Texture Measurements as a Basis for Heuristic Commitment Techniques in Constraint-Directed Scheduling

by

John Christopher Beck

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Texture Measurements as a Basis for Heuristic Commitment Techniques

in Constraint-Directed Scheduling

John Christopher Beck

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

1999

The central thesis of this dissertation is that an understanding of the structure of a problem leads to high-quality heuristic problem solving performance in constraint-directed scheduling. Our methods for gaining an understanding of problem structure focus on *texture measurements*: algorithms that implement dynamic analyses of each search state. Texture measurements distill structural information from the constraint graph representation of a search state which is then used as a basis for heuristic decision-making.

To enable the rigorous empirical investigation of the above thesis in the context of real-world scheduling problems, we define the ODO framework for constraint-directed search. The framework allows us to implement scheduling algorithms and their components, and to compare them both from the perspective of static similarities and on the basis of empirical performance.

In this dissertation, we extend the scope of scheduling problems that can be addressed, in general, by constraint-directed techniques. Specifically, we develop the concept of a generalized measure of constraint criticality that enables the construction of dynamic, opportunistic heuristic commitment techniques. Based on an analysis of the requirements of a measure of constraint criticality, we suggest the probability of breakage of a constraint as such a measure.

The investigation of our thesis focuses on three classes of scheduling problems: job shop scheduling, scheduling with inventory, and scheduling with alternative activities. In each of these problem classes we empirically demonstrate that, as a problem becomes more complex, knowledge of its structure has a dominant role in guiding heuristic search to a solution.

Acknowledgements

Many people have contributed, both directly and indirectly, to this dissertation, making it better than it otherwise would have been.

Thanks to Mark S. Fox for insight, support, and guidance for the past four and a half years (not to mention the two years before that).

Thanks to my committee members Mike Carter, Victor Lesser, Steve Smith, and John Mylopoulos for their cogent comments and guidance.

Thanks to Andrew Davenport who has been invaluable in many respects and has contributed much to the quality of the research recorded herein.

Thanks to Edward Sitarski and the R&D staff at Numetrix, Ltd. for the opportunity to spend almost three years in the real-world of scheduling and software development. In particular, I wish to thank Edward, Ioan Popescu, Scott Hadley, Drew van Camp, Georgi Grosev, Rob Morenz, and Stew Ballie for numerous discussions, their technical, commercial, and industrial knowledge, and, of course, the beer.

Thanks to my parents, Anne and John, for years of support and love.

Last and most, thanks to Angela, *sine qua non*, for all her love and support (emotional, financial, nutritional, grammatical, editorial, pastoral-comical, historical-pastoral, tragical-historical, tragical-comical-historical-pastoral).

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The central thesis of this dissertation is that an understanding of the structure of a problem leads to high-quality heuristic problem solving performance in constraint-directed scheduling. Exploration of this thesis has a history in the artificial intelligence literature [Simon, 1973; Fox, 1983; Fox et al., 1989; Sadeh, 1991] and this dissertation is a continuation of such investigations. Our methods for gaining an understanding of problem structure focus on *texture measurements*: algorithms that implement dynamic analyses of each search state. Texture measurements distill structural information from the constraint graph which is then used as a basis for heuristic decision making. Empirical results indicate that, under a number of conditions in a variety of types of scheduling problems, superior search performance is achieved from texture measurement-based heuristic commitment techniques over simpler, less knowledge-intensive heuristics.

In particular, in this dissertation:

- We create and investigate a number of new texture measurements and texture measurement-based heuristics. The texture measurements allow us to achieve a deeper understanding of the problem structure resulting in novel knowledge-based heuristic commitment techniques that outperform existing heuristics.
- We expand the scope of problems that are able to be addressed, in general, by constraint-directed scheduling techniques. Specifically, we demonstrate representation and generic solution techniques for scheduling with inventories and scheduling with alternative activities.

1.1 Motivations

This dissertation is motivated by the need to extend constraint-directed techniques to a wider range of real-world scheduling problems than are currently addressed. The nature of these extensions, including our hypotheses about how to address constraint-directed scheduling problems, closely mirror the motivations for the larger ODO project within which this work has been conducted [Beck et al., 1998]. In more detail, then, the motivations for the work in this dissertation are as follows:

1. **The investigation of texture measurements as a basis for heuristic search** – There are a wide variety of heuristics that can be used as part of a constraint-directed scheduling algorithm. Unfortunately, many of these heuristics are *ad hoc* or specific to the particular class of scheduling problems being addressed. There has been little examination of generalizations of heuristics beyond the conditions of a particular study or of why particular heuristics work in some situations. It has been suggested that a domain-independent basis for heuristic commitment

techniques can be formed from metrics on the constraint graph representation that underlies a search state [Fox et al., 1989]. Such metrics, called *texture measurements* [Fox et al., 1989], have been developed and used as the basis for heuristics commitments in constraint-directed scheduling problems, notably job shop scheduling [Sadeh, 1991; Sadeh and Fox, 1996], distributed scheduling [Sycara et al., 1991; Neiman et al., 1994], and distribution planning [Saks, 1992]. The investigation of texture measurements in this dissertation focuses on the formulation of a firm theoretical basis for texture measurements and uses this basis to extend the types of constraints to which texture measurements can be applied.

2. **The application of constraint-directed scheduling techniques to real-world problems** – One of the original premises for applying constraint-directed search techniques to scheduling was the realization that a scheduling problem is not simply the meeting of due dates [Fox, 1983]. Rather, scheduling is the satisfaction of a plethora of constraints and objectives from many parts of the organization, be it a manufacturing organization scheduling production, a transportation organization scheduling deliveries, or a rostering organization (*e.g.*, schools, hospitals) scheduling people, rooms, and equipment. Much of the scheduling research, however, has concentrated not on the wide variety of constraints, but rather on narrowly defined optimization criteria with questionable real-world relevance (*e.g.*, minimization of makespan). The primary goal of this dissertation, therefore, is to address some of the real-world constraints that have been neglected and investigate the extension of constraint-directed scheduling techniques to problems that incorporate such constraints. There are a number of scheduling systems in use within organizations that deal with a wide variety of real world constraints. However, these systems and the approaches they incorporate have yet to be analyzed in a systematic fashion. Clearly, then yet another *ad hoc* attempt to deal with such constraints will not advance the state of the research. So while we address such new constraints, we do so in a principled fashion: developing the representation to allow modeling of scheduling problems with inventory constraints and alternative activities, as well as extending and comparing solution techniques used in constraint-directed scheduling.
3. **The need for rigorous empirical comparison of scheduling techniques** – Since the inception of constraint-directed scheduling, many different constraint-based scheduling systems have been developed and successfully applied. However, most published results have reported the performance of a particular system on one or two problem sets. There has been little comparative analysis of why or when one approach may be better than another (although there has been more work on this recently, *e.g.*, [Le Pape and Baptiste, 1997]). In addition, there has been a tendency to view a scheduling algorithm as a monolithic whole rather than as a system of interacting components that may be individually investigated. Empirical comparison of components of a scheduling algorithm is critical for a deeper understanding of search behavior [Beck et al., 1997a]. One of the motivations for the work in this dissertation, therefore, is the development of a framework that can be used to model a wide variety of constraint-directed scheduling techniques and algorithms. This framework, furthermore, should not just be a way of statically comparing scheduling algorithms, but should also form an implementational basis of a scheduling shell. Such a shell would allow the implementation of a wide variety of scheduling techniques, and allow them to be rigorously and empirically tested on the same problem sets with only the components of interest varying among the algorithms.
4. **The investigation of the use of commitments as a unification of constraint-directed search** – A number of styles of search have been developed and applied to constraint-directed scheduling such as constructive search (*e.g.*, MicroBoss [Sadeh, 1991; Sadeh and Fox, 1996], PCP [Smith and Cheng, 1993; Cheng and Smith, 1997], CORTES [Sadeh and Fox, 1989]) and iterative repair (*e.g.*, SPIKE [Johnston, 1990; Johnston and Minton, 1994], GERRY [Zweben

et al., 1994; Zweben et al., 1993]). Superficially, these search styles appear quite different; however, underlying the differences are a number of fundamental concepts. Chief among these concepts is the mechanism of modification of the constraint graph by the addition and removal of commitments. A motivation for this dissertation, therefore, is the use of the commitment model as a unifying basis for disparate constraint-directed search techniques. Indeed, it is critical to the formulation of the constraint-directed scheduling framework noted above that a generic model of commitments is adopted.

1.2 Overview of Dissertation

The chapters in this dissertation address the motivations to varying degrees. In Part 1 of the dissertation (Chapters 2, 3, and 4), we primarily address the last two motivations (the rigorous empirical comparison of scheduling techniques and the use of commitments as a unifying concept in constraint-directed search) while building a foundation that will enable us to address our first two motivations (the investigation of texture measurements and the extension of constraint-directed scheduling techniques to characteristics of real-world scheduling problems).

Chapter 2 presents an introduction to constraint-directed search and scheduling, and a review of the literature of constraint-directed scheduling. We focus on three classes of techniques for scheduling and show how a wide range of scheduling research can be understood from the perspective of these three classes.

Chapter 3 presents the ODO Framework for constraint-directed search and scheduling. Given the general applicability of the three classes of scheduling techniques identified in Chapter 2, we present the framework within which we conceptualize and implement our constraint-directed scheduling algorithms.

Chapter 4, the final chapter in the first part of the dissertation, presents the empirical comparison of existing constraint-directed scheduling techniques for job shop scheduling. We examine claims that have been made in the literature and empirically test these claims with a variety of instantiations of the ODO framework.

Part 2 of this dissertation (Chapters 5, 6, and 7) turns to novel constraint-directed scheduling techniques. In particular, we extend existing texture-based scheduling heuristics and create a number of new propagation techniques. These techniques improve the performance of constraint-directed scheduling techniques when applied to standard models of scheduling problems, and allow the application of constraint-directed scheduling techniques to new problem domains. The good performance of the novel texture-based heuristics strongly supports our central thesis.

In Chapter 5, we address the foundation of texture measurements and introduce the notion of the probability of breakage of a constraint as an extension of an existing texture measurement. We create three estimations of the probability of breakage of a constraint, two of which are fully generalizable beyond the constraints typically found in job shop scheduling. In order to evaluate these new texture measurements, we first focus on job shop and show that heuristics based on the generalizable texture measurements are competitive with existing heuristic commitment techniques.

Chapter 6 is our first direct application of constraint-directed scheduling techniques to a characteristic of real-world problems. We present a model of scheduling with inventory and show how one of the extensible texture measurements presented in Chapter 5 is used to measure the probability of breakage of the inventory constraints. We create a heuristic commitment technique based on the texture measurements and introduce a number of new propagation techniques for inventory

constraints. Empirical results indicate that new propagators are critical for successful scheduling with inventories while the texture-based heuristics outperform other heuristics.

In Chapter 7, we turn to scheduling with alternative activities. By adding the explicit representation of the probability that an activity will exist in a final schedule and by further extending texture measurements, we show significant performance gains over other techniques on problems where alternative activities are used to represent alternative resources and/or alternative process plans. We also extend two existing propagators to reason about activities that may not exist in a final solution. We show that the modified propagators confer significant performance advantages, with the best performance resulting from the use of the new texture-based heuristics and the modified propagators.

Finally, in Chapter 8, we conclude and suggest a number of areas for future work.

Appendix A contains an index of important terms used in this dissertation while the other appendices contain detailed data from our experiments. Complete data from the experiments performed in this dissertation are contained in Appendix B, Appendix C, and Appendix D.

1.3 Summary of Contributions

The contributions of this dissertation are as follows:

1. An analysis, within the ODO framework, of competing algorithms in order to determine the main factors contributing to their ability to solve scheduling algorithms.
2. The investigation of the hypothesis that as a problem becomes more complex, knowledge of the problem structure has a dominant role in guiding heuristic search to a solution. The correctness of this hypothesis is demonstrated in the context of job shop scheduling, scheduling with inventory, and scheduling with alternative process plans.
3. The widening of the scope of principled constraint-directed scheduling techniques through the introduction of a more general concept of criticality applicable to any type of constraint. The utility of such a measure as a basis for heuristic commitment techniques is demonstrated in the problem classes noted above.
4. The extension the ODO representation of constraint-directed scheduling problems to represent inventory storage, consumption, and production. The estimation techniques for the probability of constraint breakage are extended to inventory constraints and it is demonstrated that heuristic commitment techniques based on such general estimations are able to outperform non-integrated heuristics as well as heuristics that do not directly reason about inventory constraints.
5. The extension of the ODO framework to represent alternative activities. The concept of the probability of existence of an activity is greatly refined and incorporated into measures of resource criticality and techniques for constraint propagation. The extended measures of resource criticality together with the propagation techniques result in significantly better overall search performance.

