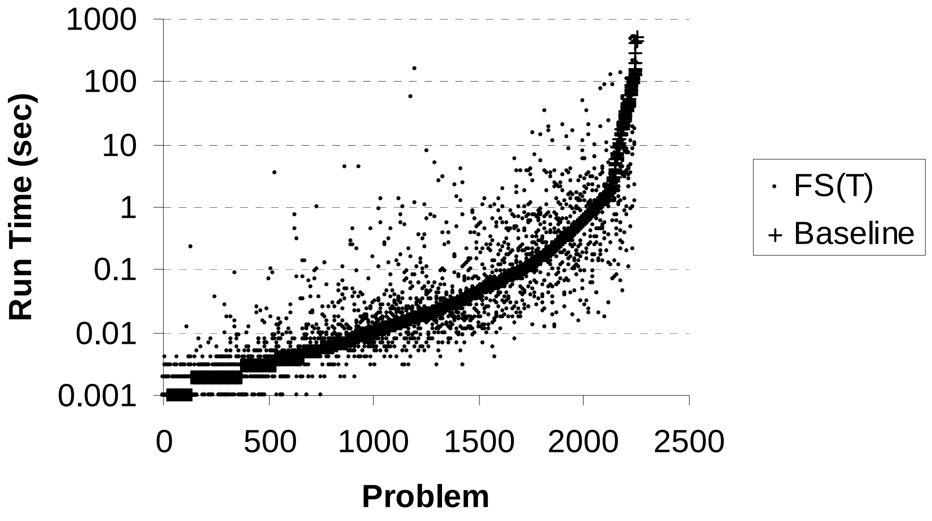

Figure 4.7. Run time of randomly generated HSPs using the baseline algorithm and focusing search on temporal variables.

Figure 4.8 provides a view of the relative scale of the effects of focusing search by showing the average percent improvement over the baseline run time using FS(*F*) and FS(*T*). As can be seen in this figure, for this set of randomly generated HSPs, focusing search on the finite-domain variables hurts efficiency considerably, increasing run time by 229.0% on average, while focusing search on the temporal variables improves efficiency slightly, reducing run time by 22.7% on average.

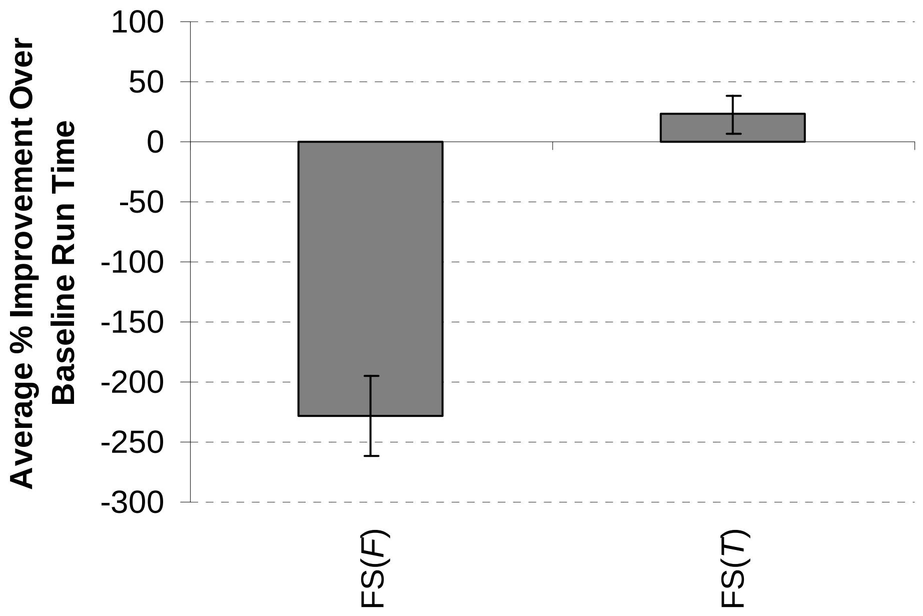

Figure 4.8. Average percent improvement over baseline run time by focusing search on finite-domain and temporal variables when solving randomly generated HSPs.

The results of Figure 4.8 provide several insights. First, they show that focusing search on the appropriate variable type can in fact improve efficiency. Second, they show that, for this set of randomly generated HSPs, the temporal variables are on average more tightly constrained than the finite-domain variables, despite a deliberate attempt to balance the tightness of constraints (on average) based on the data shown in Tables 4.1 and 4.2. Finally, focusing search on one variable type improved efficiency while focusing search on the other variable type hurt efficiency, which means that the efficiency gain of focusing search is due to its effect on the variable selection heuristic rather than a reduction in thrashing.

4.3.2. Testing FS with Air Traffic Control Problems

Focusing search on the appropriate variable type has proven effective when solving randomly generated problems, but how will it perform when applied to structured air traffic control problems? As pointed out in the introduction, air traffic control problems are so under-constrained that they can almost always be solved with the backtrack-free algorithm implemented in CTAS. In fact, when raw air traffic control problems were tested using my meta-SAT implementation, they were solved in a fraction of a second with almost no backtracking, no matter where search was focused.

In order to generate air traffic control problems that are constrained tightly enough to require backtracking, I augmented the original problems with simulated finite-domain and temporal constraints. The overall process of augmenting the raw air traffic control data is very similar to the process that was presented in Chapter 3. Each augmented problem is created by first randomly selecting a snapshot of constraints from the same set of snapshots used for the experiments on dynamic techniques. Because of the additional complexity that results from the simulated constraints, it was necessary to limit the problem size to complete the experiments in a manageable amount of time. Given the selected snapshot, 15 aircraft were randomly selected to serve as the basis of the problem, and the constraints of all other aircraft were discarded.

Given the constraints on 15 aircraft at a single moment in time, I augmented the temporal constraints by adding ordering requirements between the runway arrival times of pairs of aircraft. As in Chapter 3, two aircraft are randomly selected, an ordering between them is chosen randomly, and a temporal difference constraint is created that requires one aircraft to land before the other. This process is repeated to create a second temporal difference constraint, and these two TD constraints are disjoined to create a single disjunctive temporal constraint. The disjunctive temporal constraint requires that the schedule satisfy at least one of the two ordering constraints between the two pairs of aircraft.

To augment the finite-domain constraints, I simulated an interaction between the runway assignments of aircraft. For every subset of three aircraft, I listed all possible combinations of runway assignments. I randomly selected a fraction of these combinations and, for each combination, added a finite-domain constraint to prevent the assignment of these aircraft to these runways. In essence, each constraint states that if two of the aircraft are assigned to land on particular runways, then there is a particular runway on which the third aircraft is not allowed to land. As with the temporal constraints, it does not matter whether the finite-domain constraints make sense in the context of an air traffic control problem; it only matters that they have structure. These simulated finite-domain constraints are structured in that they only affect the runway assignments. It is impossible, for example, for one of these simulated constraints to assert a relationship between the meter fix assignments of aircraft.

As with the randomly generated problem set, I performed experiments to calculate the number of constraints that must be added to the finite-domain and temporal subproblems to achieve 10, 30, 50, 70, and 90 percent consistency. For each consistency level of each problem, I generated a set of 20 augmented air traffic control problems, for a total of 500 problems. If a problem contained a local inconsistency, it was discarded from the set and replaced.

Recall that Hypothesis 4.2 predicted that focusing search on the finite-domain variables first would improve efficiency over the baseline algorithm, and Hypothesis 4.3 predicted that focusing search on the temporal variables first would improve efficiency. To test these hypotheses further, I solved each of the augmented air traffic control problems with each of the three algorithms—baseline, FS(*F*), and FS(*T*). Figure 4.9 compares the run time of the baseline algorithm against the run time of FS(*F*), and Figure 4.10 compares the run time of the baseline algorithm against the run time of FS(*T*). In both figures, the problems are arranged in ascending order according to the run time of the baseline algorithm. Each plus sign plots the baseline run time, and each point plots the run time when the search is focused.

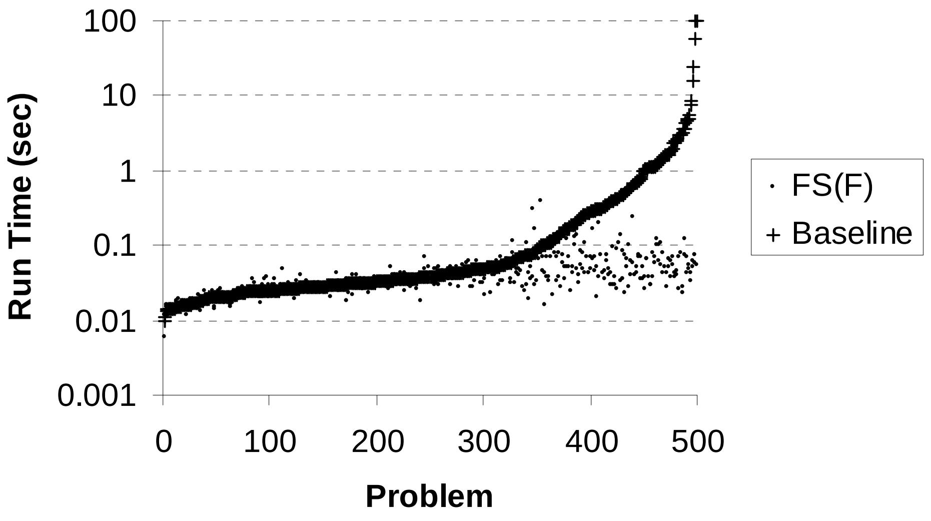

Figure 4.9. Run time of augmented air traffic control HSPs using the baseline algorithm and focusing search on finite-domain variables.

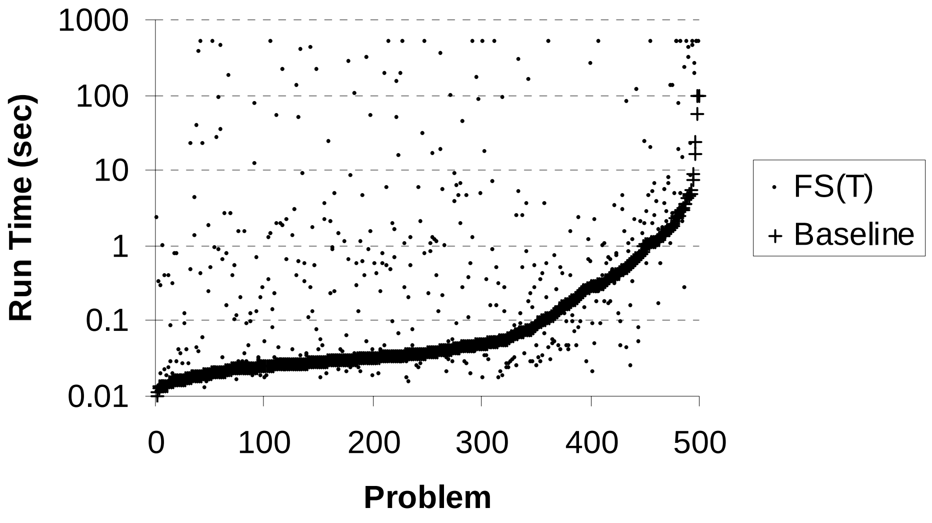

Figure 4.10. Run time of augmented air traffic control HSPs using the baseline algorithm and focusing search on temporal variables.

As can be seen in these figures, focusing the search on the finite-domain variables is clearly faster than the baseline algorithm, while focusing the search on the temporal variables is clearly slower. Student's paired t-tests confirm with 99.5% that, for this set of augmented air traffic control problems, Hypothesis 4.2 is correct, and Hypothesis 4.3 is not. While these findings are opposite of those for the randomly generated problems in Section 4.3.1, this is simply because the problem domain is different. The fact that focusing search on one subset of variables reduces run time while focusing search on the other subset increases run time once again confirms that the efficiency gain of focusing search is due to its effect on the variable selection heuristic and not a reduction in thrashing.

Figure 4.11 compares the average percent improvement over the baseline run time by focusing search when solving air traffic control problems. By focusing search on the finite-domain variables, run time was reduced by 95.2% on average, meaning that this approach averaged more than 20 times faster than the baseline. On the other hand, by focusing search on the temporal variables, run time was increased by 4104.2% on average, meaning that this approach averaged over 40 times slower than the baseline. For this test set of air traffic control problems, focusing search has a much greater impact on efficiency than it did for the randomly generated problems of Section 4.3.1.

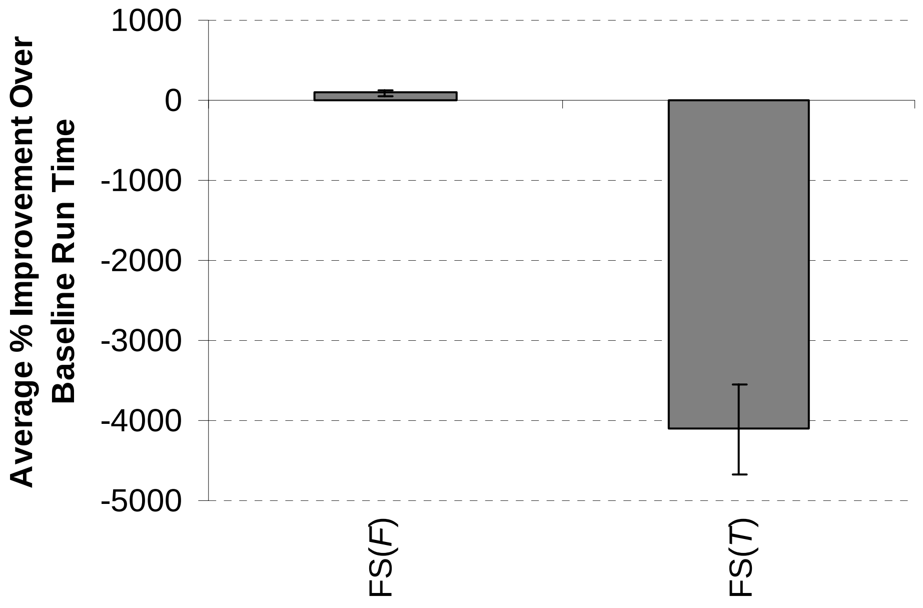

Figure 4.11. Average percent improvement over baseline run time by focusing search on finite-domain and temporal variables when solving augmented air traffic control HSPs.

How can we tell that these results are not simply an artifact of the simulated constraints? These augmented problems contain no local inconsistencies, so any inconsistencies must be a result of the interaction between the finite-domain and temporal variables through the hybrid constraints. The only simulated constraints added to these problems are either purely temporal or purely finite-domain—all of the hybrid constraints come directly from the raw air traffic control recordings. Hence, the complexity relies, at least partially, on the actual structure of real air traffic control constraints. The results of these experiments show that focusing search can have a dramatic impact on efficiency when solving structured, real-world problems.

4.4. Choosing the Most-Constrained Subset

Focusing search can improve efficiency slightly, but the approach of choosing one ordering of variable subsets to apply to every problem for a given domain has its drawbacks. First, determining which subset to place first in the ordering for a particular problem domain can be a time-consuming and expensive task. Second, applying the same subset ordering to every problem imposes a limit on the possible efficiency gains. While efficiency may be improved for a majority of the problems, there may still be a large number of problems in which the first subset in the ordering is less constrained than the second, and so efficiency suffers.

A better approach might be to somehow measure the relative tightness of the finite-domain and temporal constraints in order to focus search on the most-constrained subset (MCS) of variables. In my research, I consider three possible variable subsets that arise naturally out of the structure of an HSP: 1) the finite-domain variables *F*, 2) the temporal variables *T*, and 3) the set of all variables *A* (which can be thought of as a degenerate case of a subset). Each variable subset corresponds to a subset ordering—FS(*F*) places the subset of finite-domain variables at the beginning of the ordering, FS(*T*) places the subset of temporal variables at the beginning of the ordering, and FS(*A*) does not split the variables into subsets, which is equivalent to the baseline algorithm.