Chapter 2 Literature Review

In this chapter we present a review of the constraint-directed scheduling literature. In the first section, constraint-directed search and constraint-directed scheduling are introduced. The balance of the chapter examines techniques used in constraint-directed scheduling and examples of each technique that have appeared in the research literature. Finally, we look at techniques for scheduling with inventory and scheduling with activity alternatives. Work on these final two subjects is relevant to Chapter 6 and Chapter 7 respectively.

2.1 An Overview of Constraint-Directed Search and Scheduling

Constraint-directed search (CDS), broadly defined, is an approach to problem solving that explores the problem space under the guidance of the relationships, limitations, and dependencies among problem objects. These relationships, limitations, and dependencies together are known as *constraints*. The approach requires that these constraints are first, represented, and second, represented in such a way that search techniques can make use of them for guidance.

2.1.1 The Constraint Satisfaction Problem

The simplest application of constraint directed search is to the *finite constraint satisfaction search problem* (CSP) [Mackworth, 1977; Tsang, 1993] which can be defined as follows:

Given:

- A set of n variables $Z = \{x_1, \ldots, x_n\}$ with discrete, finite domains $D = \{D_1, \ldots, D_n\}$.
- A set of m constraints $C = \{c_1, \ldots, c_m\}$ which are predicates $c_k(x_i, \ldots, x_j)$ defined on the Cartesian product $D_i \times \ldots \times D_j$. If c_k is TRUE, the valuation of the variables is said to be *consistent* with respect to c_k or, equivalently, c_k is *satisfied*.

Find:

- An assignment of a value to each variable, from its respective domain, such that all constraints are satisfied.

An instance of a CSP (Z, D, C) can be conceptualized as a *constraint graph*, $G = \{V, E\}$. For every variable $v \in Z$ there is a corresponding node $n \in V$. For every set of variables connected by a constraint $c \in C$ there is a corresponding hyper-edge $e \in E$. Other conceptualizations of a CSP exist, including the dual constraint graph and join graph [Dechter et al., 1990].

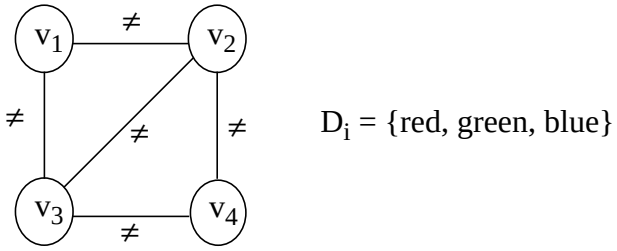


Figure 1. A Small Graph Coloring Problem Represented as a CSP.

A *consistent assignment* or *consistent valuation* of a set of CSP variables, *S*, is the assignment of a value to each variable in *S* such that all constraints in the subgraph induced by variables in *S* are satisfied.

In Figure 1 we present a constraint graph of a CSP modeling a small graph coloring problem. Each variable in the CSP is represented by a node in the graph to be colored. Each variable (node) has a domain of three values {red, green, blue} and each constraint (edge) expresses a "not equals" relationship.

The search for a solution to a CSP can be viewed as a traversal of the problem space consisting of all combinations of variable domain subsets. A solution is a state with a single value remaining in the domain of each variable and no unsatisfied constraints. The mechanism for the traversal of the problem space is the modification of the constraint graph by the addition and removal of constraints. The constraint graph, therefore, is an evolving representation of the search state. In the example of Figure 1, we may (heuristically) add a unary constraint that assigns v_4 = *green*. Alternatively, we may have at our disposal an algorithm that is able to infer that v_1 and v_4 must have the same color, and therefore we introduce a binary "equals" constraint between those two variables. Figure 2 shows a search tree for the graph coloring problem in Figure 1.

CSPs have been successfully used to model a wide range of problems, from the abstract (*e.g.*, graph coloring [Minton et al., 1992]) to the concrete (*e.g.*, design [Navinchandra and Marks, 1987; Navinchandra, 1991]). For excellent reviews of CSP solution techniques and applications, see [Kumar, 1992; Tsang, 1993].

A *constraint optimization problem* (COP) [Tsang, 1993] is defined as a CSP together with an optimization function *f* which maps every tuple to a numerical value: (*Z*, *D*, *C*, *f*) where (*Z*, *D*, *C*) is a CSP, and if *S* is the set of solution tuples of (*Z*, *D*, *C*), then *f*:*S* $\rightarrow$ numerical value. The task in a COP is to find a solution tuple with the optimal (minimal or maximal) value of *f*. A common variation of this model is one in which the optimization function is a weighted sum of the constraints violated by a particular valuation of the variables [Fox, 1983; Smith et al., 1989; Zweben et al., 1993; Zweben et al., 1994]. Rather than satisfy all constraints and optimize *f*, the goal is to minimize the cost by satisfying as many constraints as possible.

2.1.2 Why Constraints?

Constraint-directed search relies on two interdependent intuitions. First, the *representational intuition* is that to solve a problem, the relevant problem information must be represented. Second, the *search intuition* states that to solve a problem, the search should be guided with that represented information. Underlying these intuitions is the *topological assumption* [Fox, 1986]: understand-

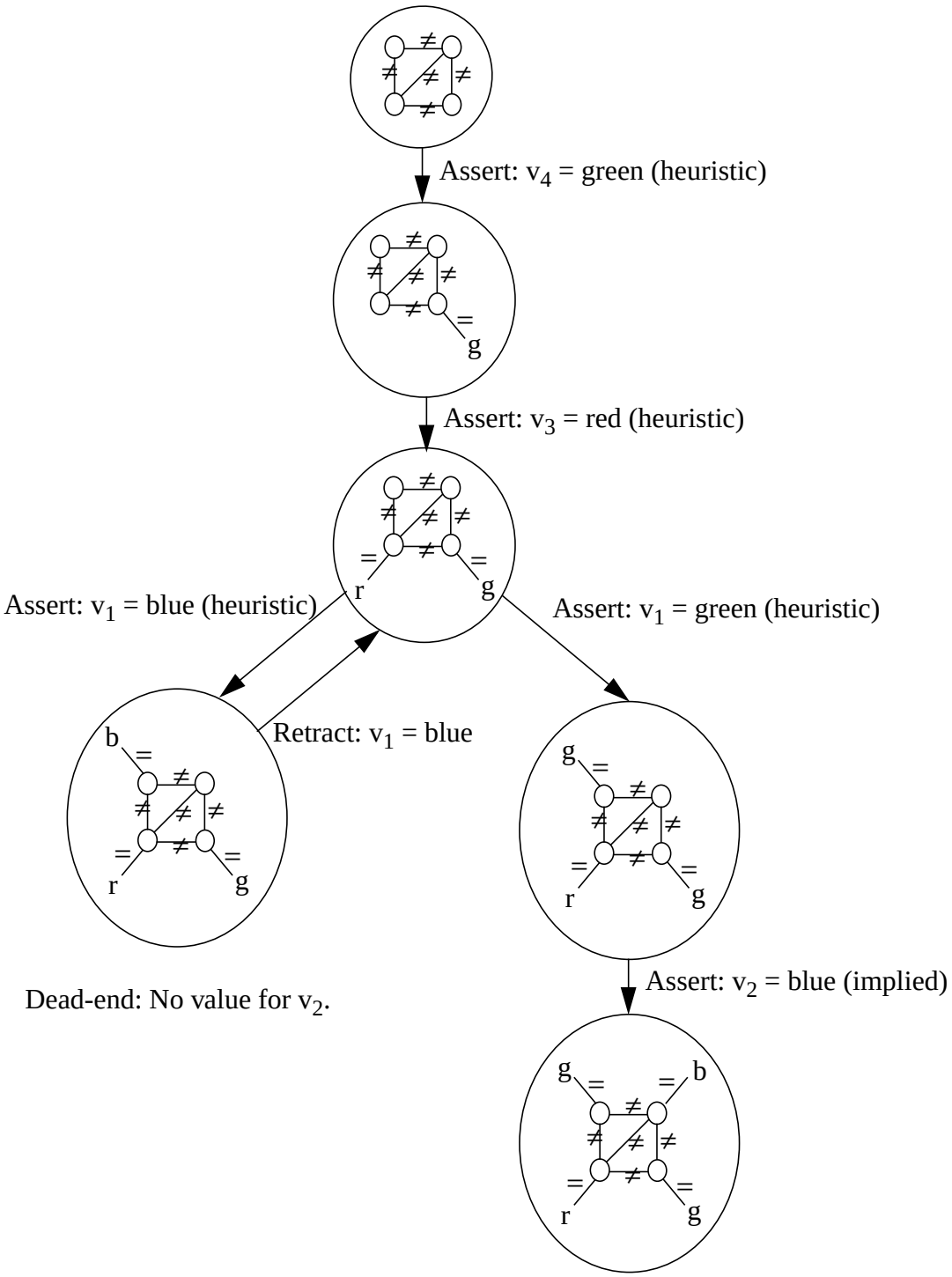


Figure 2. A Possible Search Tree for the Problem in Figure 1.

ing a problem's search space will enable the creation and selection of search techniques that can efficiently navigate the space to a solution.

In the context of CDS, the representational intuition results in the creation of a rich constraint representation that is able to express problem knowledge at a deep level. The search intuition suggests that we look to the constraints for search guidance: constraints are not passive objects that evaluate a potential solution, but rather provide an understanding of the structure of the problem which may be used to guide search in the problem space.

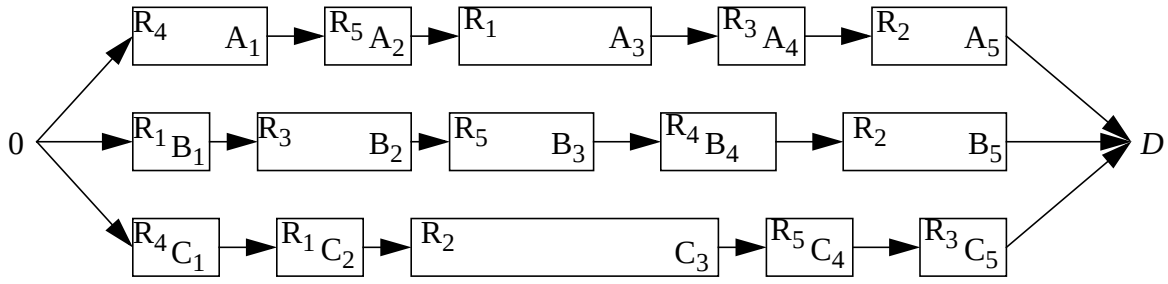


Figure 3. An Example 3×5 Job Shop Scheduling Problem.

2.1.3 Constraint-Directed Scheduling

Constraint-directed scheduling is the representation of a scheduling problem and the search for a solution to it by focusing upon the constraints in the problem. Given that even simple models of scheduling (*e.g.*, job shop scheduling) are NP-hard [Garey and Johnson, 1979], the search process typically depends on heuristic commitments, propagation of the effects of commitments, and the retraction of commitments. In more complex scheduling models, the goal is not simply meeting due dates, but also satisfying many complex (and interacting) constraints from disparate sources within the organization as a whole [Fox, 1983; Fox, 1990]. In short, scheduling is a prime application area for constraint-directed search.

2.1.3.1 The Job Shop Scheduling Problem

One of the simplest models of scheduling widely studied in the literature is the *job shop scheduling problem*. The classical $n \times m$ job-shop scheduling problem is formally defined as follows. Given are a set of n jobs, each composed of m totally ordered activities, and m resources. Each activity A_i requires exclusive use of a single resource R_j for some processing duration dur_i. There are two types of constraints in this problem:

- precedence constraints between two activities in the same job stating that if activity A is before activity B in the total order then activity A must execute before activity B;
- disjunctive resource constraints specifying that no two activities requiring the same resource may execute at the same time.

Jobs have release dates (the time after which the activities in the job may be executed) and due dates (the time by which all activities in the job must finish). In the classical decision problem, the release date of each job is 0, the global due date is D, and the goal is to determine whether there is an assignment of a start time to each activity such that the constraints are satisfied and the maximum finish time of all jobs is less than or equal to D. This problem is NP-complete [Garey and Johnson, 1979]. A recent survey of techniques for solving the job shop scheduling problem can be found in [Blazewicz et al., 1996].

An example of a 3×5 job shop scheduling problem is shown in Figure 3. In this example, there are 3 jobs (A, B, and C) and 5 resources (R_1, …, R_5). Each job has 5 activities (*e.g.*, A_1, …, A_5), a release date of 0 and a due date of D. The resource required by each activity is indicated in the upper-left corner, and the duration, though not specified, is represented by the length of each activity. The arrows represent precedence constraints.

Symbol	Description
ST_i	a CSP variable representing the start time of A_i
STD_i	the discrete domain of possible values for ST_i
est_i	earliest start time of A_i
lst_i	latest start time of A_i
dur_i	duration of A_i
eft_i	earliest finish time of A_i
lft_i	latest finish time of A_i
$lft(S)$	the latest finish time of all activities in S
$est(S)$	the earliest start time of all activities in S
$dur(S)$	the sum of the durations of all activities in S

Table 1. **Notation.**

Many scheduling problems are not simply CSPs, but rather COPs. Relatively simple optimization functions have been studied in the literature such as: the minimization of makespan (*i.e.*, find the schedule with the minimum D) [Applegate and Cook, 1991], minimization of the average (or maximum) tardiness of activities (*i.e.*, how late after their due date activities finish), or some combination of other attributes (*e.g.*, minimize work-in-process combined with tardiness) [Fox, 1983; Smith et al., 1989; Sadeh, 1991]. There has been little work that addresses the many complex and interacting objective functions that typically arise in real-world problems.

2.1.3.2 Notation

For an activity, A_i, and a set of activities, S, we use the notation in Table 1 through the balance of this dissertation. We will omit the subscript unless there is the possibility of ambiguity.

2.1.3.3 Historical Perspective

A number of threads of research have contributed to modern constraint-directed scheduling. It is beyond the scope of this document to discuss the contributions of each thread, much less those of each scheduling system. For our purposes, we note the three chief threads and direct interested readers to [Fox, 1990] and [Le Pape, 1994a] for more in-depth historical perspectives. There has been cross-fertilization among these threads as they have evolved, and some work (*e.g.*, [Carlier and Pinson, 1989; Erschler et al., 1976; Erschler et al., 1980]) spans more than one category. This categorization is not meant to indicate completely independent lines of research, but rather the areas that modern constraint-directed scheduling draws on.

The Knowledge Representation Thread. The original constraint-directed scheduling work is due to Mark Fox and Steve Smith, and their work on the ISIS scheduler [Fox, 1983]. They were the first to adopt constraints as a key knowledge representation (KR) and search guidance tool for both schedule construction and revision. In particular, this thread is responsible for the use of constraints to represent scheduling problems in their full generality, and for the use of the problem knowledge represented in the constraints as the main basis for heuristic decision making. Systems directly descended from ISIS (OPIS [Smith et al., 1989], CORTES [Sadeh and Fox, 1989], MicroBOSS [Sadeh, 1991], and DCHS [Sycara et al., 1991]), and others which have adopted the

constraint-directed philosophy (SONIA [Collinot and Le Pape, 1987], DAS [Burke and Prosser, 1994], GERRY [Zweben et al., 1993; Zweben et al., 1994], MinConflicts [Minton et al., 1992], and DisARM [Neiman et al., 1994]), investigate a wide space of constraint representations and solution techniques.

The Constraint Programming Thread. The constraint programming (CP) community has traditionally stressed representation while using more generic solution techniques: CP languages could typically represent problems far more complex than their solution techniques could handle. The CP thread, developing from Prolog, aimed to provide languages for clear, declarative problem representations, with constraint propagation being dealt with in the underlying language. Attempts to solve hard scheduling problems with these languages were often unsuccessful as the propagation in early versions of constraint programming languages was not sufficiently powerful. More recent work in this field has developed specific propagation techniques for constraints found in scheduling. These recent investigations have corrected the imbalance in solution power and have provided a number of impressive results [Van Hentenryck, 1989; Caseau and Laburthe, 1995; Caseau and Laburthe, 1996; Nuijten, 1994; Le Pape, 1994c; Le Pape and Baptiste, 1996].

The Operations Research Thread. The long(er) history of Operations Research (OR) provides a number of techniques for constraint-directed scheduling. From work which pre-dates constraint-directed scheduling itself [Baker, 1974; Little et al., 1963] to techniques which have been adopted and adapted more recently [Applegate and Cook, 1991; Carlier and Pinson, 1989; Erschler et al., 1976; Erschler et al., 1980], a variety of OR methods continue to have significant impact on both the approaches to and performance of modern constraint-directed scheduling systems. Within the OR community, scheduling techniques have been developed based on mathematical programming techniques (integer programming, column generation) and local search (tabu search [Glover, 1989; Glover, 1990; Nowicki and Smutnicki, 1996], genetic algorithms [Dorndorf and Pesch, 1995], simulated annealing [Laarhoven et al., 1992]).

2.2 Techniques for Constraint-Directed Scheduling

There are three classes of constraint-directed techniques that have been applied to scheduling: heuristic commitment techniques, propagators, and retraction techniques. In the balance of this chapter, we examine the literature on each of these classes.

Throughout the discussion of constraint-directed scheduling techniques we use the term *commitment* to refer to a decision made during search. In Chapter 3, we present a more formal definition of a commitment, but for now it can be conceptualized as a constraint that is added to or removed from the constraint graph. For example, assigning a start time to an activity can be modeled as the assertion of a unary equals commitment on the activity, while sequencing two activities can be represented as the assertion of a binary precedence constraint between the two activities.

2.3 Heuristic Commitment Techniques

A heuristic commitment technique is a procedure that finds new commitments to be added to the graph in order to (heuristically) move toward a solution. In traditional constraint-directed search (and, indeed, artificial intelligence search more generally) heuristic commitments are the primary

component of a search for a solution. A critical component of constraint-directed scheduling techniques, therefore, is the technique used to generate and select the heuristic commitments to make.

In the balance of this section, we will look at a number of heuristics that have been applied to constraint-directed scheduling.

2.3.1 The ORR/FSS Heuristic

The Operation Resource Reliance/Filtered Survivable Schedules (ORR/FSS) heuristic is based on two texture measurements: contention and reliance [Sadeh, 1991]. A *texture measurement* is an analysis of the constraint graph underlying a problem state in order to distill information that can be exploited by the heuristic commitment technique.

Contention. Contention is the extent to which variables linked by a disequality constraint compete for the same value. In the context of job shop scheduling, contention is the extent to which activities compete for the same resource over the same time interval. In ORR/FSS, contention is estimated by finding a probabilistic estimate of an activity's individual demand for the resource and then summing the individual demands for a resource to form an aggregate demand (see example below).

Reliance. Reliance is the extent to which a variable must be assigned to a particular value in order to form an overall solution. In scheduling, one illustration of reliance arises with alternative resources. If activity A_1 requires resources R_1, R_2, R_3, or R_4 and activity A_2 requires resources R_2 or R_5, clearly A_2 has a higher reliance on R_2 than does A_1. If A_1 is not assigned to R_2, it has three other resource alternatives; however A_2 has only one. Reliance can also be formulated in the context of an activity relying on being assigned to a particular start time on a particular resource. In ORR/FSS for job shop scheduling, reliance is a probabilistic estimate of the activity's time preferences.

To calculate contention for a resource, we first determine the individual demand each activity has for that resource. If an activity does not require a resource, it has no demand for it, so its demand is 0. Otherwise, to calculate an activity's individual demand, a uniform probability distribution over the possible start times is assumed: each start time has a probability of 1/|STD|. (Recall that STD is the domain of the activity's start time variable. A uniform probability distribution is the "low knowledge" default. It may be possible to use some local propagation in the constraint graph to find a better estimate of the individual demand [Sadeh, 1991; Muscettola, 1992].) The individual demand, *ID*(A, R, t), is the probabilistic amount of resource R, required by activity A, at time t. It is calculated as follows:

$$ID(A, R, t) = \sum_{t - dur_A < \tau \le t} \sigma_A(\tau) \qquad (1)$$

Where:

$$\sigma_A(\tau) = \begin{cases} \dfrac{1}{|STD_A|} & \tau \in STD_A \\ 0 & otherwise \end{cases} \qquad (2)$$

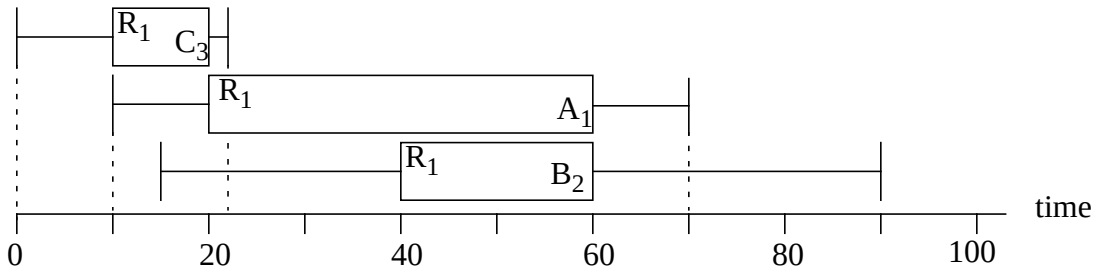


Figure 4. Activities A_1, B_2, and C_3.

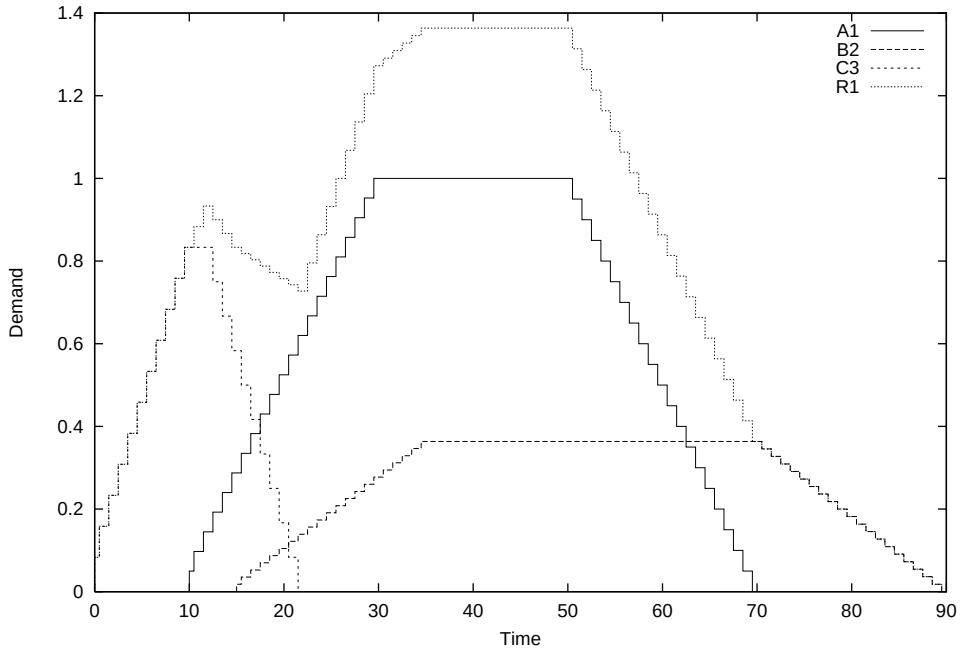


Figure 5. Individual Demand Curves (A_1, B_2, C_3) and Their Aggregate Demand Curve (R_1).

Figure 5 shows both the individual demand curves for the three activities in Figure 4 and the aggregate curve for the resource.

Contention is estimated by aggregate curves found for each resource by summing the individual demand curves. The aggregate demand curves and a time interval equal to the average activity duration are, then, used to identify the {resource, time interval} with the greatest area. By definition, the unassigned activity that contributes the most area to the critical time interval is the most critical activity. This is the activity that the ORR heuristic predicts is important to schedule at this point in the search: it is the most reliant activity on the most contended-for {resource, time interval} pair.