Choosing the most-constrained subset can be divided into two steps. First, it is necessary to define a metric of constraint tightness in order to measure the relative tightness of the finite-domain and temporal constraints of each HSP. In Figure 4.1, such a metric would provide values for the axes, making it possible to plot each individual HSP somewhere on the graph. Second, it is necessary to devise a classifier to map the tightness measurements of an individual HSP to a particular subset ordering. Such a classifier would partition the problem space of Figure 4.1 into regions, with one region corresponding to each possible subset ordering. One would expect FS(*T*) to perform best on HSPs near the lower right corner, FS(*F*) to perform best on HSPs near the upper left corner, and FS(*A*) to perform best on HSPs near the main diagonal. If the metric of constraint tightness and the mapping can both be defined appropriately, then it should be possible to choose the best subset ordering for each HSP and improve efficiency overall.

4.4.1. Metrics of Constraint Tightness

If any metric of constraint tightness is to be effective, it must be sufficiently accurate and efficient such that the time spent computing it does not exceed the time it saves by choosing better subset orderings. One way to measure the tightness of constraints would be to examine the constraints directly. One could certainly define some local measure of the tightness of an individual constraint, and then define a global measure of tightness to aggregate them. This would be similar to the family of temporal stability metrics I introduced in Section 3.5.1. Before any of these temporal stability metrics could be employed, it was necessary to represent the induced constraints of an STP in a canonical form (I chose to use the d-graph). All of the induced constraints of an STP can be determined in polynomial time; determining all of the induced constraints of a DTP or finite-domain CSP could require exponential time. Measuring tightness based on an aggregation of local measures is therefore not an option for the temporal and finite-domain constraints of an HSP.

The problem of ordering subsets within an HSP is actually quite similar to the problem of ordering agents within a distributed CSP (DisCSP). In a DisCSP, a set of agents work together to solve a single CSP. Each agent is responsible for a set of variables or a set of constraints, defining a local CSP that is a subproblem of the complete CSP being solved by the group. The constraints contained entirely within the subproblem of a single agent are called *intra-agent constraints*, and the constraints between agents are called *inter-agent constraints*. As each agent works to satisfy its own intra-agent constraints, it communicates with other agents in the group to ensure that the inter-agent constraints are satisfied as well.

As with a standard (centralized) CSP, the order in which variables are assigned values can have a dramatic effect on efficiency when solving a DisCSP. A common approach in the DisCSP literature is to split the variable ordering process into two phases—first, an inter-agent ordering heuristic chooses the next agent that will solve its subproblem, and second, an intra-agent heuristic chooses the next variable for which the chosen agent will assign a value. One can think of an HSP in the context of a DisCSP in which the finite-domain constraints are the intra-agent constraints of one agent, the temporal constraints are the intra-agent constraints of a second agent, and the hybrid constraints are the inter-agent constraints between them. In this context, an inter-agent ordering heuristic is analogous to a subset ordering heuristic, and an intra-agent ordering heuristic is analogous to a variable ordering heuristic.

Davin and Modi (2006) split the variable ordering process into an inter-agent ordering phase followed by an intra-agent ordering phase. Their inter-agent heuristic chooses the agent with the most inter-agent constraints to already ordered agents, breaking ties by counting the number of inter-agent constraints to unordered agents. While this inter-agent heuristic is useful when there are more than two agents, if the DisCSP representation of an HSP has only two agents (a finite-domain agent and a temporal agent), it provides no useful information because these counts will always be equal for both agents. Essentially, this is a degenerate case of the situation for which the heuristic was intended. As pointed out earlier, an HSP could be broken down into more than two subproblems (e.g., a third subproblem might represent the hybrid constraints). The inter-agent heuristic presented by Davin and Modi (2006) might provide more valuable information if the DisCSP representation of an HSP has more than two agents, but I will leave this for future work.

Salido, Giret, and Barber (2003) also split the variable ordering into an inter-agent heuristic and an intra-agent heuristic. They perform a pre-processing phase based on random sampling to test the tightness of the constraints of each agent's subproblem. I will leave the idea of random sampling for future work, but as I will describe shortly, I will use the idea of a pre-processing phase to measure the relative tightness of sets of constraints.

Armstrong and Durfee (1997) tested several methods for ordering agents with inter-agent heuristics, including the number of nogoods discovered for each agent, a weighted average over the domain sizes for each agent, and the number of solutions for each agent, which they found performed best. Although they do not test the idea, the authors suggest that the number of solutions (which might be exponential in the number of variables) could be estimated by testing some subset of the local problem for solutions. I will discuss shortly how I estimate the number of solutions to a subproblem as part of my measure of constraint tightness. Additionally, in the experiments of (Armstrong and Durfee 1997), the problem size of each agent's subproblem was equal, whereas the number of complete assignments for the finite-domain and temporal subproblems that I am testing can vary widely. Based on the number of solutions to each subproblem, but accounting for the differences in problem size, I measure the tightness of the constraints over a subset of variables in terms of solution density:

Definition 4.3. The *solution density* of a CSP is the fraction of complete assignments that are consistent. That is, solution density is equal to the number solutions divided by the number of complete assignments.

Determining the number of complete assignments is a simple task—it is the product of the domain size of each variable. Determining the number of solutions, on the other hand, is a very difficult task. Directly counting all of the solutions could require an exponential amount of time, so previous work has attempted to estimate the number of solutions to a CSP (Roth 1996; Kask, Dechter, and Gogate 2004). Gomes, Sabharwal, and Selman (2006) introduce a novel method for counting solutions in which an estimate is produced based on the number of additional XOR constraints that must be added to a SAT formula before it becomes inconsistent. This method can provide guarantees on the quality of the solution count, but it operates by repeatedly solving a large number of subproblems, and so it is not efficient enough for my purposes.

Kilby, *et al.* (2006) explore the related problem of estimating the size of a subtree during chronological backtracking search. The difference is that this problem estimates the total number of nodes in a search tree, whereas solution counting estimates the number of consistent leaf nodes in a search tree. By estimating the size of a search tree, the authors were able to predict the relative run time of several different search algorithms, choose the algorithm with the best predicted run time, and thereby improve efficiency. Similarly, I am attempting to improve efficiency by predicting the relative run time of several different subset orderings.

One of the methods presented in (Kilby, *et al.* 2006) assumes that backtrack nodes are distributed evenly throughout the search tree. Based on this assumption, this method counts the exact number of nodes in part of the search tree in one branch of a binary node, and then estimates that the other branch contains exactly the same number of nodes. With this simple approach, the authors were able to choose the best algorithm often enough to improve efficiency significantly.

I am attempting a very similar approach. While Kirby, *et al.* were attempting to predict which search algorithm would perform best, I am attempting to predict which variable subset ordering will perform best. They rank possible algorithms using estimates based on the assumption that backtrack nodes are distributed evenly throughout the search space. I will rank possible subset orderings based on the similar assumption that solutions are distributed evenly throughout the search space.

Although the use of heuristics might make it very unlikely that the assumption that solutions are distributed evenly throughout the search space is correct, it is important to note that a solution density estimator can effectively determine which subset ordering will be most efficient even if the estimator is not particularly accurate. Kilby, *et al*. (2006) point out that even a simple estimator can still be a useful one (page 1018):

> *One of the features of this problem is that the estimation methods only need to rank accurately. A method can, for example, consistently underestimate search tree size and still perform well at selecting the best algorithm.*

Even though the authors used a simple estimator based on the assumption that backtrack nodes are distributed evenly throughout the search space, they were able to select the best algorithm often enough to significantly improve efficiency. This same reasoning can be applied to choosing subset orderings for HSPs. I will use a simple estimator based on the assumption that solutions are distributed evenly throughout the search space, and so it is reasonable to expect that I will be able to select the best subset ordering often enough to significantly improve efficiency as well.

The process I use for estimating solution density is as follows. While searching for a solution, some fraction *f* of the search tree is pruned. After finding the first solution to a problem, it would be possible to continue to search for more solutions. Based on the assumption that solutions are distributed evenly throughout the search space, one should expect that the same fraction *f* of the search tree will be pruned before finding each of these subsequent solutions. I therefore estimate that the total number of solutions is 1 / *f*. For example, if one tenth of the search space is pruned before finding the first solution, then I estimate that the problem has ten solutions.

For each variable v_i of a finite-domain CSP, let d_i be the size of its domain, and let t_i be the number of values tested before finding a solution (that is, t_i equals the number of values pruned from the domain plus the one value selected in the solution assignment). The fraction of values tested for variable v_i is t_i / d_i. Given a problem P with n variables, the total fraction of the search tree that is pruned before finding the first solution, Fraction(P), is:

$$\text{Fraction}(P) = \prod_{i=1}^{n} (t_i / d_i) \tag{4.1}$$

With the assumption that this same fraction will be pruned before finding each subsequent solution, the estimated number of solutions to P, ENS(P), can be calculated with the following formula:

$$\begin{aligned} \text{ENS}(P) &= 1 / \text{Fraction}(P) \\ &= 1 / \prod_{i=1}^{n} (t_i / d_i) \\ &= \prod_{i=1}^{n} (d_i / t_i) \end{aligned} \tag{4.2}$$

As stated in Definition 4.3, solution density is equal to the number solutions divided by the number of complete assignments. The estimated solution density of problem P, ESD(P), is equal to the estimated number of solutions, ENS(P), divided by the product of the domain size of each variable:

$$
\begin{aligned}
\mathrm{ESD}(P) &= \mathrm{ENS}(P) \,/\, \textstyle\prod_{i=1}^{n} d_i \\
&= \textstyle\prod_{i=1}^{n} (d_i / t_i) \,/\, \prod_{i=1}^{n} d_i \\
&= \textstyle\prod_{i=1}^{n} d_i \,/\, (\prod_{i=1}^{n} t_i \prod_{i=1}^{n} d_i) \\
&= 1 \,/\, \textstyle\prod_{i=1}^{n} t_i \qquad (4.3)
\end{aligned}
$$

In (4.3), note that the numerator is the number of solutions found (in this case, just one), and the denominator is the number of complete assignments pruned along the way. In other words, (4.3) is the formula for the *exact* solution density of the subtree searched before finding the first solution, and I am using the exact solution density of this subtree as an estimated solution density of the complete search tree.

In the context of an HSP, this particular estimate of solution density can capitalize on the searches performed while attempting to detect local inconsistencies in the finite-domain and temporal subproblems. By keeping track of the number of values tested for each variable before finding a solution to a subproblem, the solution density of that subproblem can be estimated with essentially no additional overhead. In effect, I am using a pre-processing phase to measure the relative tightness of the constraints on the subproblems, as is common in the DisCSP literature (e.g., Durfee and Armstrong 1997; Salido, Giret, and Barber 2003). For my purposes, this method for estimating solution density is extremely efficient.

4.4.2. MCS Classifiers

Given an appropriate measure of constraint tightness, it is still necessary to devise a classifier to map from the tightness measurements of an individual HSP to a particular subset ordering. A perfect classifier would choose the most efficient subset ordering for every problem instance. It is possible to measure the performance of such a perfect classifier by first solving each HSP with each of the subset orderings and then selecting the ordering with the best run time after the fact. The perfect classifier provides an upper bound on the efficiency gains that are possible using this approach.

Figure 4.12 compares the run time of the baseline algorithm (which does not split the variables into subsets) against the run time of the perfect classifier (which chooses the best of FS(*F*), FS(*T*), and FS(*A*)) based on the same set of randomly generated HSPs with consistent subproblems used in Section 4.3. The problems are arranged in ascending order according to the baseline run time. Each plus sign plots the baseline run time, and each dot plots the run time of the perfect classifier. Figure 4.12 shows (and a Student's paired t-test confirms with 99.5% confidence) that the perfect classifier is significantly faster than the baseline algorithm. The average run time using the baseline algorithm is 3.10 seconds per problem, while the average run time using the perfect classifier is 0.83 seconds per problem, just 26.7% of the average baseline run time. These times include the average of 0.16 seconds per problem spent solving the finite-domain and temporal subproblems.

To define a classifier that predicts the most efficient subset ordering without actually solving the complete problem, one could simply compare the estimated solution densities of the finite-domain and temporal subproblems directly, placing the subset of variables with the lower estimated solution density at the beginning of the ordering, or using the baseline algorithm when the solution density estimates are sufficiently similar. Recall from Section 4.3 that one of the drawbacks of the most-constrained variable heuristic when applied to HSPs is that the measures of constraint tightness for the different variable types could be operating on different scales, so comparing finite-domain variables to temporal variables could be comparing apples to oranges. Similarly, the heuristics applied to the finite-domain and temporal subproblems might lead to radically different scales when estimating solution density. For example, while the estimated solution densities of the two subproblems might be equal, because they are operating on different scales, it should be interpreted to mean that the constraints of one subproblem are very tight while the constraints of the other subproblem are very loose.

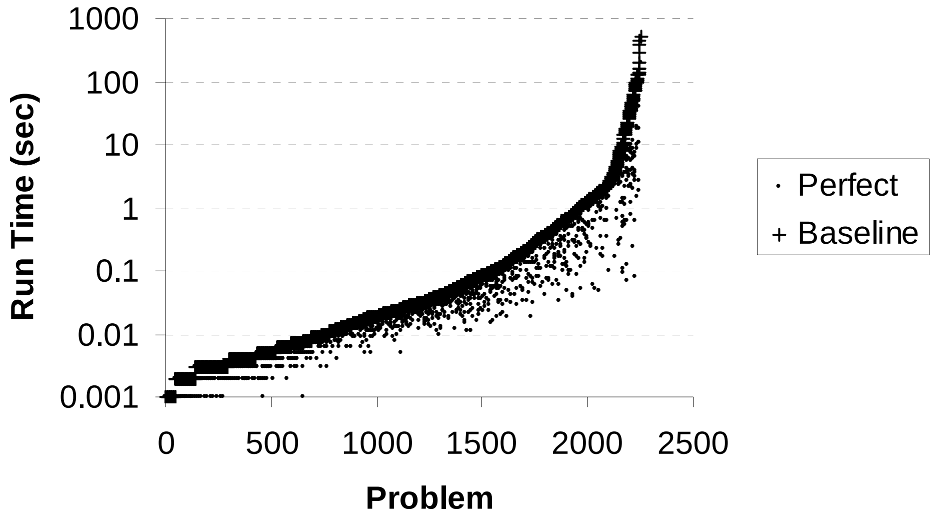

Figure 4.12. Run time of randomly generated HSPs using the baseline algorithm and the perfect classifier.

To account for the possibility of discrepancies in the scale of the solution densities of subproblems with different variable types, I will make the assumption that a linear relationship exists between the estimated solution density of the subproblems and the run time using each subset ordering. This assumption follows the basic logic behind focusing search—focusing search on the finite-domain variables should decrease run time when the finite-domain constraints are tighter and increase run time when the temporal constraints are tighter, and vice versa for the temporal variables. Based on this assumption, it is possible to use linear regression to generate a mapping function from the estimated solution densities to a predicted run time for each subset ordering. In other words, the linear regressions will map the measures of constraint tightness from potentially different units of measure (estimated solution densities operating on different scales) to a common unit of measure (run time). A classifier can then use these predictions by mapping estimated solution densities to the subset ordering with the lowest predicted run time. I will refer to such a classifier that uses a linear regression of estimated solution densities as the LR(ESD) classifier.

4.4.3. Testing MCS with Randomly Generated Problems

Based on the LR(ESD) classifier, I will test the following hypothesis:

Hypothesis 4.4. Using the LR(ESD) classifier to choose subset orderings will significantly improve efficiency over the baseline algorithm.