Once the critical activity, *A*, is identified, FSS is used to rate each of its possible start times by using the demand curves. This rating takes into account the effect an assignment to *A* will have both on activities competing directly with A and on those temporally connected to *A*. A more

detailed description of the start time assignment heuristic is beyond the scope of this document; interested readers are referred to [Sadeh, 1991].

Sadeh presents empirical evidence showing that the MicroBOSS scheduler, using ORR/FSS, dominates all of the tested dispatch rules on a set of 60 problems. Subsequent work showing the competitive performance of simpler heuristics (*e.g.*, the CBASlack heuristic (Section 2.4.1) [Smith and Cheng, 1993]) and significantly better performance using edge-finding [Nuijten, 1994] have called into question the efficacy of ORR/FSS. A difficulty with both of these subsequent works is that a number of different techniques were conflated and therefore it is difficult to determine that it is the heuristics rather than, for example, the retraction method, that lead to the poor results [Beck et al., 1997a]. No work of which we are aware has compared only the heuristic commitment techniques while holding all other components constant. In Chapter 4 we will undertake such a comparison.

2.3.2 Task Interval Entropy Heuristic

Caseau and Laburthe use *task intervals* as a basis for a heuristic commitment technique [Caseau and Laburthe, 1995; Caseau and Laburthe, 1996].[1] A task interval is defined as follows:

> If activity A_i and A_j are two activities (possibly the same), that use the same resource and such that $est_i \leq est_j$ and $lft_i \leq lft_j$, then the *task interval* $[A_i, A_j]$ is the set of tasks $\{A_k\}$, that use the same resource as A_i and A_j, and such that $est_i \leq est_k$ and $lft_k \leq lft_j$.

For task interval, *I*, and resource, *R*, the following are defined:

- U_R is the set of all activities using *R*.
- *First*(*I*) = the set of activities, in *I*, that might execute first of the activities in *I*.[2]
- *Last*(*I*) = the set of activities, in *I*, that might execute last of the activities in *I*.

$$IntervalSlack(I) = lft(I) - est(I) - dur(I) \tag{3}$$

$$ResourceSlack(R) = lft(U_R) - est(U_R) - dur(U_R) \tag{4}$$

The Task Interval Entropy (TIE) heuristic identifies the critical resource as the one which minimizes the product in expression (5).

$$IntervalSlack(I^*) \times ResourceSlack(R) \times min(First(I^*), Last(I^*), par) \tag{5}$$

1. Task Intervals were primarily formulated for use with the edge-finding propagators (see Section 2.4.2).
2. The edge-finding not-first/not-last propagator (Section 2.4.3) can be used to efficiently identify these sets of activities.

Where I^* is the task interval of R with the minimum *IntervalSlack* and *par* is a parameter, assigned empirically, to around 3.[3]

I^*_R is then defined to be I^* on the critical resource. TIE chooses the smallest set from First(I^*_R) and Last(I^*_R), and picks two activities, *A* and *B*, to sequence. If $|First(I^*_R)|$ is smaller, *A* is the activity with minimal *est* and *B* is chosen to minimize the sum of the slack resulting from sequencing *A* before *B* and sequencing *B* before *A*. Similarly, if $|Last(I^*_R)|$ is smaller, *B* is the activity with maximal *lft* and *A* is chosen to minimize the sum of the slack resulting from sequencing *A* before *B* and *B* before *A*. In either case, *A* before *B* is posted.

The intuition for choosing the activities that minimize the sum of the slack is that a search state is a branch point and it is desirable to simplify the problem as much as possible on both branches. Minimizing the slack is likely to increase the number of implied commitments that the propagators can find and therefore simplify the problem most.

Experiments are carried out with this heuristic, sophisticated propagators (including edge-finding and some extensions), and chronological backtracking on a set of problems from the Operations Research library [Beasley, 1990]. While there are no direct comparisons to other constraint-directed work (comparisons are done against a number of polyhedral (cutting plane) methods), comparison of reported results with those found in [Vaessens et al., 1994], seems to indicate that TIE is competitive or better than the other approaches.

Interestingly, the heuristic attempts to *minimize* the resulting slack on both branches because then the propagators will infer more constraints. This is in contrast to the CBASlack heuristic (Section 2.4.1) which attempts to *maximize* the slack remaining after a commitment.

2.3.3 Resource Slack, First/Last Heuristic

[Baptiste et al., 1995a] use *ResourceSlack* (equation (4) above) to identify the critical resource as the one with the smallest slack. Based on comparison of the time-windows available for each unsequenced activity on the critical resource, one is selected to execute first (or last). All activities on the critical resource are sequenced before identifying the next critical resource. Three heuristics are examined for activity selection:

1. Select an activity to execute first among the unsequenced activities. Propagators (*e.g.*, edge-finding not-first/not-last, Section 2.4.3) can be used to identify activities that cannot execute first; however, it is unclear precisely which propagators are used. Once the set of activities is identified, one activity is selected to execute first by the EST-LST rule: select the activity with the smallest *est* and break ties using the smallest *lst*.
2. Select an activity to execute last among the unsequenced activities. Analogously, identify the set of activities that can execute last from the set of activities and use a LFT-EFT[4] (select the activity with the largest *lft* and break ties using the largest *eft*) to pick the activity to schedule last.
3. Dynamically, choose to schedule an activity first or last based on the size of the sets of activities that are able to execute first or last. Select the smaller set and use the appropriate rule (EST-LST or LFT-EFT) to select an activity from that set. If the set sizes are equal, the tie is broken by comparing the difference between the earliest start times of the top two activities to sched-

3. Unfortunately, the authors provide no justification or intuition for their use of *par*.

4. [Baptiste et al., 1995a] refer to this rule as the LET-EET rule for “latest end time” and “earliest end time”.

ule first with the difference between the latest finish times of the top two activities to schedule last. If the difference is greater between the activities to execute first, then that set is used; otherwise the set of activities that can execute last is used.

Once the critical resource is identified, all activities on it are scheduled before moving to another resource. This is quite different from most modern techniques which have a more opportunistic nature in that after each commitment the critical resource is recalculated. Such resource-centered approaches have been previously investigated in OPIS [Smith et al., 1989].

Empirical evidence shows little difference among the activity selection heuristics; however, they are not directly compared with any other heuristics. Reported results of these heuristics with the edge-finding propagator and chronological backtracking seem to be a significant improvement over many techniques, though they lag behind results using the TIE heuristic. Again, head-to-head comparison is necessary for firmer conclusions to be drawn.

2.3.4 CBASlack Heuristic

The Constraint-Based Analysis Slack (CBASlack) heuristic forms the heuristic component of the Precedence Constraint Posting (PCP) scheduling algorithm [Smith and Cheng, 1993; Cheng and Smith, 1997].

Bslack (equation (8) below) is calculated for all pairs of activities and the pair with the smallest *Bslack* value is identified as the most critical. Once the critical pair of activities is identified, the sequence that preserves the most slack is the one chosen. The intuition here is that a pair with a smaller *Bslack* is closer to being implied than one with a larger value. Once we have identified this pair, it is important to leave as much temporal slack as possible in sequencing them in order to decrease the likelihood of backtracking.

$$slack(i \rightarrow j) = lft_j - est_i - (dur_i + dur_j) \tag{6}$$

$$S = \frac{min(slack(i \rightarrow j), slack(j \rightarrow i))}{max(slack(i \rightarrow j), slack(j \rightarrow i))} \tag{7}$$

$$Bslack(i \rightarrow j) = \frac{slack(i \rightarrow j)}{\sqrt{S}} \tag{8}$$

The intuition behind the use of $\sqrt{S}$ is that there is search information in both the minimum and maximum slack values. While choosing activity pairs based on minimum slack remains a dominant criteria, the authors hypothesize that more effective search guidance may be achieved with a biasing factor that will tend to increase the criticality of activity pairs with similar minimum and maximum slack values. For example, it is unclear if an activity pair with a minimum slack of 5 and a maximum slack of 5 is more or less critical than a pair with minimum slack 3 and maximum slack 20. *S* (equation (7)) is a measure of the similarity of the minimum and maximum slack values for an activity pair and it is incorporated in the biased-slack calculation as shown in equation (8).

Empirical evidence [Smith and Cheng, 1993] shows that the CBASlack heuristic combined with the CBA propagator (Section 2.4.1) competes very well with ORR/FSS though different types of commitments are made (*i.e.*, precedence constraints rather than start time assignments).

2.3.5 The Randomized Left-Justified Heuristic

In the Randomized Left-Justified Heuristic (LJRand) [Nuijten et al., 1993; Nuijten, 1994] the set of activities that can execute before the minimum earliest finish time of all unscheduled activities is identified. One of the activities is randomly selected and scheduled at its earliest start time.

In the job shop scheduling context, a new commitment can be inferred when a heursitic commitment made by LJRand is undone by chronological backtracking (or another provable retraction technique (see Section 2.5.1.1)). Because a dead-end results when the activity, *A*, is assigned its earliest start time, the earliest start time of *A* can be updated to the smallest earliest finish time of all other unscheduled activities on the same resource as *A*. The provable backtracking has shown that some other activity must come before *A* in the resource in question [Nuijten, 1994].

Experiments were run comparing LJRand (as part of the SOLVE algorithm (Section 3.3.2)) with the ORR/FSS heuristic (as part of the ORR/FSS algorithm (Section 3.3.1)) and with the ORR/FSS augmented with the propagators used in SOLVE. SOLVE strongly outperforms the augmented ORR/FSS, which, in turn, strongly outperforms, unaugmented ORR/FSS.

It is not clear from this study whether the ORR/FSS heuristic or some other component of the ORR/FSS algorithm is to blame for its poor performance. Nonetheless, these results raise questions in regard to sophisticated heuristic techniques like ORR/FSS. We address this issue in Chapter 4.

2.3.6 Local Search Heuristics

Finally, there are a number of heuristic commitment techniques often used in local search algorithms.

The basic MinConflicts algorithm [Minton et al., 1992] begins with a start time assigned to each activity and then identifies the most critical activity as the one which conflicts with the most other activities. For each possible start time of the critical activity, the number of resource violations that would result from reassigning the activity is assessed. The start time with the lowest number of violations is chosen as the new commitment. The heuristic commitment used in GERRY [Zweben et al., 1993; Zweben et al., 1994] (Section 3.3.3) is a variation of MinConflicts. MinConflicts has shown to perform very well on some large constraint satisfaction problems (*e.g.*, *N*-queens with *N*=1000000) and on a set of space shuttle scheduling problems [Zweben et al., 1993; Zweben et al., 1994]. Performance is weaker on other job shop scheduling benchmarks [Davis, 1994].

An in-depth comparison of many of the neighborhood functions that have been applied to scheduling is done in [Vaessens et al., 1994]. Based on a search state where all the activities on a resource are completely sequenced, the neighborhood functions span the following commitments:

- swapping two adjacent activities
- removing an activity from its current position and inserting it before or after another (non-adjacent) activity
- re-ordering all activities on a single machine
- modifying orderings across multiple machines

For our current purposes, the critical point is not what types of commitments can be made (though clearly the type of commitment will profoundly affect the search) but rather the heuristics that filter and/or evaluate neighboring states.

Unfortunately, the empirical comparisons embed the heuristic commitment techniques in different overall strategies (*e.g.*, one heuristic with tabu search (Section 3.3.4) compared to another heuristic with simulated annealing). Where they are able to compare the overall search strategies it is not clear how these results speak to a comparison solely based on the heuristic commitment techniques.

2.3.7 Open Issues

1. Taken together, there are few similarities among the heuristic commitment techniques presented above. We believe that texture measurements can be used to characterize the various heuristic commitment techniques, even though, with the exception of ORR/FSS, they were not originally conceived in this way. This characterization would be an effort toward an underlying, domain-independent, foundation for heuristics used in constraint-directed search. Such a foundation can lead to a deeper understanding of the existing heuristics as well as provide a firm basis for the creation of new heuristic commitment techniques. However, the foundation must be justified both theoretically and empirically. Theoretically, texture measurements require a mathematical basis for both their exact calculation and their estimation. The ability to characterize various estimation techniques based on the expected error from the true measurements is a significant open question. From an empirical perspective, a basic starting point is to demonstrate that a better estimate of texture measurement (while holding the type of commitments made constant) leads to a better heuristic commitment. In Chapter 3, we begin this characterization by demonstrating how a number of these heuristic commitment techniques can be reinterpreted as being based on texture measurements.
2. There is a major need for a head-to-head comparison of the techniques for heuristic commitments. It is not at all clear how to evaluate the techniques described above since empirical studies often embed the techniques in larger search strategies. It would seem straightforward to compare a number of these techniques (ORR/FSS, Task Interval Entropy, Resource Slack First/Last, CBASlack, and Randomized Left-Justified) while holding all other components of the search strategy constant. It is more difficult, however, to compare, say, ORR/FSS meaningfully with the various local search heuristics since their formulations use very different retraction strategies. This comparison is an important issue.[5]
3. Given the wide variety of approaches to heuristic commitment techniques, (*e.g.*, LJRand compared with ORR/FSS) is there a way to identify problems where one is likely to perform better than the other? Can we quickly analyze a problem and discover internal structure indicating that it will be particularly amenable to a simple heuristic in a local search strategy or, conversely, a sophisticated heuristic in a constructive strategy?
4. Propagators vs. Heuristics. It seems possible that if a heuristic technique is good enough, there is no need for propagators: the heuristics will make the same decisions that the propagators will make. If propagators are more expensive than the heuristic technique, a trade-off is suggested between spending effort in finding and making sound decisions as opposed to just making heuristic commitments. Such a trade-off has not been observed.

---

5. Work in the operations research literature [Hooker, 1996] raises this issue, among many others concerning the empirical study of heuristics.

5. The fact that CBASlack maximizes remaining slack, while TIE minimizes it raises an interesting question as to the intuition behind the heuristics. The slack-maximizing algorithm preserves slack in a least-commitment approach: by leaving more slack it is more likely that a solution will be found because there is a higher probability that the to-be-scheduled activities can be scheduled. The slack-minimizing algorithm attempts to minimize the slack resulting from either sequence at a branch point. The intuition is that both branches will terminate more quickly (either in solution or failure). Work on branching rules for satisfiability problems [Hooker and Vinay, 1995] shows that a popular heuristic works not because it increases the probability that the chosen branch contains a solution (as was conjectured), but that it simplified the problem the most (and allowed a propagator to infer more implied commitments). We interpret this work as supporting the slack-minimizing intuition.
6. Few of the heuristic techniques discussed above are directly applicable to more complex constraints outside of the job shop scheduling model. If research is to address real world problems, the more complex constraints need to be addressed.

2.4 Propagators

A propagator is a function that analyzes the current search state to find constraints that are implied by, but are not explicitly present in, the constraint graph. By making these constraints explicit, we can use them to prune the number of possibilities to be explored in the search space. For example, these constraints may be used to reduce the start time domains of some of the activities in the problem. The advantages of propagators stem from the soundness of their commitments (a propagator will never infer a constraint that is not a logical consequence of the current problem state) and the fact that, when a constraint is explicitly present in the graph, it not only reduces the search space, but it is often possible to use additional propagators to further prune the search space.

Examples of propagators from a CSP perspective include the various consistency enforcement algorithms such as arc-consistency and *k*-consistency [Mackworth, 1977; Freuder, 1978; Freuder, 1982]. These algorithms are typically viewed as removing values (or tuples of values) from variable domains; however, we treat them as adding implied constraints that, for example, rule out domain values (*e.g.*, a unary “not-equals” constraint).

Many powerful propagation techniques have been developed for constraint-directed scheduling in recent years, for instance, many variations of edge-finding [Carlier and Pinson, 1989; Caseau and Laburthe, 1994; Nuijten, 1994; Le Pape and Baptiste, 1996] and shaving [Carlier and Pinson, 1994]. It has long been known that search can be drastically reduced by enforcing various degrees of consistency [Freuder, 1982]. The effort to achieve high degrees of consistency, however, appears to be at least as expensive as more traditional algorithms. The goal for propagator research, then, is to find the trade-off between complexity and the resultant easing of the search effort.

In the balance of this section, we examine three propagators used in job shop scheduling.

2.4.1 Constraint-Based Analysis

Constraint-Based Analysis (CBA) [Erschler et al., 1976; Erschler et al., 1980; Smith and Cheng, 1993; Cheng and Smith, 1997] enforces arc-B-consistency on unary resource constraints. Arc-B-consistency [Lhomme, 1993] (where ‘B’ stands for “bounds”) ensures that for the minimum and maximum values of any variable, v_1, there exists at least one consistent assignment for

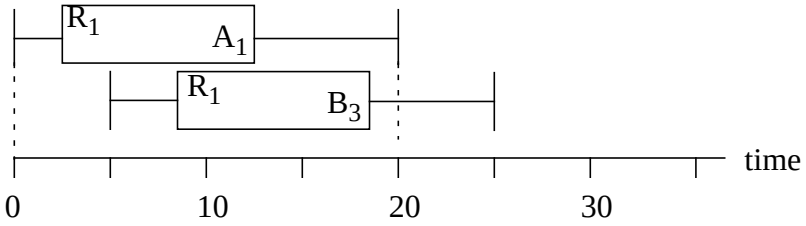


Figure 6. An Example Where CBA Can Infer a New Constraint: A_1 Before B_3.

any other connected variable, v_2 (when considered independently of all other variables). Clearly, arc-B-consistency is limited to variables where there is a total ordering over the values. The start time variables in scheduling meet this requirement.

CBA analyzes the start and end times of all pairs of activities executing on the same unary capacity resource. Given activities, A_i and A_j, that compete for the same resource, CBA identifies the following cases:

1. If $lft_i - est_j < dur_i + dur_j \leq lft_j - est_i$ then A_i must be before A_j.
2. If $dur_i + dur_j > lft_j - est_i$ and $dur_i + dur_j > lft_i - est_j$ then the current state is a dead-end.
3. If $dur_i + dur_j \leq lft_j - est_i$ and $dur_i + dur_j \leq lft_i - est_j$ then either sequence is still possible.

If, after looking at all pairs of activities on each resource, CBA finds that all pairs are in Case 3, it cannot infer any new constraints: all the resource constraints are arc-B-consistent.The worst-case time complexity for CBA is $O(mn^2)$, where n is the number of activities on one resource and m is the number of resources.

In the example shown in Figure 6, CBA is able to infer that A_1 must be before B_3 because the interval between the earliest start time of B_3 and the latest end time of A_1 is 15, which is smaller for the combined durations of the activities which is 20.

Empirical evaluation of CBA with the CBASlack heuristic (Section 2.3.4) has demonstrated good performance on one set of job shop scheduling benchmarks. The experimental design, however, did not evaluate CBA independently of the heuristic commitment technique and so it cannot be judged whether the performance was due to CBA, the heuristic, or the combination.

2.4.2 Edge-finding Exclusion

Given, S, a non-empty sub-set of activities executing on the same resource, and activity $A \notin S$, on the same resource as the activities in S, edge-finding exclusion operationalizes implications (9) and (10).

$$\left[\begin{array}{l}(lft(S) - est(S) < dur_A + dur(S)) \\ \wedge\, (lft(S) - est_A < dur_A + dur(S))\end{array}\right] \rightarrow est_A \geq est(S) + dur(S) \qquad (9)$$

$$\left[\begin{array}{l}(lft(S) - est(S) < dur_A + dur(S)) \\ \wedge\, (lft_A - est(S) < dur_A + dur(S))\end{array}\right] \rightarrow lft_A \leq lft(S) - dur(S) \qquad (10)$$

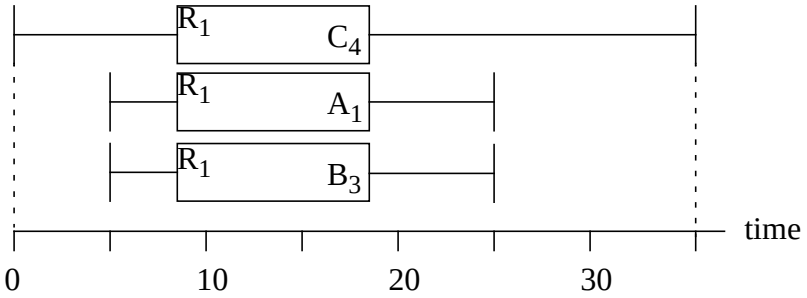


Figure 7. An Example Where Edge-finding Exclusion Can Infer a New Constraint: $ST_C \geq 25$.

Implication (9) states that if *A* is scheduled at its earliest start time and there is not enough room for all the activities in *S* before the latest finish time of *S*, then *A* must occur after all the activities in *S* have finished. Implication 1 can be used to derive a new earliest start time of *A*. Similarly, Implication (10) is used to find a new latest end time.

Complete examination of the 2^n subsets of activities on a resource is not practical. However, it is possible to deduce the same consequences by examining only n^2 subsets of activities on each resource [Carlier and Pinson, 1989; Caseau and Laburthe, 1994; Caseau and Laburthe, 1995]. Algorithms to do this with time-complexity of $O(n^2)$ [Nuijten, 1994] and $O(n \log n)$ [Carlier and Pinson, 1994] for each resource have been presented.

Figure 7 presents an example where edge-finding exclusion is able to infer not only that C_4 must execute after A_1 and B_3, but also that the earliest possible start time of C_4 is 25. Implication (9), where $S = \{A_1, B_3\}$ and $A = C_4$, is applicable and results in a new lower bound on the start time of C_4.

Empirical evaluation of edge-finding for job shop scheduling has been performed on the Operations Research library of benchmarks [Beasley, 1990]. Results show that the LJRand heuristic (Section 2.3.5) plus edge-finding exclusion and a number of simpler propagators can find makespans approximately 25% closer to optimal than the heuristic by itself and further out-performs ORR/FSS with the same set of propagators [Nuijten, 1994].

Subsequent work has extended the exclusion rule (and the not-first/not-last rule discussed below) beyond the unary capacity constraints of job shop to multi-capacitated resources [Nuijten, 1994; Nuijten and Aarts, 1997], cumulative resources [Caseau and Laburthe, 1996], and preemptive scheduling [Le Pape and Baptiste, 1996; Le Pape and Baptiste, 1997]. Other work [Caseau and Laburthe, 1995; Caseau and Laburthe, 1996] has extended edge-finding by adding additional valid implications and has demonstrated very favorable empirical results against a number of polyhedral (cutting plane) methods from the OR literature.

2.4.3 Edge-finding Not-First/Not-Last

While the edge-finding exclusion rules discover that an activity must execute before or after a set of activities, the edge-finding not-first/not-last rules discover that an activity cannot execute first (or last) among a set of activities.

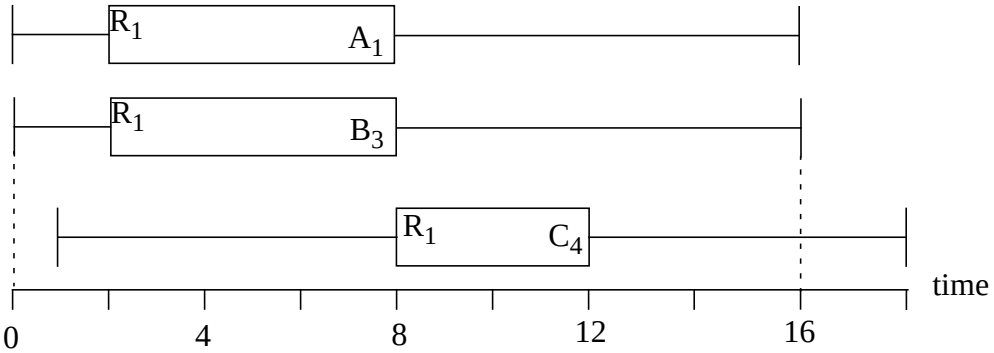


Figure 8. An Example Where EdgeFinding Not-First/Not-Last Can Infer That C_4 Must Execute After Either A_1 or B_3 (adapted from [Nuijten, 1994]).

Given, *S*, a non-empty sub-set of activities executing on the same resource, and activity $A \notin S$, on the same resource as the activities in *S*, edge-finding not-first/not-last implements implications (11) and (12).

$$(lft(S) - est_A < dur_A + dur(S)) \rightarrow est_A \geq min_{B \in S}(eft_B) \quad (11)$$

$$(lft_A - est(S) < dur_A + dur(S)) \rightarrow lft_A \leq max_{B \in S}(lst_B) \quad (12)$$

Implication (11) states that if *A* is scheduled at its earliest start time and there is not enough room for all the activities in *S* before the latest finish time of *S*, then *A* must occur after at least one of the activities in *S* have finished. Implication 1 can be used to derive a new earliest start time of *A*. Similarly, Implication (12) is used to find a new latest finish time for *A*.