To create the LR(ESD) classifier, I performed linear regressions on the set of 2,250 randomly generated HSPs with consistent subproblems from Section 4.3. First, I solved the finite-domain and temporal subproblems of each HSP, measuring the estimated solution density as in formula (4.3). Some of the estimated solution densities were such small fractions that the program used to perform the linear regressions had difficulty handling them, so I used the logarithms of the estimated solution densities as the observed inputs. Next, I solved each complete HSP using each of the three subset orderings and used the measured run times as the observed outputs. I then performed three linear regressions, one for each subset ordering. The result of each linear regressions is a set of coefficients a, b, and c, such that the predicted run time of subset ordering X, PRT(FS(X)), can be calculated with the following formula:

$$\mathrm{PRT}(\mathrm{FS}(X)) = a \log(\mathrm{ESD}(F)) + b \log(\mathrm{ESD}(T)) + c \tag{4.4}$$

The actual coefficients produced by these linear regressions are listed in Table 4.3.

	a	***b***	***c***
	log(ESD(*F*))	**log(ESD(*T*))**	**1**
PRT(FS(*F*))	1.328	0.421	-30.335
PRT(FS(*T*))	0.467	0.091	8.165
PRT(FS(*A*))	-0.282	0.129	8.121

Table 4.3. Coefficients for calculating predicted run time as derived from linear regressions based on estimated solution densities.

Figure 4.13 shows how these formulas partition the problem space. The horizontal axis measures the tightness of temporal constraints (with tighter constraints to the left and looser constraints to the right), while the vertical axis measures the tightness of the finite-domain constraints (with tighter constraints to the bottom and looser constraints to the top). The values here are negative because constraint tightness is measured as the logarithm of estimated solution density, and solution densities are always fractions between 0 and 1. Note that the estimated solution densities of finite-domain and temporal subproblems can be scaled very differently. Each point on the graph represents an individual HSP. The visible clusters of HSPs are a result of the parameters chosen to randomly generate the problems in the set. The three lines show how the LR(ESD) classifier partitions the search space. In this figure, FS(*T*) has the lowest predicted run time in the upper left region, FS(*F*) has the lowest predicted run time in the upper right region, and FS(*A*) has the lowest predicted run time in the lower region.

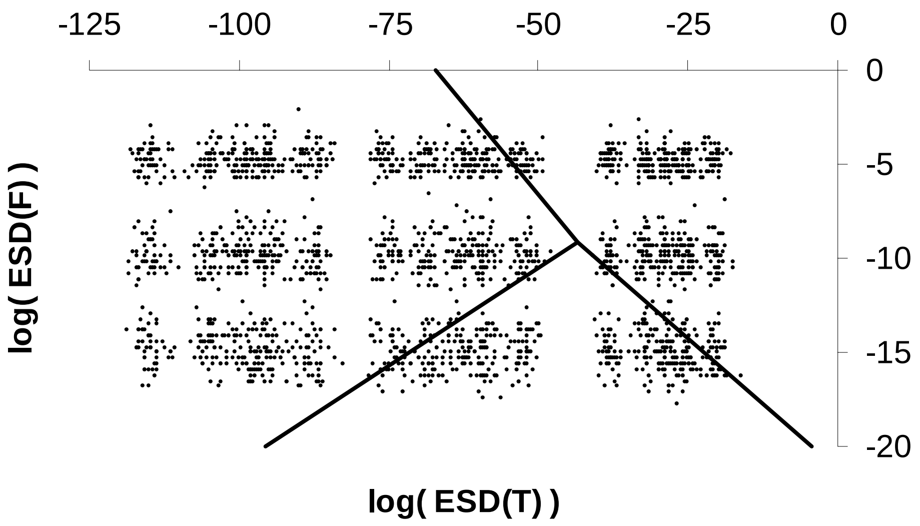

Figure 4.13. Problem space of randomly generated HSPs partitioned by the LR(ESD) classifier.

To test Hypothesis 4.4, Figure 4.14 compares the run time of each HSP using the LR(ESD) classifier to the run time using the baseline algorithm. Each plus sign plots the run time of the baseline algorithm, each point plots the run time of the LR(ESD) classifier, and the problems are arranged in ascending order according to the baseline run time. The run times include the time spent solving the subproblems to detect local inconsistencies and to estimate solution densities, but they do not include the time spent deriving the linear regressions; for now, it is assumed that these linear regressions are derived offline beforehand. The average baseline run time is 3.10 seconds per problem, while the average run time of the LR(ESD) is 2.34 seconds per problem. This reduction in average run time is 33.4% of the upper limit as measured by the perfect classifier. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 4.4 is correct—using the LR(ESD) classifier to choose subset orderings significantly improves efficiency over the baseline algorithm.

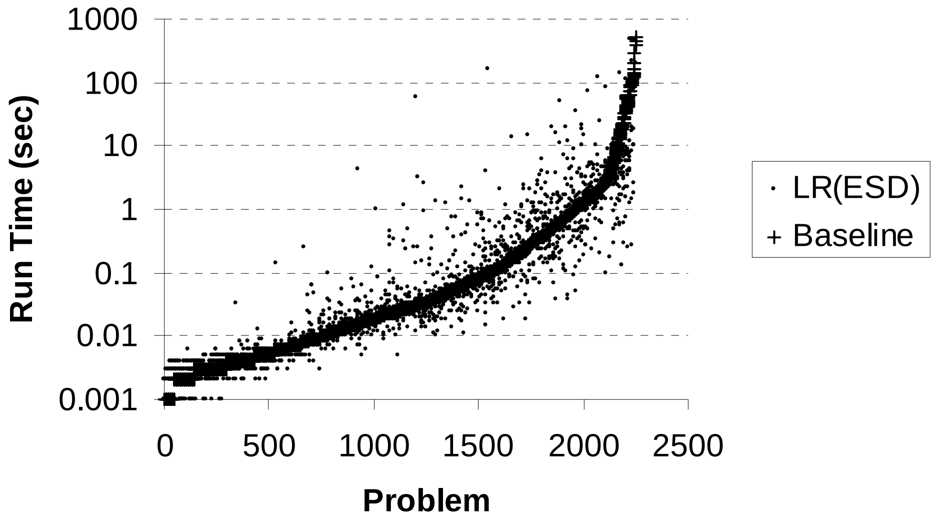

Figure 4.14. Run time of randomly generated HSPs using the baseline algorithm and the LR(ESD) classifier.

Even though solution density was defined to take into account the size of the search space, estimated solution density and search space size are not independent. Figures 4.15 and 4.16 show the relationship between problem size and estimated solution density. In these figures, each point represents one of the randomly generated HSPs in the problem set. Figure 4.15 plots the number of variables against the estimated solution density of the finite-domain subproblem of each HSP, and Figure 4.16 plots the number of variables against the estimated solution density of the temporal subproblem of each HSP. In both cases, estimated solution density decreases as problem size increases.

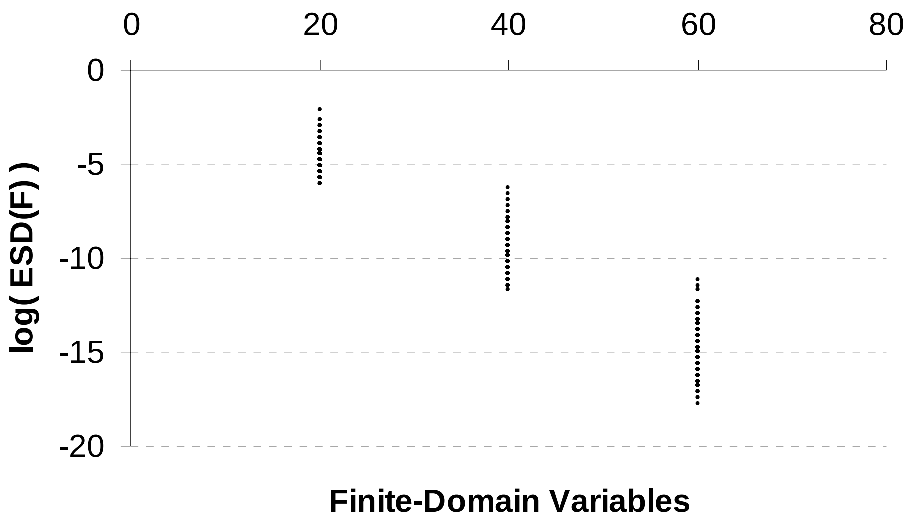

Figure 4.15. Relationship between the number of variables and the logarithm of the estimated solution density of the finite-domain subproblems of randomly generated HSPs.

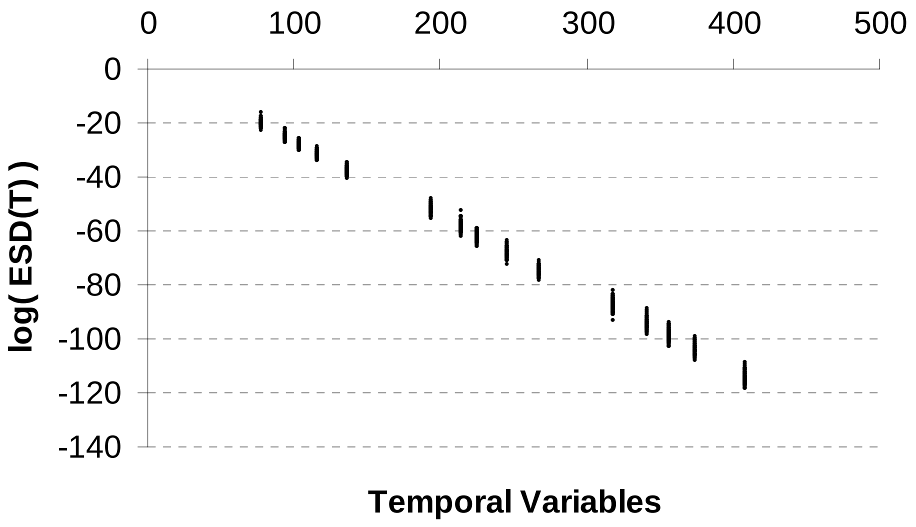

Figure 4.16. Relationship between the number of variables and the logarithm of the estimated solution density of the temporal subproblems of randomly generated HSPs.

This dependency between problem size and estimated solution density is unaccounted for in the LR(ESD) classifier, and so it could have an adverse effect on its performance. To remedy this, I performed a second set of linear regressions that use as observed inputs both the logarithm of the estimated solution densities and the number of variables of the finite-domain and temporal subproblems. That is, the problem space is organized along four dimensions instead of just two. As before, the observed outputs are the run times using each of the three subset orderings. Formula (4.5) shows the calculation for predicted run time based on estimated solution densities and number of variables, and Table 4.4 lists the coefficients that result from these linear regressions (Vars(*F*) and Vars(*T*) refer to the number of finite-domain and temporal variables, respectively).

$$\mathrm{PRT}(\mathrm{FS}(X)) = a \log(\mathrm{ESD}(F)) + b \log(\mathrm{ESD}(T)) + c \,\mathrm{Vars}(F) + d \,\mathrm{Vars}(T) + e \qquad (4.5)$$

	a **log(ESD(*F*))**	***b*** **log(ESD(*T*))**	***c*** **Vars(*F*)**	***d*** **Vars(*T*)**	***e*** **1**
PRT(FS(*F*))	4.504	-3.015	1.541	-0.735	-30.257
PRT(FS(*T*))	-0.597	-0.019	-0.034	0.020	-8.189
PRT(FS(*A*))	-0.026	-0.847	0.067	-0.203	-7.211

Table 4.4. Coefficients for calculating predicted run time as derived from linear regressions based on estimated solution densities and number of finite-domain and temporal variables.

Because this classifier is based on linear regressions that take into account both estimated solution density and number of variables, I will refer to it as the LR(ESD, Vars) classifier. The high dimensionality of the problem space on which this classifier is based makes it impossible to depict how it partitions the problem space, as was done for the LR(ESD) classifier in Figure 4.13. Using the LR(ESD, Vars) classifier, I will test the following hypotheses:

Hypothesis 4.5. Using the LR(ESD, Vars) classifier to choose subset orderings will significantly improve efficiency over the baseline algorithm.

Hypothesis 4.6. Using the LR(ESD, Vars) classifier to choose subset orderings will significantly improve efficiency over using the LR(ESD) classifier to choose subset orderings.

Figure 4.17 compares the run time using the LR(ESD, Vars) classifier to the run time using the baseline algorithm. Each plus sign plots the run time of the baseline algorithm, each point plots the run time using the LR(ESD,Vars) classifier (once again, not including the time required to derive the linear regressions offline), and all problems are sorted along the horizontal axis in ascending order according to the baseline run time. The average run time using the LR(ESD, Vars) classifier is 1.98 seconds per problem, which a Student’s paired t-test confirms with 99.5% confidence is significantly faster than the average of 3.10 seconds per problem using the baseline algorithm. Hypothesis 4.5 is correct—using the LR(ESD, Vars) classifier significantly improves efficiency over the baseline algorithm.

To test Hypothesis 4.6, Figure 4.18 compares the run time using the LR(ESD, Vars) classifier to the run time using the LR(ESD) classifier. Each plus sign plots the run time using the LR(ESD) classifier, each point plots the run time using the LR(ESD,Vars) classifier, and all problems are sorted along the horizontal axis in ascending order according to the LR(ESD) classifier run time. A Student’s paired t-test confirms with 99.5% confidence that the LR(ESD, Vars) classifier is significantly faster than the

LR(ESD) classifier. Whereas the LR(ESD) classifier was able to improve average run time by 33.4% of the upper limit as measured by the perfect classifier, the LR(ESD, Vars) classifier was able to improve run time by 49.5% of the upper limit of possible improvement. Improving performance this much is impressive considering the simplicity of the inputs to the linear regressions and the speed with which they can be computed. Hypothesis 4.6 is correct—by incorporating problem size into the linear regressions, using the LR(ESD, Vars) classifier significantly improves efficiency over the LR(ESD) classifier.

To give a relative perspective of performance, Figure 4.19 shows the average percent improvement over the baseline run time using each of the classifiers presented in this section. For this set of randomly generated HSPs, a perfect classifier would reduce run time by 77.4% on average relative to the baseline. The LR(ESD) classifier reduced run time by 25.8% on average, while the LR(ESD, Vars) classifier reduced run time by 38.3% on average. The error bars in Figure 4.19 represent standard error.

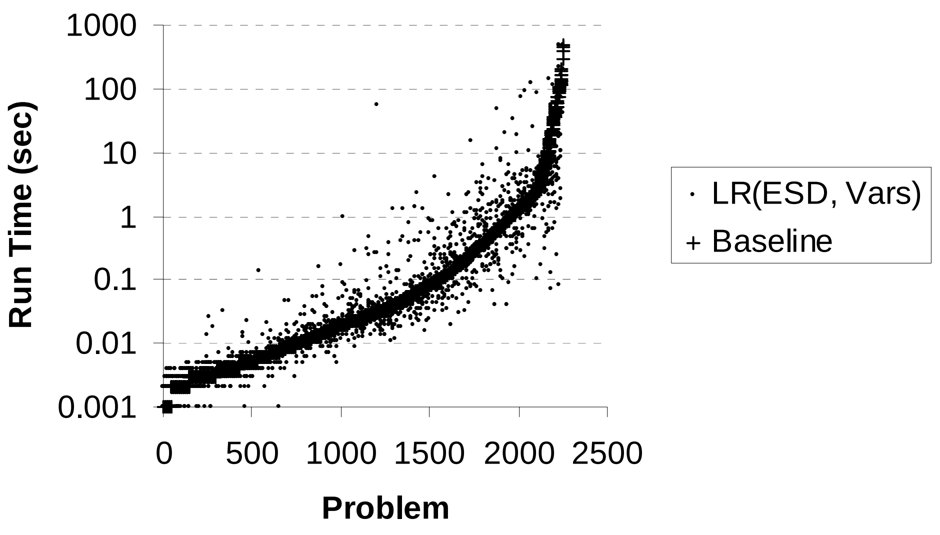

Figure 4.17. Run time of randomly generated HSPs using the baseline algorithm and the LR(ESD, Vars) classifier.