An $O(n^3)$ for edge-finding not-first/not-last is presented in [Nuijten, 1994] and a number of other partial implementations exist at lower complexity (*e.g.*, [Carlier and Pinson, 1989; Carlier and Pinson, 1994; Caseau and Laburthe, 1994; Baptiste et al., 1995b]). The current best known complexity for the full implementation of the not-first/not-last rules is an $O(n^2)$ due to [Baptiste and Le Pape, 1996].

Figure 8 presents an example (taken from [Nuijten, 1994]) where edge-finding not-first/not-last is able to infer that activity C_4 must execute after either A_1 or B_3. The earliest start time of C_4 can therefore be updated from 1 to 6, the minimum earliest finish time of $S = \{A_1, B_3\}$.

2.4.4 Open Issues

1. Given the theoretical basis for propagators as consistency techniques, there may be other forms of consistency that can be used in scheduling. Given the excellent results reported, novel propagators may further improve performance.
2. Can we selectively apply propagators to particular sub-graphs rather than blindly apply them across all resources? Anecdotal evidence suggests that on some problems, the constraints

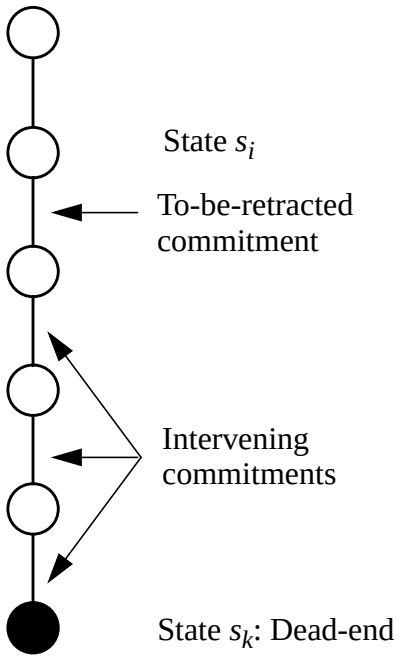


Figure 9. A Search at a Dead-end.

inferred by edge-finding often involve a small number of activities. If structures within the scheduling problem where a propagator is likely to be useful can be cheaply identified, it may be possible to avoid the use of propagators where they are unlikely to contribute. (An example of such a selective application of a propagator in a close interaction with a commitment retraction technique is presented in [Sadeh et al., 1995]).

3. Applications of edge-finding to other types of scheduling (multiple capacitated [Nuijten, 1994; Nuijten and Aarts, 1997], cumulative [Caseau and Laburthe, 1996] and preemptive [Le Pape and Baptiste, 1996]) have resulted in evidence indicating that edge-finding, though still useful, has less impact where constraints are more complex. There is, therefore, the dual challenge of extending these techniques for more complex constraints and, should they be found wanting, identifying new propagators.
4. Because propagators find implied commitments, it is reasonable to expect that they will perform better in more constrained problems. Indeed, the results on non-job shop constraints may be due to the fact that the more complex constraints are not as constraining. A correlation of constraint tightness (and other problem characteristics) to propagator performance will not only aid in understanding propagators, but may speak to the other research issues.

2.5 Retraction Techniques

Assume that a search algorithm moves through a sequence of states $S = (s_0, s_1, s_2, \ldots, s_k)$ as a result of the assertion of a number of commitments $A = (a_1, a_2,...,a_k)$. Further assume that a mistake is made: as a result of one or more of the commitments in A we have reached a search state s_k which is inconsistent with respect to the constraints in the problem. This is a dead-end in the search. This scenario is presented in Figure 9.

To deal with a dead-end we must retract some commitments, $C \in A$. The retraction component of the search strategy must then answer two questions:

1. Which commitments should be retracted?
2. In retracting a commitment that was made, say at state s_i, where $i < k$, what should be done with the intervening commitments, that is those made in all states s_j, where $i < j < k$?

Different retraction techniques supply different answers to these questions.

2.5.1 Choosing Commitments to Retract

Techniques for identifying, *C*, the commitment to be retracted, fall into two general categories:

1. Provable: Provably, no solution exists in the area of the search space defined by *C* and the commitments made prior to *C*. The proof is typically necessary but not sufficient: at least the commitment *C* must be retracted; however, prior commitments may also need to be retracted to escape the dead-end.
2. Heuristic: Based on a heuristic evaluation, it is determined that *C* is likely to be the cause of the dead-end.

2.5.1.1 Provable Retraction

Provable retraction appears reasonable: until it can be proved that there is no solution given *C* and the commitments preceding *C*, the search should concentrate on the states in that area.

There is a distinction between provability and completeness. It is possible to have a retraction technique (such as Limited Discrepancy Search (LDS) [Harvey, 1995; Harvey and Ginsberg, 1995] (Section 2.5.1.2)) that is complete but not provable, that is, that retracts commitments even though there may be a solution in the sub-space.To maintain completeness, the sub-space is revisited later in the search if no solution has been found elsewhere.

The simplest scheme is chronological retraction: the most recent commitment is retracted. Clearly, since the search space under the most recent commitment contains one state and it is a dead-end, we can retract the most recent commitment without missing a solution. However, it may be that a commitment made much earlier in the search is responsible for the dead-end that has only been discovered now. Chronological retraction will exhaustively search the sub-space below that wrong commitment before finding and retracting it.

If chronological retraction is to be improved upon, while still maintaining provability, it is necessary to exploit the situation where chronological retraction does too much work. For example, assume some set of commitments, C_1, lead to a state s_1, and then a second set of commitments C_2 lead from state s_1 to state s_2. Further suppose that in s_2 it is discovered that none of the values for variable, *V*, are consistent with the current search state. It might be the case that, had we decided to make commitments involving *V* at state s_1, we would have discovered precisely the same dead-end. In other words, the commitments in C_2 do not contribute to the dead-end. Chronological retraction will revisit all the search states between s_1 and s_2, and try all the possible commitments exhaustively before returning to a state prior to s_1 where, finally, it may be possible to escape the dead-end. All provable retraction techniques rely on some mechanism to identify the most recent state at which it is possible to escape the dead-end.

Full dependency-directed backtracking [Stallman and Sussman, 1977] is the most extreme of these methods as it ensures that the correct state will be identified. The time and space complexity

of maintaining the dependency information is in the same order (exponential) as the search itself. Other methods (*e.g.*, backjumping [Gaschnig, 1978], conflict-directed backjumping [Prosser, 1993], graph-based backjumping [Dechter, 1990], and dynamic backtracking [Ginsberg, 1993]) use various techniques to prove necessity rather than sufficiency: it is necessary to jump back *at least* as far as the identified state, but it might be that subsequent search shows that the jump was not sufficient to escape the dead-end. The intervening commitments are not causes of the dead-end and so no solutions will be missed. Empirically, these techniques are able, in the average case, to make significant improvements over chronological retraction.

While many of these techniques show good average time performance on constraint satisfaction problems, few have been applied to scheduling where the most common method of provable retraction is chronological.

2.5.1.2 Heuristic Retraction

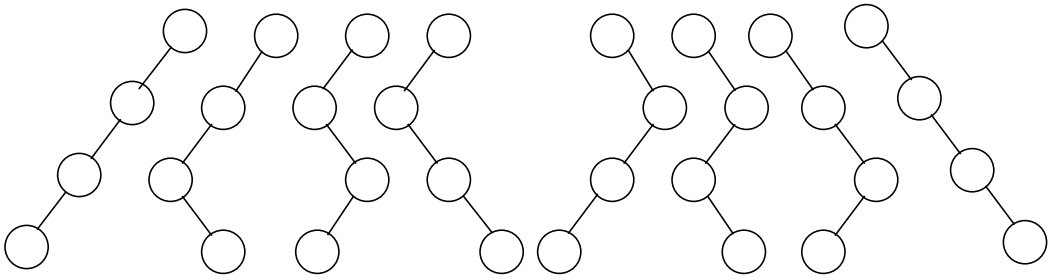
The major difficulty with provable retraction is that the search can get stuck in a sub-space that takes so long to prove to not contain a solution that the problem cannot be solved in a reasonable amount of time. It might be better to be more opportunistic in the selection of commitments to be retracted and therefore move more freely in the search space. The problems with heuristic retraction are that it may abandon completeness (*i.e.*, there is no guarantee that the strategy will find a solution or prove that none exists) and that multiple search-states can be re-discovered a large number of times.

Restart. The simplest heuristic retraction technique is to restart the search whenever a dead-end is found. This technique is called iterative sampling or restart. Provided the heuristic commitment component has some randomness,[6] restarting the search is very likely to explore new states. Unless the search space is fairly dense in terms of solutions, however, incompleteness can be damaging. Nonetheless, there has been successful scheduling done using simple iterative sampling on a set of job shop scheduling problems [Crawford and Baker, 1994].

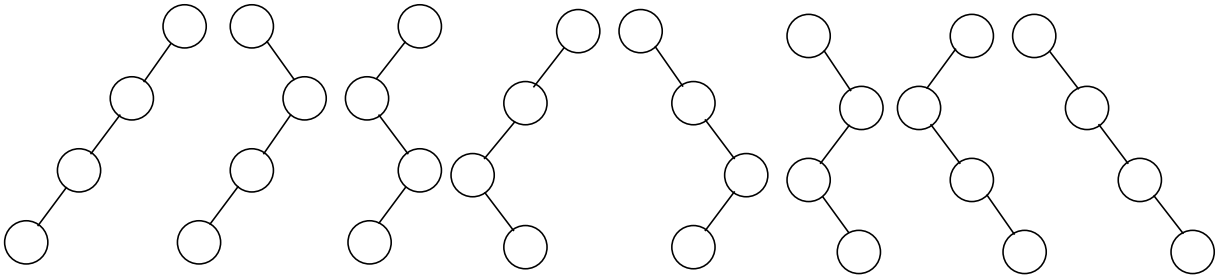
It is easy to adapt iterative sampling to produce bounded chronological retraction with restart [Nuijten, 1994]. When a dead-end is found, a bounded number of chronological retractions are performed to investigate the neighborhood of the dead-end. If no solution is found, it is heuristically concluded that there is no solution in the area and so the search is restarted. Good results with this technique as part of a full strategy are discussed in Section 2.3.5. A similar retraction technique, the Incomplete Backjumping Heuristic (IBH), is proposed in [Sadeh et al., 1995]. After a bounded number of chronological retractions, the retraction returns to the root commitment. Sadeh's heuristic (ORR/FSS) does not have a random component, so rather than restart, the search selects the next best commitment in the initial search state. Empirical results indicate that IBH is particularly effective where a dead-end is the result of a complex interaction among activities on more than one resource.

There has recently been some interesting theoretical evidence supporting the use of retraction techniques that include restart [Gomes et al., 1998]. The authors show the "heavy-tailed cost distribution" phenomenon where at any time during an experiment there is a non-trivial probability of finding a problem requiring exponentially more search effort than any problem that has yet been encountered. This heavy-tailed distribution also exists when the search procedure is run mul-

6. This does not necessarily mean that the forward search must be completely random. It is straightforward to construct a forward step where the heuristic's suggestion for a commitment is probabilistically ignored.



Chronological Backtracking



LDS (Only the first visit to each leaf-node is shown)

Figure 10. A Comparison of Traversals of the Search Space for a Binary Tree of Depth 4.

tiple times on a single problem. These matching distributions suggest that the way to solve problems in a class that exhibits a heavy-tailed cost distribution is with a partially randomized forward step and a retraction technique that includes restart. If a particular probe of the search enters the high-cost region of the distribution, not too much effort will be spent before restarting. If the probe encounters the low-cost area of the distribution, a solution is likely to be found before the restart is triggered. It is not clear what characteristics of a problem class lead to the heavy-tailed cost distribution or which classes of scheduling problems might exhibit these distributions; however, this work begins to work toward an interesting theoretical justification for incorporating some form of restart in a retraction technique.

Limited Discrepancy Search. Limited Discrepancy Search (LDS) [Harvey, 1995] maintains completeness at the cost of significant effort spent in the rediscovery of search states that have already been visited. If the search exhausts the entire search space, LDS visits polynomially more search states than chronological retraction. LDS is based on three ideas:

1. With good heuristics, the solution is expected in states where the heuristic is wrong a limited number of times. Indeed, this seems a reasonable definition of a “good” heuristic.
2. Heuristics are more likely to be wrong earlier in the search.
3. A wrong commitment early in the search has more impact.

Based on these intuitions, LDS follows the heuristic during its first probe into the search tree until arriving at a dead-end. The search is restarted and the probes investigate all paths in the search tree with up to one search state where the heuristic is ignored: all paths with discrepancy level 1. Discrepancies early in the search are explored first. In the second phase of the search, the first path (the one where there were no discrepancies) will be rediscovered. The discrepancy level is incremented every time all the search paths up to the current level are exhausted. The search of a binary tree up to level 4 for both chronological backtracking and LDS is shown in Figure 10.

Each iteration with a particular discrepancy level will rediscover all the search states found at all previous levels. This weakness is overcome in Improved LDS (ILDS) [Korf, 1996]. ILDS generates more states than chronological retraction but, at worst, the number of ILDS states is bounded by a linear factor of the number of chronologically generated states.

Empirical evaluation of LDS on job shop scheduling problems (with a partially randomized heuristic commitment technique) shows significant gain over chronological retraction and iterative sampling, and comparable results to bounded backtracking with restart. The best performance was attained with a combination of LDS with bounded backtracking [Harvey, 1995].

Retraction for Local Search. At the extreme of opportunistic retraction techniques are the local search methods that move in the search space with few limitations. The retraction is completely driven by the heuristics. If the search is free to follow the gradient defined by the neighborhood function, however, it is likely to discover a locally optimal state from which it cannot move: all neighboring states are worse than the current one, yet the current one is not a global solution. A number of techniques to escape local optima have been suggested:

- simulated annealing [Zweben et al., 1993; Zweben et al., 1994] where, with some probability, a worse neighbor will be accepted as a new state and where the acceptance probability decreases as problem solving continues,
- tabu search (see Section 3.3.4),
- genetic algorithms (see Section 3.3.5), and
- the shuffle technique [Baptiste et al., 1995a] where each commitment is probabilistically retracted and forward search is continued with the remaining commitments.

In general, local search with such escape techniques has been shown to perform very well. However, these techniques are often treated as abstract frameworks (“meta-heuristics”) for algorithms rather than algorithms themselves, and so comparison between local search algorithms specifically crafted for a benchmark set with more general strategies is questionable.

2.5.2 Dealing with Intervening Commitments

When the to-be-retracted commitment has been identified, there are three encompassing ways to deal with commitments made in states between the dead-end and the state where the to-be-retracted commitment was made: retract all, retract some, or retract none.[7]

2.5.2.1 Retract All

The most principled way to deal with intervening commitments is to retract them all. The intuition for this is that each commitment depends on all commitments made previously. If the search is jumping back to retract a commitment, all the commitments that were made subsequently are retracted because their justification no longer exists. Iterative sampling or restart, dependency directed backtracking, the host of provable retraction techniques (except dynamic backtracking), and LDS all follow this method.

7. Chronological retraction does not need to be concerned with intervening commitments as it always chooses to retract a commitment in the most recent state and therefore there are, by definition, no intervening commitments.

Bounded chronological backtracking with restart is a special case as sometimes (*i.e.*, when performing chronological backtracking) there are no intervening commitments to retract, while at others (*i.e.*, restart) all the intervening commitments are retracted.

2.5.2.2 Retract Some

Assuming that the justification for a commitment no longer exists simply because a previous commitment is retracted may be too conservative. The retracted commitment may have had no bearing on some of the intervening commitments: by retracting an intervening commitment the search is discarding information that must be rediscovered later when precisely the same commitment is made again. This is strongly argued in [Ginsberg, 1993] and used as motivation for dynamic backtracking.

In dynamic backtracking, limited dependency information (used to choose the to-be-retracted commitment in backjumping) is cached so that it is not necessary to make the assumption that all intervening commitments depend on the retracted commitment. In fact, only those intervening constraints that may have depended (based on the cached information) on the retracted commitment are retracted. Empirical evidence [Ginsberg, 1993] shows significant improvement over backjumping on the CSPs tested. Interestingly, further evidence [Baker, 1994] shows that in some situations dynamic backtracking can be exponentially worse than chronological backtracking! The intuition is that dynamic backtracking changes the heuristic ordering of commitments as the search proceeds. This ordering has significant impact on the search and therefore modifying it can lead to highly inefficient behaviour. From another perspective, this intuition can be restated as follows: although the intervening commitments do not directly depend on the to-be-retracted commitment, the state created by the to-be-retracted commitment leads via the heuristic to the intervening components. Given the retraction, the heuristic may well guide the search in a completely different direction that would not result in the intervening commitments being made at all. While the intervening commitments are not directly dependent on the retracted commitment, they are dependent *via the heuristic component*.

2.5.2.3 Retract None

Finally, in typical local search algorithms such as hill-climbing, tabu, simulated annealing, and shuffle none of the intervening commitments are retracted. The intuition here is that if they actually do need to be retracted, subsequent iterations will determine the retraction that must be done.

2.5.3 Open Issues

1. There are a number of retraction techniques that make compromises between the ability to heuristically move through the search space and the maintenance of completeness [Havens, 1997]. These techniques can often be characterized by the cache of information that is used to select the commitment to retract and/or determine the fate of the intervening commitments. Dynamic backtracking and tabu search begin to look very similar when it is realized that the contents and size of the cache are the main differences. So too for dependency-directed backtracking (with an exponential size cache) as compared with restart and LDS (with caches of zero size). Can we use the size and content of the cache to construct new retraction techniques with varying trade-offs between the completeness and heuristic responsiveness?
2. There would seem to be a middle ground between local search algorithms and, say, LDS, in terms of the ability to respond heuristically. For example, it would be interesting to record a

confidence with which all commitments are made and, when a dead-end is reached, use the confidences to identify the commitment to retract.

3. Three examples of retraction techniques discussed above are actually combinations of other retraction techniques: bounded chronological backtracking with restart, IBH, and bounded LDS backtracking with restart. Are there other retraction techniques that may be combined to improve performance over the individual techniques by themselves?
4. Is the completeness of the retraction technique relevant? In a large problem it is practically impossible to explore more than a small percentage of the search space, so why should we care that "eventually" our search will cover the entire space?

2.6 Scheduling with Inventory

The management of inventory, its storage, production, and consumption, represents the core function of a manufacturing organization, be it a diversified global manufacturing enterprise or a single factory work-center. Manufacturing is primarily concerned with the transformation of raw materials into finished goods. The economic viability of a manufacturer depends on the efficiency with which this transformation can be achieved.

2.6.1 Problem Definition

In manufacturing, activities may produce and/or consume inventory that must be stored before consumption and after production subject to minimum and maximum limits on the amount of each inventory that can be stored at any time. Scheduling, then, must take into account not only the temporal and resource constraints, but also these minimum and maximum inventory constraints.

An inventory problem can be defined as follows, given:

- A set of *process plans*. Each process plan defines a temporally connected set of activities for the production of a particular inventory or set of inventories. Each activity has a duration, resource requirements, and, perhaps, inventory requirements.
- A set of *resources*. Each resource has maximum capacity specifying the number of activities that can simultaneously execute on the resource. This maximum may vary across the scheduling horizon.
- A set of *inventories*. Each inventory has both a minimum and maximum storage capacity, specifying, respectively, the smallest and largest amount of the inventory that can exist at any time. The capacity constraints may vary, independently, across the scheduling horizon with the only requirement being that the minimum can not be greater than the maximum at any time point. A non-zero amount of each inventory may be present at the beginning of the scheduling horizon.
- A set of *supply and demand event*s. Each event indicates a time interval during which an external force will instantaneously add (supply) a specific amount of inventory to or remove (demand) it from the plant.

Find:

- An instantiation of process plans and an assignment of start times and resources to each instantiated activity such that all resource, inventory, and temporal constraints are satisfied. It is, in addition, desirable to minimize the inventory levels at any time (while still satisfying the minimum inventory constraints).

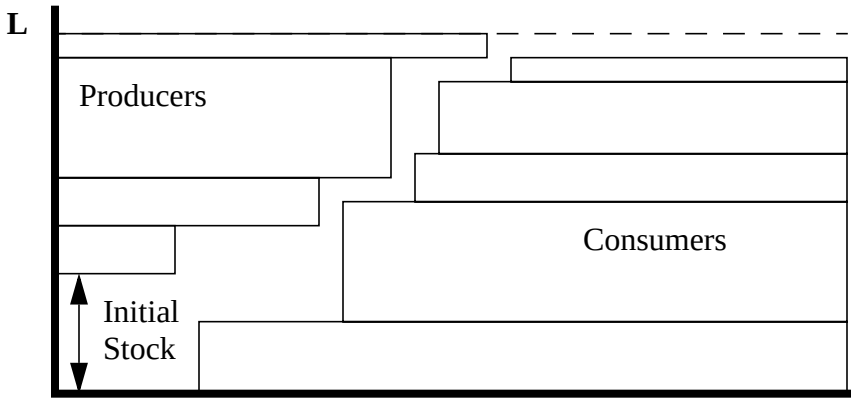


Figure 11. Using the Cumulative Constraint to Model Inventory (from [Simonis and Cornelissens, 1995]).

The inventory problem encompasses the Generalized Resource Constrained Project Scheduling Problem [Herroelen and Demeulemeester, 1995] while adding inventory production and consumption, inventory constraints, and the need to reason about and instantiate the to-be-scheduled activities.

The above description of inventory scheduling contains many characteristics that have not been systematically addressed in the scheduling literature. Indeed, the necessity of instantiating process plans blurs the line between scheduling and planning (see Chapter 7). The problem model addressed in this dissertation will therefore be simpler than the above description (see Section 6.1.1).

2.6.2 Previous Work

Given the importance of inventory in industrial scheduling, many commercial scheduling systems represent and reason about inventory in some way. Unfortunately, little is known about the techniques that are used by these systems, and, indeed, a comparative study of inventory scheduling techniques does not appear in the literature. We look at the few systems for which published descriptions do exist: CHIP, ILOG Scheduler, and KBLPS.

In CHIP [Simonis and Cornelissens, 1995], the cumulative constraint is used to represent inventory as a re-usable resource. An activity which produces an inventory at some time, t_1, is represented as an activity that uses the corresponding resource from time 0 to time t_1. At t_1 the activity ends and the resource is released for use by other activities. Similarly, an activity which consumes an inventory at some time, t_2, is represented as an activity which uses the corresponding resource from t_2 to the end of the scheduling horizon. The consuming activity, therefore, begins at t_2 and ends at the end of the horizon. Graphically, the model can be pictured as in Figure 11, taken from [Simonis and Cornelissens, 1995]. As long as the producers and consumers do not overlap, the inventory minimum constraint is satisfied.

The cumulative constraint is defined by three sets of domain variables, $[S_1, \ldots, S_n]$, $[D_1, \ldots, D_n]$, $[R_1, \ldots, R_n]$ and a natural number L. Intuitively, L is the maximum amount of resource that can be shared by a set of activities at any time-point. S_j and D_j are, respectively, the start time and the duration of activity j, and R_j is the amount of the resource used by activity j.

Using the convention that for variable *V*, min(*V*) and max(*V*) are respectively the smallest and largest values in the domain of *V*, the cumulative constraint holds if:

$$\forall(t \in [a, b])\forall(j | S_j \leq t \leq S_j + D_j - 1)\sum R_j \leq L \tag{13}$$

Where:

- $a = \min(\min(S_1), \ldots, \min(S_n))$
- $b = \max(\max(S_1)+\max(D_1), \ldots, \max(S_n)+\max(D_n))$

This formulation is directly applicable to the inventory minimum constraint of 0. The authors are able to model inventory minimums of $m > 0$ by simply decreasing the initial stock level by *m*. This essentially removes *m* units of the inventory from the model so that reasoning about an inventory minimum of 0 is valid. A further extension, by reversing the role of the producer and consumer activities, allows for the modeling of a maximum constraint on the inventory. Other extensions (*e.g.*, variable rate production and consumption) are discussed by the authors.

In ILOG Scheduler, one method of inventory modeling is the use of a time-table mechanism [Le Pape, 1994c; Le Pape, 1994b].[8] Each inventory has a time-table defining the time-varying minimum and maximum capacity constraints. Activities produce and consume inventory, and propagation is done through the time-tables to prune start times that are not consistent with the inventory constraints.