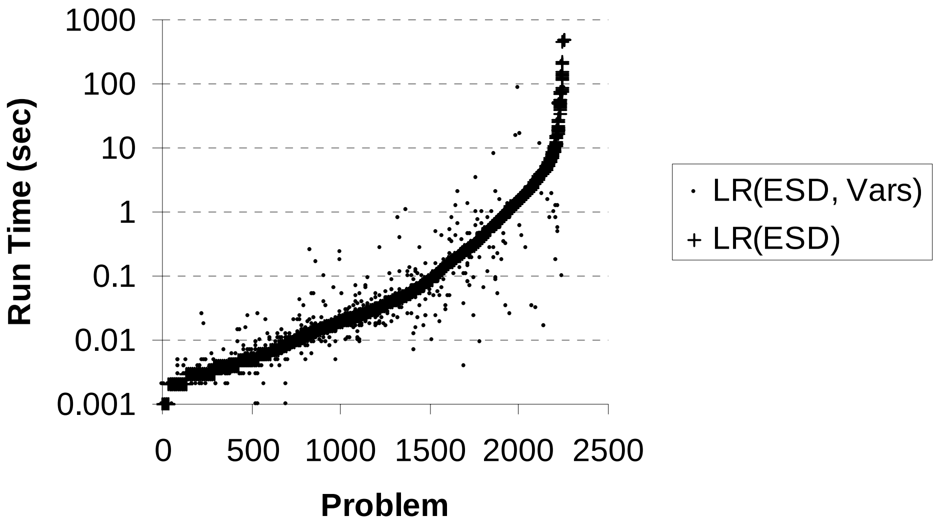

Figure 4.18. Run time of randomly generated HSPs using the LR(ESD) classifier and the LR(ESD, Vars) classifier.

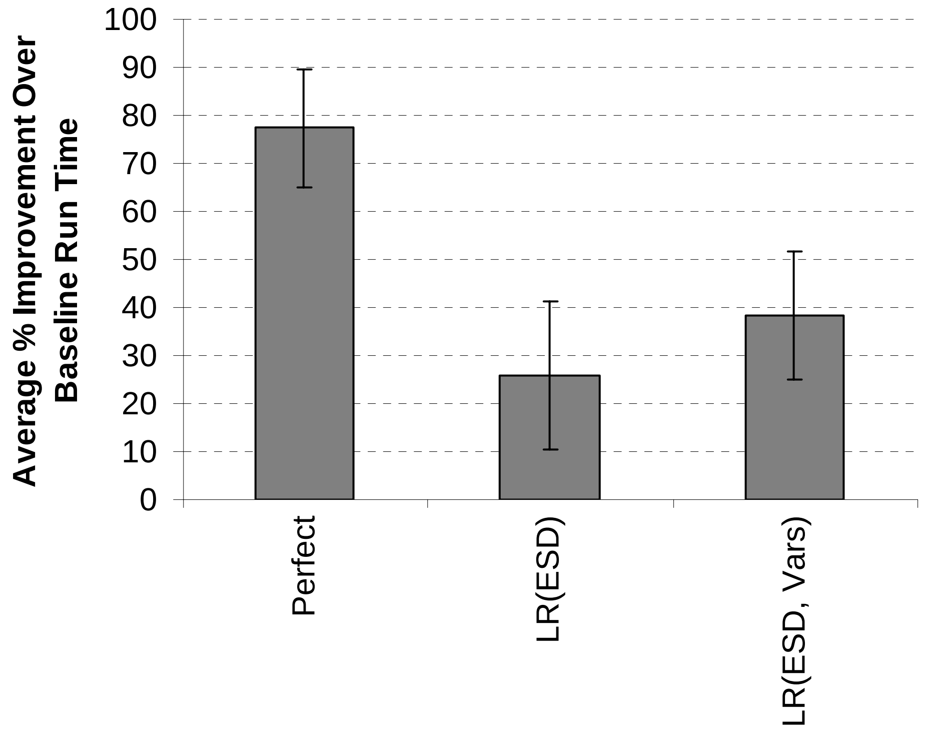

Figure 4.19. Average percent improvement over baseline run time using a variety of different classifiers to choose the most-constrained subset when solving randomly generated HSPs.

Figure 4.20 provides some perspective on the amount of time spent solving each subproblem versus the amount of time spent solving the complete HSPs. The black region of each bar represents the average time spent solving the finite-domain subproblem, the light gray region represents the average time spent solving the temporal subproblem, and the dark gray region represents the average time spent solving the complete HSP. The average time spent solving the finite-domain subproblems of this randomly generated set was only 0.001 seconds per problem, which is too small to see on this graph. The average time spent solving the temporal subproblems was 0.161 seconds per problem, which is still relatively small compared to the total average run time. A classifier can only affect the run time of the complete HSP, not the run time of the subproblems, so only the dark gray regions of Figure 4.20 are different. None of these problems contains a local inconsistency, so any reduction in run time is due entirely to the ability of the classifier to choose subset orderings intelligently. Figure 4.20 shows that the LR(ESD, Vars) classifier clearly outperforms the LR(ESD) classifier, and is almost exactly halfway between the baseline algorithm and the perfect classifier.

Choosing the most-constrained subset of each individual HSP offers several advantages over focusing search on the same subset for all HSPs. First, it avoids the potential problem of focusing the search on the less-constrained subset for many of the problems in the domain. Second, the inputs to the linear regressions could be learned online. That is, as each problem is solved, the estimated solution densities, number of variables, and observed run time can be added to the linear regression calculation. Because of the efficiency of computerized mathematical tools, the linear regressions can be updated very quickly. If necessary, the updates could be spaced apart to save time.

On the other hand, by focusing the search of every problem on the same variable subset, it would not be possible to learn which subset ordering performs best on average because the other subset orderings would never be tested.

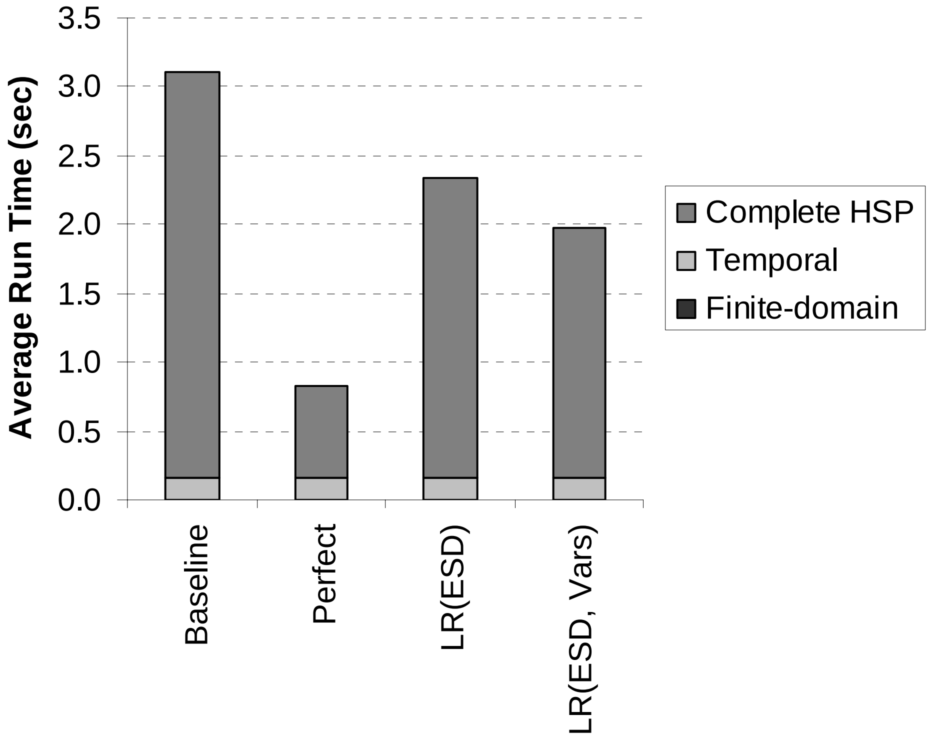

Figure 4.20. Average run time of randomly generated HSPs using the baseline algorithm and a variety of different classifiers to choose the most-constrained subset.

4.4.4. Testing MCS with Air Traffic Control Problems

Using this same approach, we can attempt to create a classifier for the augmented air traffic control problems from Section 4.3.2. Unfortunately, the FS(*F*) algorithm outperforms both the FS(*T*) algorithm and the FS(*A*) algorithm so often that there is almost no room for improvement. As was done for the randomly generated problems, one can observe the theoretical performance of a perfect classifier by measuring the run time of the best subset ordering after trying all three. Figure 4.21 compares the run time using the FS(*F*) algorithm against that which a perfect classifier would produce for the set of augmented air traffic control problems. Problems are arranged in ascending order according to the run time of the FS(*F*) algorithm. Each plus sign plots the FS(*F*) run time, and each point plots the run time of the perfect classifier.

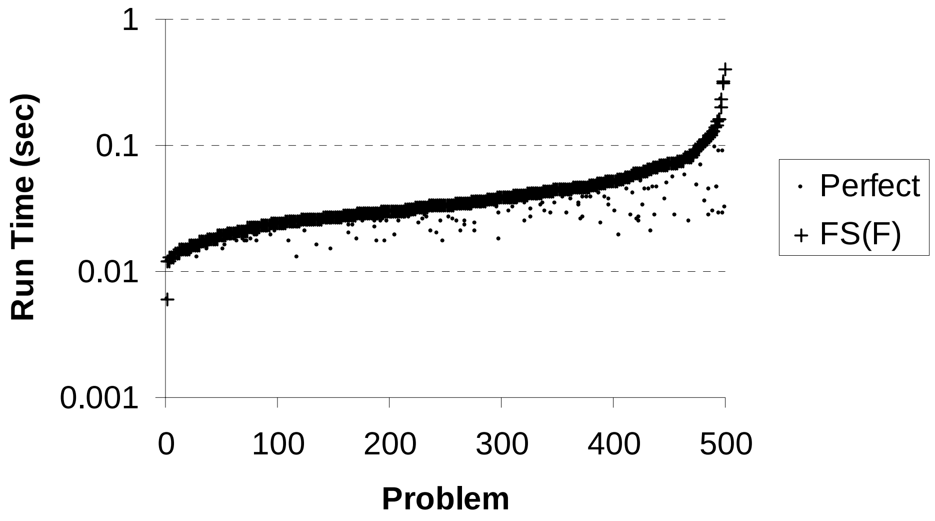

Figure 4.21. Run time of augmented air traffic control HSPs using the FS(*F*) algorithm and the perfect classifier.

As can be seen in this figure, run time can rarely be improved by choosing a subset ordering besides FS(*F*). In fact, the perfect classifier would only improve the average run time by 0.006 seconds per problem over FS(*F*). While this difference is statistically significant with 99.5% confidence according to a Student's paired t-test, even a single mistake by a classifier could increase the average run time by so much that it completely overwhelms any potential efficiency gains. This means that, for the air traffic control domain, it is reasonable to always assign the runways first before scheduling a time for each aircraft to pass through its reference points.

In fact, this is exactly how the scheduling algorithm in the CTAS Dynamic Planner (DP) is implemented. The developers of the DP designed the algorithm to mimic the process the air traffic controllers use to schedule incoming aircraft. When they spoke with the controllers, the controllers said they assign the runways first and assign times second, so this strategy is employed in the DP. Basically, what the DP developers were able to learn by speaking to domain experts, I was able to confirm empirically.

The fact that FS(F) outperformed FS(T) even when the temporal constraints were relatively tight and the finite-domain constraints were relatively loose indicates that the theory that the best variable subset ordering can be chosen based on the relative tightness of the constraints is incomplete. In the air traffic control domain, any two aircraft landing on the same runway must land in first-come-first-served order. Since the simulated temporal constraints were all ordering constraints, assigning the finite-domain variables in this case actually forced many of the assignments of the temporal variables. An assignment to a temporal variable, on the other hand, may or may not influence the finite-domain variable assignments, as this ordering can only be contradicted if the two aircraft are landing on the same runway. It would appear that the influence of the hybrid constraints in this case is *directional*—the finite-domain assignments influence the temporal assignments more than the temporal assignments influence the finite-domain assignments. It might be possible to improve the performance of a classifier by somehow measuring the directionality of hybrid constraints and incorporating that information, but I will leave this avenue of research for future work.

4.5. Conclusions

In this chapter, I defined the Hybrid Scheduling Problem, which is capable of representing complex scheduling problems that contain both finite-domain and temporal constraints. I showed that by splitting hybrid problems into finite-domain and temporal subproblems, it is possible to solve both subproblems in only a fraction of the time it takes to solve the complete problem. Based on the insight that all variables are not equal, I presented several techniques for improving efficiency when solving HSPs.

First, by solving the subproblems of an HSP before the complete problem, it is possible to quickly detect any local inconsistencies; if a local inconsistency is detected, run time is reduced significantly because there is need to solve the complete problem. Second, if no local inconsistencies are detected for a particular problem, efficiency can be further improved when solving the complete problem by focusing the search on either the subset of finite-domain or the subset of temporal variables. This was demonstrated empirically with both randomly generated problems and augmented air traffic control problems. Finally, by predicting run time with linear regressions based on the estimated solution densities and number of variables of the subproblems, it is possible to improve efficiency even more by focusing search on the most-constrained subset of variables for each individual HSP. With this approach, it was possible to improve efficiency by nearly half of the theoretical limit.

The real advantage of these techniques is that they can be applied across a variety of problem domains. If it is known that even a relatively small fraction of problems in a particular domain might contain a local inconsistency, then it should be possible to reduce average run time by solving the subproblems first to detect these local inconsistencies before attempting to solve the complete problem. If it is known that one subset of variables will generally be constrained more tightly than the other, then it should be possible to save time overall by focusing the search on the more tightly constrained subset. Finally, if it is possible to solve a sample of problems and perform a linear regression offline, then it should be possible to improve efficiency by solving the subproblems to detect local inconsistencies and estimate solution density, and then focusing search on the most-constrained subset. While the successful application of these hybrid techniques does require some knowledge of the domain, it is reasonable to expect that, in many domains, this knowledge can be acquired through a combination of discussions with domain experts and experimentation.

So far, I have shown that solving the subproblems as a preprocessing stage before solving the complete HSP serves two purposes—detecting local inconsistencies and estimating solution densities. The time spent solving the subproblems is so small relative to the time spent solving the complete problem that these two purposes both proved very useful. Detecting local inconsistencies can save time even if relatively few problems contain a local inconsistency. Estimating solution densities to choose the most-constrained subset can save time even if none of the problems contain a local inconsistency. In the next chapter, I will show how solving the subproblems first can improve efficiency even further when I combine the hybrid techniques of this chapter with the dynamic techniques of Chapter 3.

Chapter 5

Combining Dynamic and Hybrid Techniques

5.1. Introduction

At this point, I have described techniques for handling both dynamic constraints (in Chapter 3) as well as hybrid constraints (in Chapter 4). In this chapter, I introduce methods for using these techniques in combination. First, I will discuss how dynamic techniques can be applied to improve efficiency when solving static HSPs. In particular, I will empirically demonstrate the effectiveness of recording nogoods when solving the subproblems of an HSP. Second, I will discuss ideas for handling scheduling problems with hybrid constraints that change over time.

5.2. Applying Dynamic Techniques to Static HSPs

Recall from Chapter 4 that an HSP can be broken into two subproblems, a finite-domain subproblem and a temporal subproblem. The constraints of each subproblem are a subset of the constraints of the complete HSP; therefore, the complete HSP is a restriction of each of its subproblems. It is possible to think of solving a subproblem and then the complete problem as a dynamic sequence—the subproblem represents the initial problem in the sequence, and the complete HSP represents the modified problem. By casting a hybrid problem as a dynamic problem in this manner, it becomes evident that dynamic techniques can be applied when solving static HSPs.

As described in Section 4.2, one can detect local inconsistencies by solving each subproblem of an HSP before solving the complete problem. Thinking of the HSP as a dynamic problem, if a local inconsistency is detected in a subproblem, then the subproblem justification can be reused as a justification for the complete problem because the complete problem is a restriction of the subproblem. On the other hand, if no local inconsistency is detected, it is necessary solve the complete HSP, and the time spent solving the subproblems is wasted. In terms of a dynamic problem, this is an initially consistent problem that has been restricted, so it should be possible to improve efficiency by recording subproblem nogoods and subproblem oracles. The question is: How much of the time wasted by unsuccessfully trying to detect local inconsistencies can be recovered using dynamic techniques?

5.2.1. Recording Subproblem Nogoods

Because the complete HSP is a restriction of its subproblems, any nogood recorded while solving a subproblem can be applied when solving the complete problem. As in dynamic problems, these recorded nogoods should improve efficiency by quickly identifying dead-end branches while solving the complete HSP. I will refer to this approach as recording subproblem nogoods (RSN). Like detecting local inconsistencies (DLI), the process of RSN begins by solving the finite-domain subproblem. If this subproblem is consistent, then all of the nogoods recorded while solving it are stored. Next, the temporal subproblem is solved. If the temporal subproblem is consistent, then all of the nogoods recorded while solving it are also stored. At this point, no local inconsistencies were detected, so all of the nogoods recorded while solving the subproblems are added to the complete problem, and the complete problem is solved.

Recording subproblem nogoods requires no overhead beyond that of detecting local inconsistencies, so it is reasonable to expect that RSN will reduce the amount of time wasted by DLI, as suggested in Hypothesis 5.1:

Hypothesis 5.1. Recording subproblem nogoods will significantly improve efficiency over detecting local inconsistencies when solving static HSPs.

To test this hypothesis, I implemented RSN in same solver used in Chapters 3 and 4, and then used it to solve the same set 2,250 HSPs with consistent subproblems from Section 4.3.1. Figure 5.1 compares the run time of each HSP using RSN to the run time using DLI. The problems are arranged along the horizontal axis in ascending order according to the run time using DLI. Each plus sign plots the run time using DLI, and each point plots run time using RSN. All reported run times include the time spent solving the subproblems. The vertical axis measures run time on a logarithmic scale. Because of the ordering of the problems on the horizontal axis and the logarithmic scale on the vertical axis, differences in run time on the right side of the graph are larger than differences in run time on the left of the graph. The average run time using DLI was 3.10 seconds per problem, whereas the average run time using RSN is 2.19 seconds per problem. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 5.1

is correct—recording subproblem nogoods significantly improves efficiency over detecting local inconsistencies.

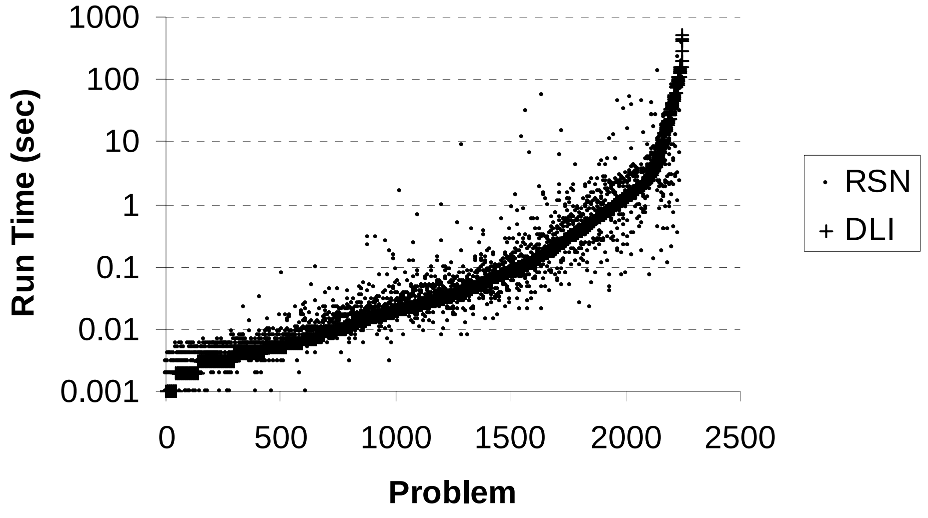

Figure 5.1. Run time of randomly generated HSPs using recording subproblem nogoods and detecting local inconsistencies.

For a particular problem domain, it might be known *a priori* that no HSP contains a local inconsistency. In this situation, DLI can only increase run time, so it would be better to solve each complete HSP directly with the baseline algorithm. On the other hand, if the subproblem nogoods improve efficiency substantially, it might actually be better to spend the time to solve the subproblems first with RSN, as suggested in Hypothesis 5.2:

Hypothesis 5.2. Recording subproblem nogoods will significantly improve efficiency over the baseline algorithm when solving static HSPs.

Figure 5.2 tests Hypothesis 5.2 by comparing the run time of RSN to the run time of the baseline algorithm. Each plus sign plots the run time of the baseline algorithm, each point plots the run time of RSN, and problems are arranged along the horizontal axis in ascending order according to the baseline run time. The average baseline run time was 2.94 seconds per problem, while the average run time of RSN was 2.19 seconds per problem. While this difference is not quite as large as the difference between RSN and DLI, a Student's paired t-test confirms that it is significant with 99.5% confidence. This means that Hypothesis 5.2 is correct—recording subproblem nogoods saves more time while solving the complete problem than it spends while solving the subproblems, and so it improves efficiency over the baseline algorithm to a statistically significant degree.

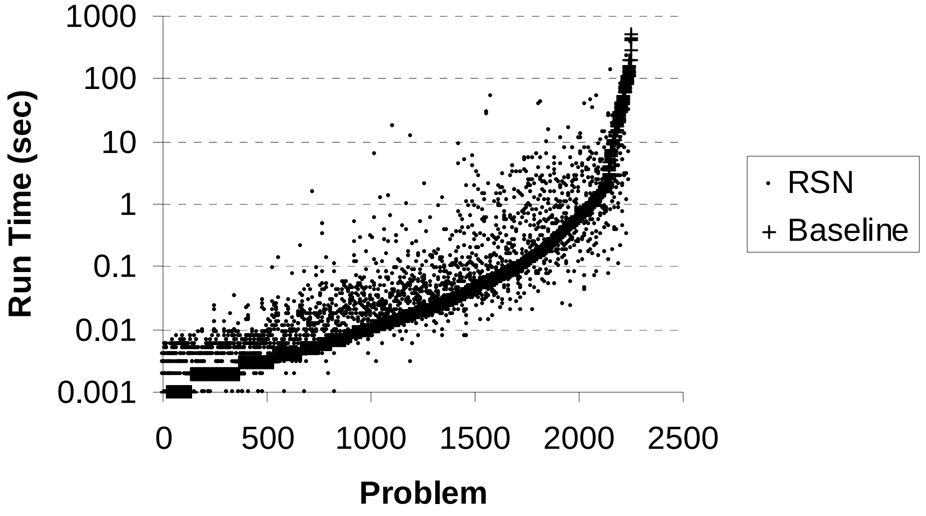

Figure 5.2. Run time of randomly generated HSPs using recording subproblem nogoods and the baseline algorithm.

After solving the subproblems, RSN can then be combined with the idea of focusing search from Section 4.3. I will refer to the combination of recording subproblem nogoods and focusing search on the finite-domain variables as RSN/FS(*F*), and I will refer to the combination of recording subproblem nogoods and focusing search on the temporal variables as RSN/FS(*T*). As pointed out in Section 4.4, the baseline algorithm can be thought of as a degenerate case of a variable subset ordering in which all variables have been placed in a single subset, and so I also refer to the baseline algorithm as FS(*A*). Hence, I will refer to the combination of recording subproblem nogoods and focusing search on the degenerate subset of all variables as RSN/FS(*A*).

By proving Hypothesis 5.2, I have just showed that RSN/FS(*A*) is more efficient than FS(*A*). It is therefore reasonable to expect that RSN/FS(*F*) is more efficient than FS(*F*), and that RSN/FS(*T*) is more efficient than FS(*T*).

Hypothesis 5.3. Recording subproblem nogoods combined with focusing search on finite-domain variables will significantly improve efficiency over focusing search on finite-domain variables alone.

Hypothesis 5.4. Recording subproblem nogoods combined with focusing search on temporal variables will significantly improve efficiency over focusing search on temporal variables alone.

Figures 5.3 and 5.4 test Hypotheses 5.3 and 5.4, respectively, based on the set of randomly generated HSPs. Figure 5.3 compares the run time of RSN/FS(*F*) to the run time of FS(*F*), and Figure 5.3 compares the run time of RSN/FS(*T*) to the run time of FS(*T*). In both figures, each plus sign plots the run time based on focusing search, and each point plots the run time based on recording subproblem nogoods and then focusing search. Problems are arranged along the horizontal axis in ascending order according to the run time based on focusing search without recording subproblem nogoods.

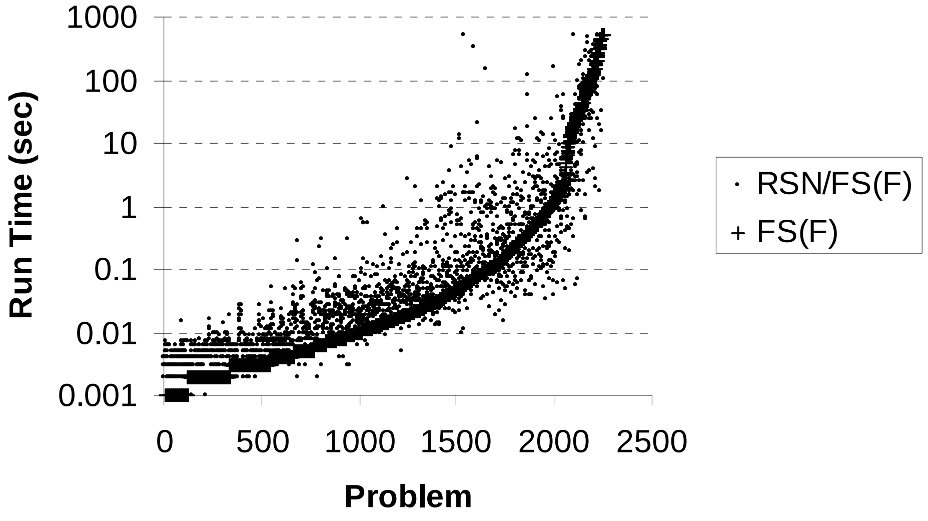

Figure 5.3. Run time of randomly generated HSPs using the combination of recording subproblem nogoods and focusing search on finite-domain variables and focusing search on finite-domain variables alone.

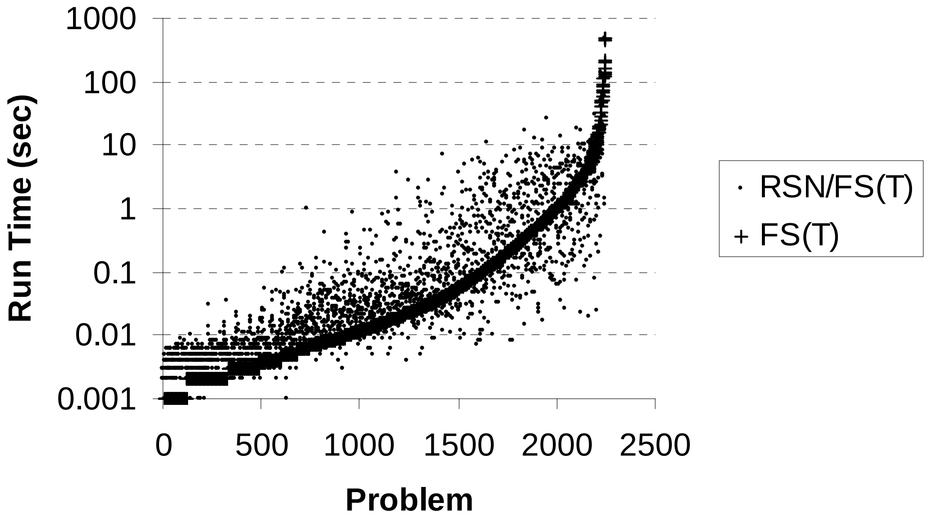

Figure 5.4. Run time of randomly generated HSPs using the combination of recording subproblem nogoods and focusing search on temporal variables and focusing search on temporal variables alone.

Recording subproblem nogoods reduces the average run time of focusing search on finite-domain variables from 9.67 to 8.87 seconds per problem, and it reduces the average run time of focusing search on temporal variables from 2.27 to 1.99 seconds per problem. As expected, Student's paired t-tests confirm with 99.5% confidence that Hypotheses 5.3 and 5.4 are both correct—recording subproblem nogoods before focusing search significantly improves efficiency over focusing search alone because the amount of time it saves while solving the complete HSP is even greater than the time spent while solving the subproblems.

For a clearer picture of the relative effect of recording subproblem nogoods when combined with focusing search, Figure 5.5 graphs the average percent improvement in run time achieved by combining recording subproblem nogoods with focusing search. On average, RSN reduces run time by 25.7% when combined with FS(*A*), 8.2% when combined with FS(*F*), and 12.5% when combined with FS(*T*) for this set of randomly generated HSPs. Even though the standard error is slightly larger than the average in the third column of Figure 5.5, Student's paired t-tests confirm with 99.5% confidence that the effects of RSN are statistically significant in all three cases, as stated earlier.

While the average reduction in run time might not appear to be very dramatic, it is important to remember that recording subproblem nogoods requires solving the subproblems, whereas focusing search does not. It is not surprising that recording subproblem nogoods improves efficiency over detecting local inconsistencies—if no local inconsistency is detected, then the time spent solving the subproblems is wasted, and recording subproblem nogoods can reclaim some of that wasted time. It is less obvious, however, that the information gleaned by recording subproblem nogoods would reduce the time spent solving the complete HSP by even more than the time spent solving the subproblems. The fact that it does means that, even if it is known *a priori* that an HSP contains no local inconsistencies, it is still worthwhile to solve the subproblems in order to record the subproblem nogoods.

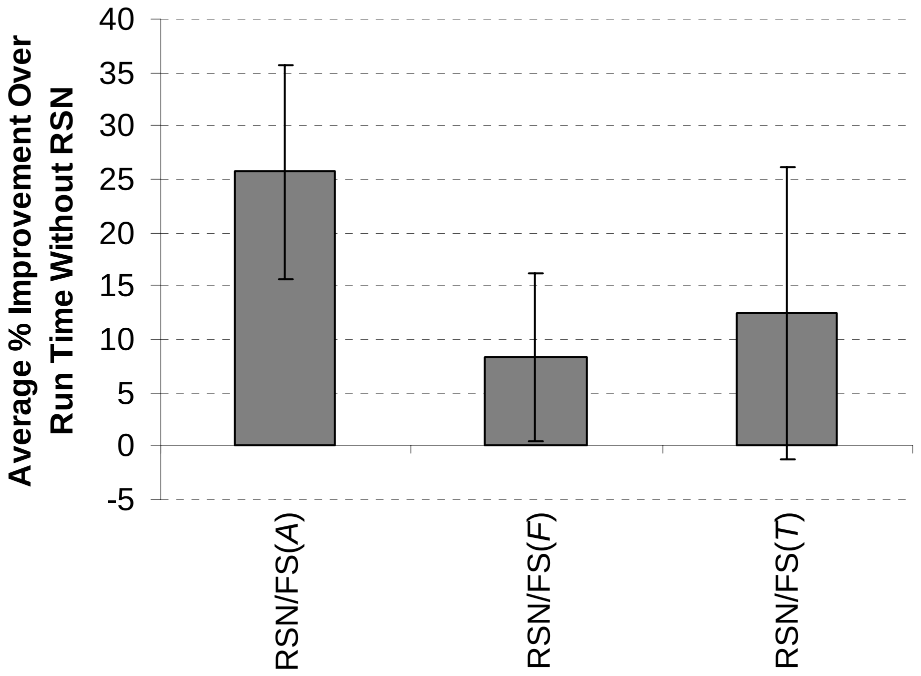

Figure 5.5. Average percent improvement in run time using combinations of focusing search and recording subproblem nogoods over focusing search alone when solving randomly generated HSPs.

Finally, it is possible to combine RSN with choosing the most-constrained subset, as was done in Section 4.4. Recall from Section 4.4 that the problem behind choosing the most-constrained subset is to develop a classifier that can select the most efficient subset ordering for each individual HSP. I made the assumption of a linear relationship between the estimated solution densities of the finite-domain and temporal subproblems of an HSP and the run time of each subset ordering. Based on this assumption, I performed a set of linear regressions to create the LR(ESD, Vars) classifier, which successfully improved efficiency by choosing subset orderings intelligently.

Using this same approach, it should be possible to create a classifier that can improve efficiency when combined with recording subproblem nogoods. The LR(ESD, Vars) classifier is designed to predict run times without RSN, so it will be necessary to create a new classifier to predict run times with RSN. A perfect classifier would choose the best subset ordering every time; I will refer this to this perfect classifier as RSN/Perfect. The effectiveness of the RSN/Perfect classifier can be measured by solving each problem with each subset ordering combined with recording subset nogoods, then choosing the best subset ordering after the fact. The amount by which the RSN/Perfect classifier outperforms the RSN/FS(*A*) algorithm will provide an upper bound on the efficiency gains that are possible using this approach.

Figure 5.6 compares the run time using the RSN/FS(*A*) algorithm against the run time using the RSN/Perfect classifier. Each plus sign plots the run time using the RSN/FS(*A*) algorithm, and each point plots the run time using the RSN/Perfect classifier. Problems are arranged in ascending order according to the run time using the RSN/FS(*A*) algorithm. A Student's paired t-test confirms with 99.5% confidence that the perfect classifier does significantly improve efficiency when combined with recording subproblem nogoods. For this set of randomly generated HSPs, the RSN/FS(*A*) algorithm averaged 2.19 seconds per problem, while the RSN/Perfect classifier averaged 0.98 seconds per problem, which is 44.8% of the RSN/FS(*A*) average run time.

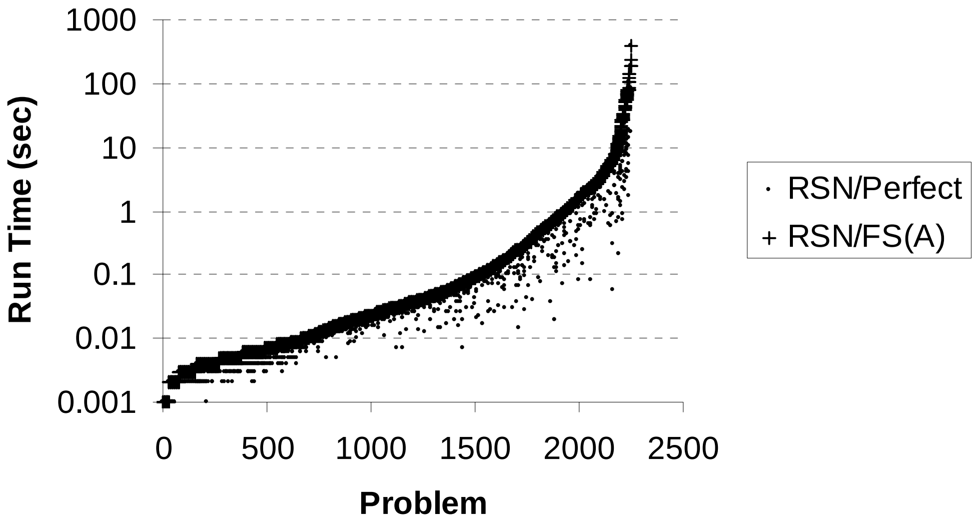

Figure 5.6. Run time of randomly generated HSPs using recording subproblem nogoods alone and recording subproblem nogoods combined with a perfect classifier.

To create an actual classifier that incorporates recording subproblem nogoods, I performed a set of linear regressions using estimated solution densities and the number of variables in the finite-domain and temporal subproblems as inputs (as was done with the LR(ESD, Vars) classifier), and the run time using RSN/FS(*A*), RSN/FS(*F*), and RSN/FS(*T*) as outputs. Because this algorithm combines RSN with the LR(ESD, Vars) classifier, I will refer to this as the RSN/LR(ESD, Vars) classifier. Formula (5.1) shows how the predicted run time of recording subproblem nogoods and focusing search on the variables of subset *X*, PRT(RSN/FS(*X*)), is calculated. As before, Vars(*X*) refers to the number of variables in *X*. Table 5.1 lists the coefficients that result from the linear regressions. The run time using RSN and focusing search on a particular subset can be calculated by inserting the constants from Table 5.1 into Formula (5.1).

$$\mathrm{PRT}(\mathrm{RSN/FS}(X)) = a \log(\mathrm{ESD}(F)) + b \log(\mathrm{ESD}(T)) + c\, \mathrm{Vars}(F) + d\, \mathrm{Vars}(T) + e \qquad (5.1)$$

	a **log(ESD(*F*))**	***b*** **log(ESD(*T*))**	***c*** **Vars(*F*)**	***d*** **Vars(*T*)**	***e*** **1**
PRT(RSN/FS(*F*))	3.529	-3.038	1.256	-0.753	-26.905
PRT(RSN/FS(*T*))	-0.220	-0.203	0.037	-0.035	-6.725
PRT(RSN/FS(*A*))	0.057	-0.802	0.066	-0.202	-4.563

Table 5.1. Coefficients for calculating predicted run time with recording subproblem nogoods as derived from linear regressions based on estimated solution densities and number of finite-domain and temporal variables.

Given these coefficients, I hypothesize that it should be possible to improve efficiency by recording subproblem nogoods and choosing subset orderings:

Hypothesis 5.5. Choosing subset orderings with the RSN/LR(ESD, Vars) classifier will significantly improve efficiency compared to just recording subproblem nogoods with the RSN/FS(*A*) algorithm.

Figure 5.7 compares the run time using the RSN/LR(ESD, Vars) classifier to the run time using the RSN/FS(*A*) algorithm. Each plus sign plots the run time using RSN/FS(*A*), each point plots the run time using RSN/LR(ESD, Vars), and the problems are arranged in ascending order according to RSN/FS(*A*) run time. The average run time using RSN/FS(*A*) was 2.19 seconds per problem, while the average run time using RSN/LR(ESD, Vars) was 1.53 seconds per problem. Comparing the efficiency improvement using the RSN/LR(ESD, Vars) classifier to the efficiency improvement of the perfect classifier shows that RSN/LR(ESD, Vars) achieves 54.5% of the possible efficiency gains using this approach. A Student's paired t-test confirms with 99.5% confidence that Hypothesis 5.5 is correct—when combined with recording subproblem nogoods, a classifier based on estimated solution densities and number of variables significantly improves efficiency over the baseline algorithm.

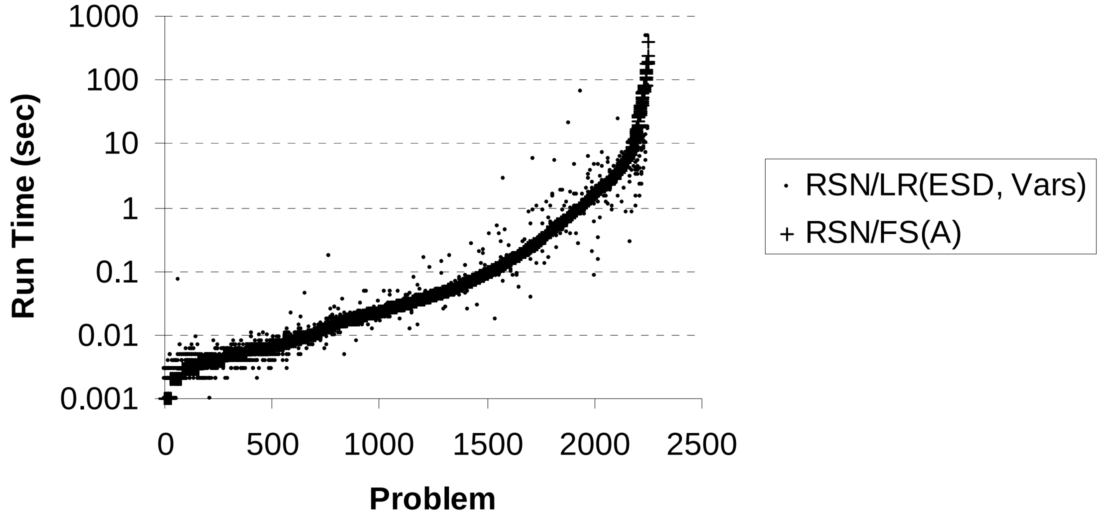

Figure 5.7. Run time of randomly generated HSPs using recording subproblem nogoods alone and recording subproblem nogoods combined with the LR(ESD, Vars) classifier.

Hypothesis 5.5 compared the performance of recording subproblem nogoods with and without a classifier. Similarly, it makes sense to compare the performance of a classifier with and without recording subproblem nogoods. Because I have already shown that recording subproblem nogoods improves efficiency when combined with any other hybrid technique, it stands to reason that it should also improve efficiency when combined with a classifier:

Hypothesis 5.6. The RSN/LR(ESD, Vars) classifier will significantly improve efficiency compared to using the LR(ESD, Vars) classifier.

Figure 5.8 compares the run time using RSN/LR(ESD, Vars) against the run time using LR(ESD, Vars) when solving randomly generated HSPs. Each plus sign plots the run time using LR(ESD, Vars), each point plots the run time using RSN/LR(ESD, Vars), and the problems are arranged in ascending order according to LR(ESD, Vars) run time. The average run time using LR(ESD, Vars) is 1.98 seconds per problem. By incorporating recording subproblem nogoods, the average run time using RSN/LR(ESD, Vars) is 1.53 seconds. A Student's paired t-test confirms that Hypothesis 5.6 is correct for this data set with 99.5% confidence—the efficiency of a classifier based on the estimated solution densities and number of variables of subproblems can be significantly improved by recording subproblem nogoods.

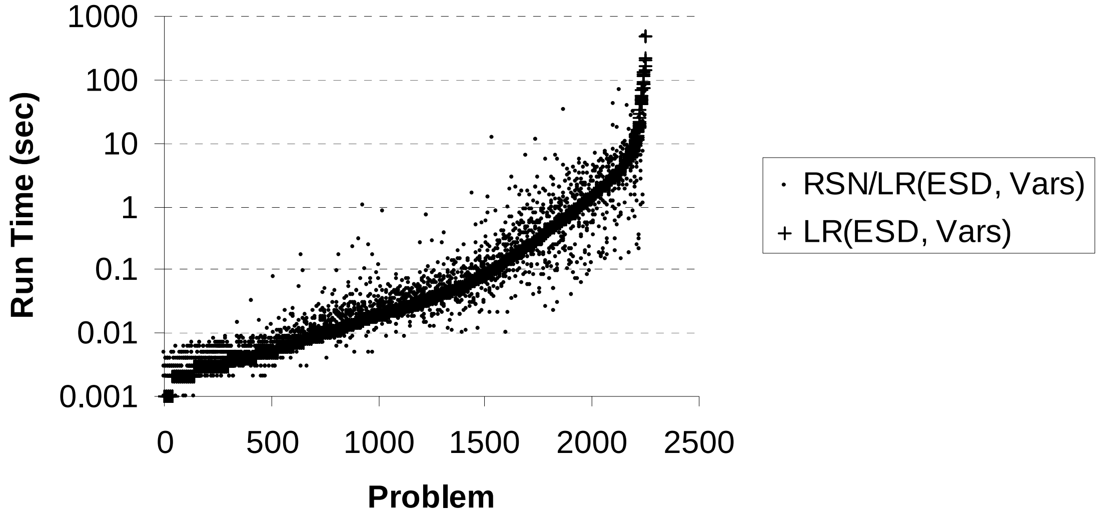

Figure 5.8. Run time of randomly generated HSPs using the LR(ESD, Vars) classifier and the RSN/LR(ESD, Vars) classifier.

Figure 5.9 shows the relative efficiency gains of combining recording subproblem nogoods with classifiers that choose the most-constrained subset. The first two columns show the average percent improvement in run time achieved by combining the perfect classifier and the LR(ESD, Vars) classifier, respectively, with RSN over the run time using RSN alone. For these randomly generated HSPs, the perfect classifier would reduce run time by 55.2% on average over RSN alone, and the LR(ESD, Vars) classifier reduced run time by 30.1% on average over RSN alone. This shows that linear regressions based on estimated solution density and number of variables prove once again to be effective predictors of relative run time, as evidenced by the fact that the RSN/LR(ESD, Vars) classifier improves efficiency by more than half of the theoretical limit. The third column of Figure 5.9 shows that by adding RSN to the LR(ESD, Vars) classifier, run time is reduced by 22.7% on average over the LR(ESD, Vars) classifier alone.

By combining recording subproblem nogoods with hybrid techniques, I have demonstrated that dynamic techniques can improve efficiency when solving static hybrid problems. Specifically, recording subproblem nogoods has three advantages: 1) it can be used in combination with any of the hybrid techniques introduced in Chapter 4, 2) it requires no overhead beyond the time spent solving subproblems, and 3) recording subproblem nogoods reduces the average run time to solve the complete HSP by even more than the time spent solving the subproblems to discover the nogoods.

At the end of Chapter 4, I pointed out that solving subproblems has two purposes—detecting local inconsistencies and estimating solution density. Here, I have demonstrated that solving subproblems can serve a third purpose—recording subproblem nogoods quickly exposes dead end branches that can be avoided later to improve efficiency when solving the complete HSP.

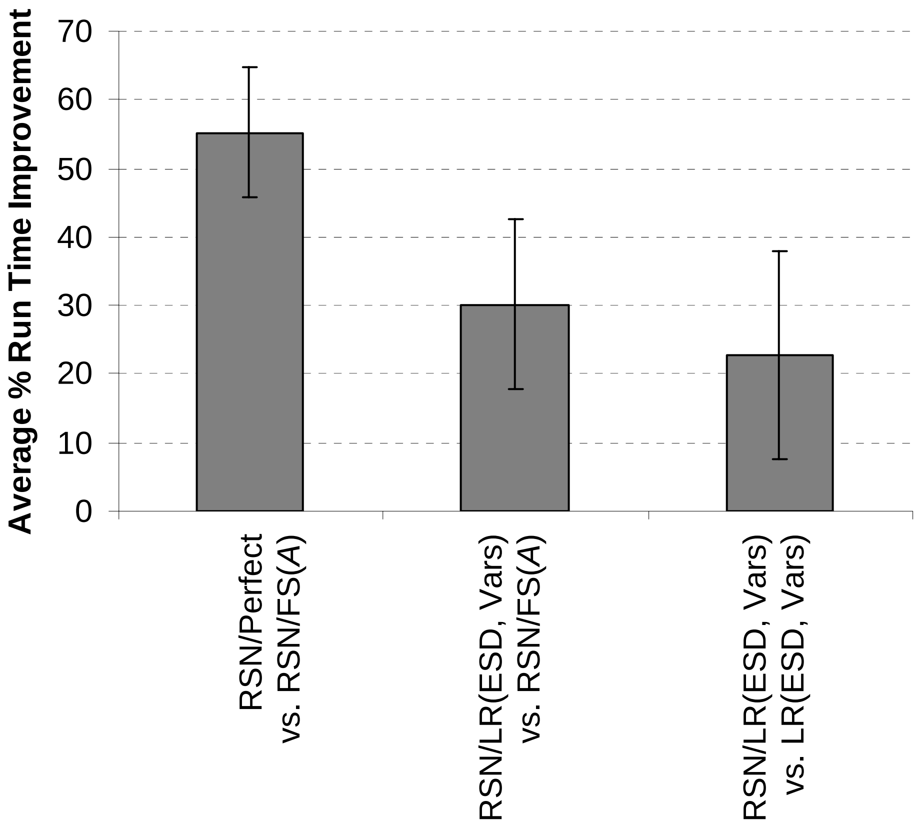

Figure 5.9. Average percent improvement in run time using combinations of choosing the most-constrained subset and recording subproblem nogoods when solving randomly generated HSPs. For techniques *X* and *Y*, column label "*X* vs. *Y*" means that the column height indicates the average percent improvement in run time using technique *X* over using technique *Y*.

5.2.2. Recording Subproblem Oracles

When solving the subproblems of an HSP, in addition to recording subproblem nogoods, it is also possible to record subproblem oracles. Recall that an oracle records the search path to a solution. In a dynamic problem, if an oracle is recorded while solving one problem, then that oracle can be used to guide the search for a solution to the next problem in the sequence. In Chapter 2, I reviewed some previous research that showed that oracles improve efficiency for dynamic finite-domain CSPs, and in Chapter

3, I built on this work to show that oracles can also improve efficiency for dynamic DTPs. In the context of an HSP, we have a finite-domain oracle, which records the solution path of the finite-domain subproblem, and a temporal oracle, which records the solution path of the temporal subproblem.

Given these two subproblem oracles, how can they be combined to guide the search for a solution to the complete HSP? Let O_F and O_T be the finite-domain and temporal oracles, respectively. At any point during the search, let v_F be the next variable according to O_F, and let v_T be the next variable according to O_T. The problem of combining two subproblem oracles is equivalent to the problem of deciding whether to assign v_F or v_T next.

One method for combining the two subproblem oracles is to merge them using the variable ordering heuristic. Any variable ordering heuristic ranks the unassigned variables to help determine which variable should be assigned next. If v_F is the next variable in the finite-domain oracle and v_T is the next variable in the temporal oracle, the variable ordering heuristic can choose the variable with the higher rank. This method does not give any preference to the variables of one oracle over the other, so it is analogous to the FS(A) subset ordering, which does not partition variables according to type.

Another method for combining the two subproblem oracles is to simply append one oracle to the end of the other. By placing one subproblem oracle before the other, all of the variables of one type will be assigned before the variables of the other type, so this method implies focusing search with either FS(F) or FS(T). Without loss of generality, assume the finite-domain oracle precedes the temporal oracle. Assume that all of the finite-domain variables have been assigned values, and that the temporal oracle is now guiding the search. In Chapter 3, I pointed out that when a dead end is encountered while following an oracle, the oracle is discarded and search continues. Assume then that a dead end is encountered while following the temporal oracle, and the search must backtrack all the way into the finite-domain variables. At this point, both of the subproblem oracles have been discarded. If search is focused on the finite-domain variables, the finite-domain variables that were unassigned during backtracking will be assigned next. Once the finite-domain variables have been assigned, the temporal oracle can be reinstituted to guide the search once again. In this manner, the search will attempt to follow the first subproblem oracle only once, but might attempt to follow the second subproblem oracle repeatedly.

After choosing a variable based on one of the subproblem oracles, a value can then be assigned to that variable based on the same oracle. In Chapter 3, I introduced the meta-value oracle, which chooses the same meta-level value for a variable as was chosen in the previous solution, and the temporal bounds oracle, which chooses the meta-level value that minimizes dynamic distance. Choosing a variable and choosing a value are two separate actions, so whether using a meta-value oracle or a temporal bounds oracle, the temporal oracle can be combined with the finite-domain oracle using any of the methods described above.

Recording subproblem oracles can be combined with other hybrid and dynamic techniques. I have already discussed how subproblem oracles relate to focusing search. In the previous section, I showed how recording subproblem nogoods can be combined with choosing the most-constrained subset. Using the same method, a new set of linear regressions should be able to predict the most efficient subset ordering when following subproblem oracles. In Chapter 3, I showed that the combination of oracles and nogood recording was more efficient than either technique alone when solving Dynamic DTPs. Similarly, it should be possible to improve efficiency when solving static HSPs by combining recording subproblem oracles with recording subproblem nogoods. I will leave the empirical comparison of recording subproblem oracles for future work.

5.3. Dynamic Hybrid Scheduling Problems

Many real-world scheduling problems, such as the air traffic control domain, contain both dynamic constraints that change over time and hybrid constraints that define the interaction between temporal and finite-domain variables. Such problems can be represented as a Dynamic Hybrid Scheduling Problem:

Definition 5.1. The *Dynamic Hybrid Scheduling Problem (DHSP)* is a pair $\langle P_0, C\rangle$, where P_0 is a static HSP (the initial HSP of the sequence), and C is a sequence of change sets (defined below). If C contains n change sets, then the solution to the DHSP is a sequence of static HSP solutions $S = s_0, s_1, \ldots, s_n$, where s_i is the solution to problem P_i, and each HSP P_i is created by modifying P_{i-1} according to change set c_i, for $1 \leq i \leq n$.

A *change set* is a finite set of atomic DHSP changes. As with the Dynamic CSP and the Dynamic DTP, all of the changes in a Dynamic HSP can be expressed as changes to the constraints. Any purely finite-domain or purely temporal constraints can be changed as in DCSPs and DDTPs defined earlier. The new types of atomic DHSP changes affect the hybrid constraints. Recall from Definition 4.1 that a hybrid constraint is the disjunction of a finite-domain constraint and a temporal constraint. Any DCSP change can affect the finite-domain disjunct of a hybrid constraint, and any DDTP change can affect the temporal disjunct. Some DHSP changes can actually affect the type of the constraint. For example, removing the finite-domain disjunct from a hybrid constraint transforms that constraint into a purely temporal constraint. Rather than listing out every single type of DCSP and DDTP change again, I will only list the new types of DHSP changes that affect constraint type. As with any dynamic problem, restrictions (1-2 below) can only remove solutions from the solution set, while relaxations (3-4 below) can only add solutions to the solution set.

1) **Remove finite-domain disjunct.** Remove the finite-domain disjunct from a hybrid constraint, transforming it into a purely temporal constraint.
2) **Remove finite-domain disjunct.** Remove the finite-domain disjunct from a hybrid constraint, transforming it into a purely temporal constraint.
3) **Add finite-domain disjunct.** Add a finite-domain disjunct to a purely temporal constraint, transforming it into a hybrid constraint.
4) **Add temporal disjunct.** Add a temporal disjunct to a purely finite-domain constraint, transforming it into a hybrid constraint.

By dividing each HSP of a dynamic sequence into its finite-domain and temporal subproblems, an interesting structure between the subproblems and complete problems in the sequence is exposed. Figure 5.10 shows this structure. This figure depicts an initial

HSP (enclosed in the left light gray block) followed by two modifications (in the center and right light gray blocks) in a dynamic sequence. Each dark gray oval represents a complete HSP, each white circle represents a finite-domain subproblem, and each black circle represents a temporal subproblem. Each solid arrow represents a modification from one problem in the sequence to the next, and each dashed arrow represents a restriction from a subproblem to its complete HSP.

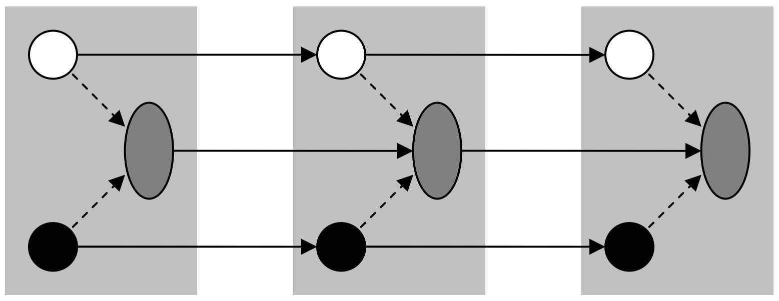

Figure 5.10. Relationships between complete HSPs and subproblems in a DHSP dynamic sequence.

Given the inherent structure of a DHSP and the variety of dynamic and hybrid techniques presented in this dissertation, one could apply combinations of these techniques to improve performance when solving DHSPs. For example, each HSP in the sequence should be relatively similar the one before. If using a classifier based on linear regressions to choose the most-constrained subset, the result of each HSP in the sequence can be incorporated into the linear regressions. Because of the similarity between HSPs in the sequence, the linear regressions should become better at predicting run time in specifically the regions of the problem space in which future problems are likely to appear. Hence, the performance of the classifier should improve over time.

As can be seen in Figure 5.10, the HSPs define one dynamic sequence, and the finite-domain and temporal subproblems define two other dynamic sequences. It is therefore possible to record nogoods, follow oracles, and test justifications for each subproblem independently before solving the complete HSP. As was shown in Chapter 3, these dynamic techniques should improve efficiency and stability when solving the subproblems.

Figure 5.10 also shows another interesting aspect of the structure of a DHSP. By definition, each HSP is a modification of the previous HSP in the sequence. As pointed out above in Section 5.2, each HSP is also a restriction of each of its subproblems. Each HSP is therefore actually a modification of three problems that have been solved previously—its finite-domain subproblem, its temporal subproblem, and the previous HSP in the sequence. Each subproblem oracle that is recorded can be applied to guide search while solving the complete HSP as well as the next subproblem of the same type in the sequence. Greater similarity between consecutive solutions to the subproblems should then translate to greater similarity between consecutive solutions to the complete HSPs. Hence, the effects of hybrid and dynamic techniques could even be enhanced by the structure of a DHSP.

Another important question when considering DHSPs is how to define hybrid stability. A reasonable approach would be to measure hybrid stability in terms of hybrid distance, where the global hybrid distance metric is an aggregation of local distance metrics. Hamming distance has traditionally served as a distance metric for finite-domain assignments, and in Section 3.5.1, I presented several possible distance metrics for temporal assignments. When defining hybrid distance metrics, local distance metrics can measure the distances between finite-domain and temporal assignments independently, but the global distance metric must find a sensible means to combine local distance metrics of different types. A useful first step might be to scale all of the local distance metrics—for example, scale each local distance between 0 (representing no change) and 1 (representing the maximum change possible). After scaling, the global distance metrics defined in Section 3.5.1 could be applicable to a variety of real-world domains. I will leave the further development of hybrid stability metrics to future research.

Chapter 6

Conclusions

6.1. Conclusions

This dissertation extends existing representations of scheduling problems to handle the expressivity and complexity of dynamic and hybrid constraints. The presence of such complex constraints is apparent in many real-world applications, as demonstrated by the air traffic control domain. To manage this complexity, I followed the time-tested approach within Artificial Intelligence of identifying and exploiting structure to improve performance. Based on this approach, I have developed a set of techniques to improve both efficiency and stability when solving complex scheduling problems. I demonstrated the effectiveness of these techniques by testing them on a variety of randomly generated problems as well as problems based on actual air traffic control recordings.

6.2. Contributions

The goal of this dissertation was *not* to find techniques that improve performance only for special cases, subsets of problems, or limited domains. The goal was instead to develop techniques that apply across a wide range of scheduling situations. The main contributions of this dissertation are thus a set of representations and techniques that apply to a broad variety of complex scheduling problems.

Working toward this goal of broadly applicable techniques, I was able to identify assumptions and omissions in the existing literature. I was able to extend existing techniques to handle new situations and develop new techniques to fill in the gaps. Hence, the end result of this work is not a single technique that leads to dramatic speedup for a small set of special cases, but rather a set of techniques that can be combined in many interesting ways to improve performance across the problem space. In the following subsections, I review the major insights, representations, and techniques that I presented throughout this dissertation.

6.2.1. Dynamic Scheduling Problems

To handle scheduling problems with constraints that change over time, I introduced the Dynamic Disjunctive Temporal Problem (DDTP). Just as the Dynamic Constraint Satisfaction Problem (DCSP) is a dynamic sequence of static CSPs, the DDTP is a dynamic sequence of static DTPs. I demonstrated that nogood recording and oracles are effective not only for DCSPs, but also for DDTPs.

When reviewing the existing literature on dynamic constraint satisfaction, I realized that previous researchers had made the assumption that whenever the constraints of a dynamic sequence are tightened to the point of inconsistency, the most recent set of restrictions that caused the inconsistency will be retracted. If other constraints are relaxed instead, the resulting problem may or may not be consistent. In this situation, nogood recording was the only existing technique that could be applied to improve efficiency. Based on the insight that after the relaxation, the variables of the justification are probably still more tightly constrained than the other variables, I developed justification testing to fill the niche left open by previous research. I demonstrated empirically that justification testing is over ten times as effective as nogood recording when applied to randomly generated initially consistent DDTPs.

As I extended the idea of oracles from finite-domain constraint problems to temporal constraint problems, I realized that the basic concept could be interpreted in more than one way. I presented meta-value oracles and temporal bounds oracles, and showed empirically that temporal bounds oracles outperformed meta-value oracles in both efficiency and stability.

Finally, I defined the concept of temporal stability to measure the similarity between subsequent solutions within a DDTP. I presented a family of local and global temporal distance metrics that should apply across a variety of application domains. I described several sophisticated temporal distance metrics from within this family that could be applied to the air traffic control domain. I also pointed out that some temporal distance metrics are monotonically non-decreasing with the depth of the search tree, allowing the use of several well-known search optimization techniques. The combination of nogood recording, two types of oracles, justification testing, and a family of temporal stability metrics provide a means to enhance the performance of a DDTP solver in many different situations.

6.2.2. Hybrid Scheduling Problems

To handle scheduling problems that contain a mixture of finite-domain and temporal variables and constraints, I define the Hybrid Scheduling Problem (HSP). I observed that previous work on hybrid constraint processing ignored the distinction between variable types to simplify the meta-level search algorithm. Rather than ignoring this structure, I exploited it to modify the meta-level variable ordering heuristic and improve efficiency.

First, I recognized that hybrid problems can be readily broken into finite-domain and temporal subproblems. By breaking the problem apart, I was able to organize the problem space based on the relative tightness of the finite-domain and temporal subproblem constraints. This organization made it immediately evident that solving the subproblems before the complete problem could lead to rapid detection of any local inconsistencies, sometimes preventing the need to solve the complete HSP. Furthermore, when both subproblems are consistent, it is unlikely that the finite-domain and temporal variables are constrained equally across all problems within a domain, so I demonstrated that efficiency can be improved by focusing search on one subset of variables before the other. Using this method, I was able to confirm empirically that the air traffic controllers were correct when they told the developers of the CTAS Dynamic Planner that the algorithm should assign the runways before scheduling the arrival times.

Focusing search on one subset of variables across all problems in a domain suffered from two drawbacks—first, it required someone to determine which variable subset is usually the most-constrained, and second, it limited the possible efficiency gains because the search would focus on the less-constrained subset first for some of the problems. To avoid these shortcomings, I extended work from distributed CSPs to develop solution density as a metric of constraint tightness, and I adapted work from estimating tree size to quickly estimate solution density. I then demonstrated that a classifier based on linear regressions of estimated solution density could effectively choose the most-constrained subset and improve efficiency by about half of the theoretical limit. The combination of detecting local inconsistencies, focusing search, and choosing the most-constrained subset can effectively improve efficiency across the problem space.

6.2.3. Dynamic Hybrid Scheduling Problems

By casting hybrid problems as dynamic problems in which the complete problem is a restriction of each of its subproblems, I was able to apply dynamic techniques to improve efficiency when solving static HSPs. I demonstrated empirically that recording subproblem nogoods improves efficiency when combined with any of the other hybrid techniques, and I discussed how subproblem oracles could be recorded and combined to guide the search while solving the complete problem.

In addition, since many real-world applications contain both dynamic and hybrid constraints, I went on to define and conduct an initial set of investigations of the Dynamic Hybrid Scheduling Problem (DHSP). A dynamic sequence of HSP subproblems and complete problems creates an interesting structure, offering new opportunities to improve performance. I concluded with some thoughts on defining hybrid stability metrics that combine previous research on finite-domain stability with my own research on temporal stability. The techniques developed in this dissertation should extend directly to improve efficiency and stability when solving a DHSP, and the structure of a DHSP should offer many new interesting avenues of research.

6.3. Future Work

The ideas presented in this dissertation can be extended in many ways. I will briefly discuss several possible branches of future work in this section.

6.3.1. Extensions of Dynamic Scheduling

In Chapter 3, I developed and tested the concept of justification testing for DDTPs. Justification testing can be used when the previous problem of a dynamic sequence was inconsistent, and the constraints over the variables of the justification have been loosened. The basic insight behind justification testing is that, even after the justification variables have been loosened, they are most likely still constrained more tightly than the other variables in the problem. This basic insight applies to the meta-level variables of the problem, not the timepoints, and so the same idea should be applicable to finite-domain DCSPs. In Section 5.3, I suggested that efficiency could be improved when solving a DHSP by testing justifications for each subproblem independently before solving the complete HSP. The effects of justification testing should be measured empirically in the context of finite-domain DCSPs before being applied to the finite-domain subproblems of a DHSP.

Recent work (Sheini, *et al.* 2005; Moffitt and Pollack 2006) has explored the Disjunctive Temporal Problem with Preferences (DTPP), an extension of the DTP in which a preference function ranks the possible solutions. These works consider various families of preference functions, representations of the problem space, and related algorithms for finding an optimal solution. In Chapter 3, I presented several possible temporal stability metrics for use in a wide variety of application domains. A temporal stability metric can be used to rank the possible solutions of one DTP according to their similarity to the solution of the previous DTP in the sequence. In this capacity, a temporal stability metric can serve as a preference function, mapping solutions with greater similarity to higher preference levels. A deeper examination of the research on DTPPs might inspire new temporal stability metrics and optimization techniques that could be applied to DDTPs.

6.3.2. Extensions of Hybrid Scheduling

In Chapter 4, I presented a set of techniques that exploit the distinction between variable types to improve efficiency when solving HSPs. An HSP is just one example of a problem that contains a mixture of different types of variables and constraints. Recent work in the field of formal verification has explored a family of problems known as Satisfiability Modulo Theory (SMT). Each class of SMT is defined by the types of variables and constraints that can be represented. While many classes of SMT are even more expressive than the HSP, state of the art SMT solvers perform a meta-level transformation followed by a depth-first search, just like the algorithm I tested in Chapter 4. It therefore stands to reason that the same techniques that effectively improved my HSP solver should also effectively improve an SMT solver.

While an HSP has only two types of variables, an SMT class could have more. This presents an interesting question: If the subproblems are solved before the complete problem, as was done when solving HSPs, how should the subproblems then be combined to maximize efficiency? On one hand, increasing the number of combinations of variable subsets that are solved before the complete problem adds to the run time, but on the other hand, it also increases the odds that a local inconsistency will be detected and the number of nogoods that can be recorded and applied to other combinations.

Assume that the variables have been partitioned into four subsets, *A*, *B*, *C*, and *D*, and that each subproblem defined by these subsets has already been solved. Here are four options for combining these subproblems:

1) **Immediate combination.** After solving the subproblems independently, immediately combine all of the variable subsets and solve the complete problem. This option does not spend much time solving subproblems, but might miss some local inconsistencies.
2) **Linear combination.** Combine two of the subsets to solve the problem defined by $A \cup B$, and then add one more subset at a time, solving $A \cup B \cup C$, and finally $A \cup B \cup C \cup D$. This option spends more time solving subproblems, but is more likely to detect a local inconsistency along the way. It will be necessary to decide the order in which the subsets are combined.
3) **Recursive combination.** Combine the subsets into pairs to solve $A \cup B$ and $C \cup D$, and then combine the pairs recursively to solve $A \cup B \cup C \cup D$. With a larger number of subsets, this option will solve more subproblems than the previous option, but these subproblems will be smaller on average. As before, it will be necessary to decide the order in which the subsets are combined.
4) **Total combination.** Combine the subsets into all six possible pairs, then all four possible sets of three, and then finally the complete problem with all four variable subsets. This option maximizes the likelihood of detecting a local inconsistency

at the expense of solving every possible subproblem before moving on to the complete problem. Many of the subproblems will contain some of the same variable subsets, so recording subproblem nogoods and subproblem oracles could mitigate the costs.

6.3.3. Extensions of Dynamic Hybrid Scheduling

I demonstrated in Chapter 5 that keeping subproblem nogoods can improve efficiency when combined with other techniques to solve a static HSP. I went on to discuss some ideas about recording subproblem oracles and several methods for combining them to guide the search when solving the complete problem. Expanding, implementing, and testing these methods of recording subproblem nogoods should be one of the first extensions of this dissertation.

Chapter 5 also combined the expressive power of the DDTP and the HSP into the single framework of the DHSP. Any of the techniques described in this dissertation can be applied when solving a DHSP, and should be tested in this context. I briefly discussed some ideas for combining measures of finite-domain stability with measures of temporal stability into a single measure of hybrid stability. These ideas should be formalized and fleshed out to define useful hybrid stability metrics that can be applied to real-world problem domains. The DHSP is a highly complex, yet highly structured representation, so it is a prime candidate for the common Artificial Intelligence approach of exploiting structure to improve performance.

After defining the DHSP, I pointed out that a classifier used to choose the most-constrained subset of each HSP in the sequence might be able to improve its performance over time by recalculating the linear regressions using the newly observed estimated solution densities and run times as an additional input/output pairs to the linear regressions. The reasoning was that each HSP in the sequence should be similar to those that came before, so the additional information given to the linear regressions should pertain specifically to the region of the problem space in which the next HSP in the sequence should fall. To test this theory, one could compare the performance of a baseline classifier, which predicts run times based solely on an initial set of input/output pairs, versus a learning classifier that continues to update its linear regressions as it solves more HSPs in the dynamic sequence.

Designing an effective learning classifier presents several interesting challenges. First, recalculating the linear regressions will add to the run time, so it might be more effective to wait until some fixed number of HSPs has been solved before updating, rather than updating after each individual HSP. Second, the time spent recalculating will increase with the number of observations, so some mechanism for discarding observations from the training set will be necessary. Since each HSP is similar to its predecessor, the dynamic sequence can be seen as a path that slowly moves through the problem space; hence, learning might be improved by gradually decaying the influence of earlier observations, relying more heavily on the most recent observations to predict run times in the current region of the search space. Finally, as is common with many learning mechanisms, one should consider the tradeoff of exploration versus exploitation—it might be worthwhile to occasionally test a suboptimal subset ordering in order to make more accurate run time predictions in the future.

Bibliography

Armstrong, A., and Durfee, E.H. 1997. Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 620-625.

Baptiste, P., Le Pape, C., and Nuijten, W. 1995. Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling. *First International Workshop on Artificial Intelligence and Operations Research*.

Barták, R. 1999. Constraint Programming: In Pursuit of the Holy Grail. *Proceedings of Week of Doctoral Students*, 555-564.

Bartold, T., and Durfee, E. 2003. Limiting Disruption in Multiagent Replanning. *Second International Joint Conference On Autonomous Agents and Multiagent Systems (AAMAS '03)*.

Bian, F., Burke, E.K., Jain, S., Kendall, G., Koole, G.M., Landa Silva, J.D., Mulder, J., Paelinck, M.C.E., Reeves, C., Rusdi, I., and Suleman, M.O. 2005. Measuring the Robustness of Airline Fleet Schedules. In: Kendall G., Burke E., Petrovic S., Gendreau M. (eds.), Multidisciplinary Scheduling: Theory and Applications, Selected Papers from the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2003), 381-392, Kluwer Academic Publishers.

Bockmayr, A., and Kasper, T. 1998. Branch-and-Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. *INFORMS Journal on Computing*, 10(3):287-300.

Clement, B., and Durfee, E. 1998. Scheduling High-Level Tasks among Cooperative Agents. *Proceedings of the Third International Conference on Multi-Agent Systems*, pp. 96-103, 1998.

Cooper, M.C. 1990. An Optimal k-consistency Algorithm. *Artificial Intelligence*, 41(1):89-95.

Cormen, T., Leiserson, C., and Rivest, R. 1990. Introduction to Algorithms. Cambridge, Mass.: MIT Press.

Dago, P., and Verfaillie, G. 1996. Nogood Recording for Valued Constraint Satisfaction Problems. *Proceedings of the Eighth IEEE International Conference on Tools with Artificial Intelligence*, 132-139.

Davin, J., and Modi, P.J. 2006. Hierarchical Variable Orderings for Multiagent Agreement Problems. *Proceedings of the Seventh Workshop on Distributed Constraint Reasoning*.

Dechter, R. 2003. Constraint Processing. San Francisco: Morgan Kaufmann Publishers.

Dechter, R., and Dechter, A. 1988. Belief Maintenance in Dynamic Constraint Networks. *Proceedings of AAAI-88*, 37-42.

Dechter, R., Meiri, I., and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49:61-95.

Devore, J.L. 1995. Probability and Statistics for Engineering and the Sciences. Pacific Grove: Brooks/Cole Publishing Company.

Dutertre, B., and de Moura, L. 2006. A Fast Linear-Arithmetic Solver for DPLL(T)*. *Proceedings of Computer-Aided Verification (CAV'06).*

Een, N., and Sörensson, N. 2003. An Extensible SAT-Solver. *Proceedings of SAT'03.*

Fox, M., Gerevini, A., Long, D., and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. *In Proceedings of the International Conference on AI Planning and Scheduling (ICAPS-06).*

Fox, M., Howey, R., and Long, D. 2006. Exploration of the Robustness of Plans. *In Proceedings of the National Conference on AI (AAAI-06).*

Gallagher, A.T., Zimmerman, T.L., and Smith, S. 2006. Incremental Scheduling to Maximize Quality in a Dynamic Environment. *Proceedings of ICAPS 2006.*

Gent, I., MacIntyre, E., Prosser, P., Smith, B., and Walsh, T. 1996. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. *Principles and Practice of Constraint Programming (CP'96)*, 179-193.

Ginsberg, M.L. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence*, 1:25-46.

Gomes, C. 2001. On the Intersection of AI and OR. *Journal of Knowledge Engineering Review*, 16(1):1-4.

Gomes, C., Sabharwal, A., and Selman, B. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. *Proceedings of the 21st National Conference on Artificial Intelligence*, 54-61.

Hebrard, E., Hnich, B., O'Sullivan, B., and Walsh, T. 2005. Finding Diverse and Similar Solutions in Constraint Programming. *Proceedings of AAAI'05*, 372-377.

Kask, K., Dechter, R., and Gogate, V. 2004. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. *Constraint Programming 2004*.

Kilby, P., Slaney, J., Thiebaux, S., and Walsh, T. 2006. Estimating Search Tree Size. *Proceedings of AAAI-06*, 1014-1019.

Kramer, L., and Smith, S.F. Maximizing Flexibility: A Retraction Heuristic for Over-subscribed Scheduling Problems. *In Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03).*

Kumar, V. 1992. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32-44.

Mackworth, A.K. 1977. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99-118.

Minton, S., Johnston, M.D., Philips, A.B., and Laird, P. 1992. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling. *Artificial Intelligence*, 58:161-205.

Moffitt, M.D., Peintner, B., and Pollack, M.E. 2005. Augmenting Disjunctive Temporal Problems with Finite-Domain Constraints. *Proceedings of the 16th International Conference on Automated Planning and Scheduling*.

Moffitt, M.D., and Pollack, M.E. 2006. Temporal Preference Optimization as Weighted Constraint Satisfaction. *Proceedings of the 21st National Conference on Artificial Intelligence*.

Montanari, U. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(66):95-132.

Morris, P. 1993. The Breakout Method for Escaping from Local Minima. *Proceedings of AAAI-93*, 40-45.

Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. *Proceedings of the 38th Design Automation Conference (DAC'01).*

Muscettola, N., Morris, P., and Tsamardinos, I. 1998. Reformulating Temporal Plans for Efficient Execution. *Proceedings of the 6th Conference on Principles of Knowledge Representation and Reasoning*.

Nieuwenhuis, R., and Oliveras, A. 2005. Decision procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). *12th Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, December 2005, Jamaica.

Petre, E. 1991. Time Based Air Traffic Control in an Extended Terminal Area—A Survey of Such Systems. *Doc. 912009, Eurocontrol*, Brussels, Belgium.

Policella, N., Smith, S.F., Cesta, A., and Oddi, A. 2003. Steps Toward Computing Robust Schedules. *In Proceedings CP-03 Workshop on Online Constraint Solving: Handling Change and Uncertainty*.

Policella, N., Smith, S.F., Cesta, A., and Oddi, A. 2004. Generating Robust Schedules through Temporal Flexibility. *In Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04).*

Prosser, P. 1993. Hybrid Algorithms for Constraint Satisfaction Problems. *Computational Intelligence*, 9(3):268-299.

Roos, N., Ran, Y., and van den Herik, H.J. 2000. Combining Local Search and Constraint Propagation to Find a Minimal Change Solution for a Dynamic CSP. *Proceedings of the 9th International Conference of Artificial Intelligence: Methodology, Systems, and Applications*, 272-282.

Salido, M.A., Giret, A., and Barber, F. 2003. Distributing Constraints by Sampling in Non-Binary CSPs. *IJCAI Workshop on Distributed Constraint Reasoning*, 79-87.

Schiex, T., and Verfaillie, G. 1993. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools* 3(2):187-207.

Selman, B., Levesque, H., and Mitchel, D.G. 1992. GSAT: A New Method for Solving Hard Satisfiability Problems. *Proceedings of AAAI-92*.

Sheini, H., Peintner, B., Sakallah, K., and Pollack, M.E. 2005. On Solving Soft Temporal Constraints using SAT Techniques. *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*.

Sheini, H., and Sakallah, K. 2005. A SAT-based Decision Procedure for Mixed Logical/Integer Linear Problems. *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*.

Sokkappa, B.G., and Steinbacher, J.G. 1977. Performance of En Route Metering Methods. *MITRE Technical Report MTR-7575*.

Srivastava, B., Kambhampati, S., Nguyen, T., Do, M., Gerevini, A., and Serina, I. 2007. Domain Independent Approaches for Finding Diverse Plans. *IJCAI 2007*.

Stergiou, K, and Koubarakis, M. 2000. Backtracking Algorithms for Disjunctions of Temporal Constraints. *Artificial Intelligence*, 120:81-117.

Strichman, O., Seshia, S.A., and Bryant, R.E. 2002. Deciding Separation Formulas with SAT. *Proceedings of Computer-Aided Verification (CAV'02).*

Tsamardinos, I., Morris, P., and Muscettola, N. 1998. Fast Transformation of Temporal Plans for Efficient Execution. *Proceedings of the 15th National Conference on Artificial Intelligence*, 254-261.

van Hentenryck, P.V., and Provost, T.L. 1991. Incremental Search in Constraint Logic Programming. *New Generation Computing*, 9:257-275.

Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. *Proceedings of AAAI-94*.

Weld, D.S., Anderson, C.R., and Smith, D.E. 1998. Extending Graphplan to Handle Uncertainty & Sensing Actions. *In Proceedings of AAAI'98*, 897-904.

Wong, G. L. 2000. The Dynamic Planner: The Sequencer, Scheduler, and Runway Allocator for Air Traffic Control Automation. *NASA/TM-2000-209586*.

Zhang, L., Madigan, C., Moskewicz, M., and Malik, S. 2001. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *ICCAD*, 279-285.