In the KBLPS distribution planner (DP) [Saks, 1992], inventory is treated in a more discrete fashion with the notion of a *Product Supply.* A Product Supply is an entity with the characteristics of product type, uncommitted quantity, time available, and initial location. The DP algorithm treats the initial inventory and subsequent incoming supplies as separate discrete quantities. In addition, DP schedules all the activities in one order before moving to the next order. This enables the algorithm to use texture measurements to identify the most contended for resource or Product Supply, and to schedule an order which relies most on that resource or Product Supply. Such an order-based reasoning reduces the complexity of the inventory representation and reasoning as all the inventories, from raw materials through work-in-process to finished goods are accounted for when an order is scheduled.

2.6.3 Discussion

In CHIP and ILOG Scheduler, it appears that the primary use of the inventory modeling is for dead-end detection and propagation.[9] With such an approach, traditional scheduling algorithms can be used to assign start times to activities, while the inventory propagation maintains the inventory constraints. One omission of this approach is that no heuristics examine the inventory con-

8. Different, proprietary inventory constraint representations are used in recent version of ILOG Scheduler [Nuijten, 1999].

9. While no sophisticated propagators have been defined specifically for inventory constraints, edge-finding has been extended to the cumulative constraint [Nuijten, 1994; Caseau and Laburthe, 1996; Baptiste and Le Pape, 1996] and therefore we expect that it has been incorporated into the algorithms in CHIP. In the case of ILOG Scheduler, such an edge-finding extension does exist, but for propagation of inventory constraints other, proprietary, techniques are used [Nuijten, 1999].

straints. Even if inventory constraints are critically constrained and, therefore, the major challenge in solving a problem, there does not appear to be any commitments that attempt to directly decrease inventory criticality. Given much of the work on scheduling heuristics, it is expected that the ability to identify and reason about inventory criticality would improve scheduling performance.

The KBLPS model, in contrast, directly represents and reasons about inventory as part of the heuristic commitment technique. However, the approach depends on the complete scheduling of an entire order and all inventory transitions for that order, before moving to another order. This order-based decomposition is reminiscent of early constraint-directed schedulers like ISIS [Fox, 1983] and OPIS [Smith et al., 1989] rather than micro-opportunistic approaches [Sadeh, 1991; Nuijten et al., 1993]. In the latter approach, the solver has the flexibility to address the activities in any order, driven by heuristics, rather than based on orders.

2.7 Scheduling with Alternative Activities

It is not uncommon in a realistic scheduling problem to have a number of choices that are not typically represented in constraint-directed scheduling approaches. In particular, there are two extensions of typical research scheduling models that we examine in this dissertation: alternative resources and alternative process plans.

2.7.1 Alternative Resources

Given a facility in need of scheduling (*e.g.*, a chemical plant) and difficulty in creating schedules due to high competition for a resource, the company may attempt to reduce contention by purchasing an additional resource that can run the same activities as the current bottleneck resource. Using our chemical plant example, additional reaction vessels may be purchased to both expand the capacity of the plant and to loosen the scheduling problem. The existence of an alternative resource, however, introduces the need to decide which activities will be performed on which resource. If the alternatives are truly identical, then it may be possible to represent the alternatives with a single multi-capacity resource [Nuijten, 1994]. However, in many cases the resources are not truly identical: one resource may incorporate new technology and so is able to process activities more quickly, produce higher quality inventory, etc.

In its most basic form, the alternative resource problem can be represented with the use of multi-capacity resources. A resource of capacity k can be used to model a resource set of k identical unary capacity resources and the activities can simply be scheduled on the multi-capacity resource without representing the unary capacity resources. For such a model to be correct, the unary capacity resources must be completely substitutable, which implies two characteristics of the problem. First, an activity's characteristics such as duration must be independent of the actual unary capacity resource on which it is executed. Second, the resource sets must be mutually exclusive so an activity that can execute on one unary resource within a set can execute on any of the unary resources in the set. Significant representational savings can be gained from this type of representation. For example, representing a pool of 100 skilled crew-members as a single multi-capacity resource is likely to lead to more successful problem solving than using 100 separate unary resources. Unfortunately, elaborations of the resource model, such as requiring changeover activities [Brucker and Thiele, 1996] tend to invalidate one or both of the problem characteristics necessary to use the multi-capacity resource model.

ISIS [Fox, 1983] provided representation for alternative resources within an order-based Incremental Decomposition and Incremental Scheduling (IDIS) [Kott and Saks, 1998] approach. Initially, an order represents activities together with their resource alternatives. A multi-level scheduling algorithm heuristically identifies an order to be added to the schedule, and then heuristically determines the resource reservations for each activity based on an analysis of the temporal aspects of the order (*e.g.*, due date, precedence constraints, activity durations) and on the set of activities that have already been given resource reservations. All the activities for one order are assigned start times and resource assignments before the next order is selected for scheduling.

A similar order-based decomposition is used in the Dynamic Scheduling System (DSS) [Hildum, 1994]. Based on a black-board system and stochastic order arrival, the DSS system instantiates process plans with resource alternatives that include preference information. Based on analyses of the existing schedule (including resource contention, activity slack-time, and resource availability) and the represented preferences, resource reservations are created for each of the newly instantiated activities. While the scheduling process typically assigns activities to resources in one order before moving to the next, in more challenging scheduling problems, DSS may cancel previous reservations in order to search for a global solution among the currently instantiated activities.

In a constraint programming language, such as ILOG Solver and Scheduler [Baptiste and Le Pape, 1995], alternative resources can be modeled with a set of boolean variables attached to the resource requirements of each activity. These “demand” variables are such that each possible resource for an activity has a corresponding variable. An activity uses a resource only if the corresponding demand variable is TRUE. By using a constraint stating that only one of the demand variables can be TRUE, the alternative resource requirements can be modeled and demand variables can be incorporated into solution techniques.

A combination of the multi-capacity representation and the low-level constraint representation is also possible by using multiple representations of the same resource requirements. In [Nuijten, 1994], multi-capacity resources are used to represent resource sets while, at the same time, the unary capacity resources are directly represented. [Nuijten, 1994] implements a multi-capacity version of the edge-finding exclusion propagator, and uses it on the multi-capacity resources at the same time as using edge-finding and other propagators on the unary capacity resources. Scheduling begins with all activities assigned to their resource sets. The heuristic commitment technique (a modification of the LJRand heuristic (Section 2.3.5)) assigns an activity to its earliest start time and to a randomly selected resource on which the activity can execute at its earliest start time. Extensions of these techniques allow activities to have overlapping resource sets and allow the duration of an activity to depend on the assigned resource alternative. Empirical results show that while the edge-finding leads to better problem solving performance, the gain is not as great as when edge-finding is applied to unary capacity resources in job shop.

Building on the use of multiple representations, [Davenport et al., 1999] formulate a number of heuristic commitment techniques for alternative resource problems. Again, both the multi-capacity resource and unary capacity resources are explicitly represented. The central algorithm for scheduling has two-phases: in the first, all the activities not assigned to one of their resource alternatives are assigned (based on texture measurements of both the multi-capacity and unary capacity resources), and in the second phase the activities on the unary capacity resources are sequenced using job shop scheduling techniques. Throughout the phases, both unary and multi-capacity edge-finding is done on the respective resources. Experimental results show that a slack-based heuristic and a heuristic based on the VarHeight texture measurement (see Section 5.2.4 and

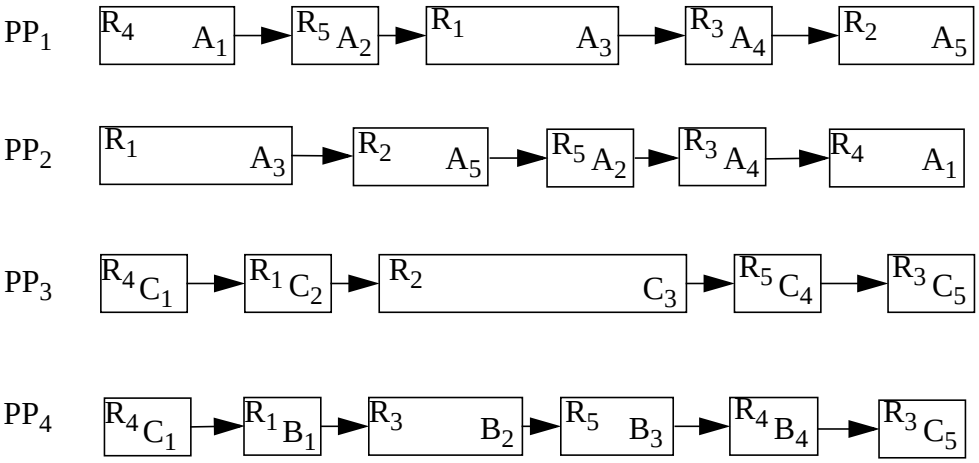


Figure 12. Four Alternative Process Plans.

Section 7.6.1) deliver good performance, significantly improving on the results of [Nuijten, 1994].

2.7.2 Alternative Process Plans

Depending on the flexibility of the production facility, it may be the case that there are multiple ways to achieve the same goal. The different ways may result from flexibility in the sequence of activities, separate processes that result in the same goal, or choices within a process plan. Figure 12 displays four alternative process plans (PP_1, …, PP_4). The label in the upper-left corner of each activity represents the activity's resource requirement while the lower-right label is the name of the activity. Thus, activities with the same name (*e.g.,* A_3 in PP_1 and A_3 in PP_2) are the same. The first two process plans, PP_1 and PP_2, are simply different orderings of the same activities. The third process plan, PP_3, is a completely different recipe while the fourth, PP_4, is a variation on the third: the first and last activities are identical, but the middle ones are different.

As noted by [Kott and Saks, 1998], there is a spectrum of approaches to alternative process plan scheduling. At one extreme, the *Multiple Alternative Decomposition* (MAD) approach, the alternatives are fully represented and integrated in the scheduling process. The other extreme is the *Complete Decomposition Prior to Scheduling* (CDPS) approach where all alternatives are decided in a pre-scheduling phase of the search. In the initial phase, specific alternatives are selected for scheduling based, perhaps, on an abstract analysis of the evolving scheduling problem, but without detailed scheduling knowledge. An intermediate approach along this spectrum is the *Incremental Decomposition and Incremental Scheduling* (IDIS) approach where the search through the alternatives and the scheduling is interleaved. A subset of alternatives may be decided and then scheduled before the remaining alternatives are addressed.

2.7.2.1 Complete and Incremental Decomposition

The order-based decomposition approach of ISIS discussed above (Section 2.7.1) in the context of resource alternatives can also be used to represent and solve problems containing alternative process plans [Fox, 1983]. In addition to alternative resources, the order representation in ISIS allows multiple alternative routings of a single order through a factory. After such an order is chosen for scheduling at the order-selection, the resource allocation heuristics are used to choose a single routing and resource reservations for all activities along the chosen routing.

The KBLPS scheduling system [Saks, 1992; Fox, 1999] builds on the order-based decomposition of ISIS by the direct representation of alternative process plans and the incorporation of alternative information into texture-based heuristics. At a search state, the routing and resource reservation heuristics of ISIS were based on examination of the set of orders that had already been scheduled. In KBLPS, rather than limiting the resource analysis to the already scheduled activities, a probabilistic representation of all activities was used in the spirit of the reliance and contention texture measurements due to [Sadeh, 1991]. Therefore, resource reservation decisions could be influenced not only by the existing partial schedule, but also by a probabilistic representation of the demand from yet-to-be-scheduled activities. The innovation, here, in terms of representation of alternative process plans was to recognize that the probability that a particular routing is chosen depends on the number of alternative routings that are available: the individual resource demand for each activity in a routing was therefore biased to represent the likelihood that the routing itself was chosen. For example, if there were four possible routings for a single order, the *a priori* probability that an activity on one of the routings would execute is 0.25.[10] Therefore, the individual demand of that activity is biased to be only one-quarter what it would be if the activity was a member of a routing with no alternatives. The overall scheduling process remains similar to ISIS in that orders are selected and scheduled before the next order is examined. The difference, however, is that the resource reservation heuristics have deeper probabilistic resource demand information on which to base their commitments.

[Kott and Saks, 1998] extend the KBLPS work by embedding alternative process plan scheduling in the context of combining planning and scheduling. Viewing alternative process plans as the result of decomposing goals, the authors examine the scheduling of multiple, alternative goal decompositions. Using domain specific rules, a set of alternative process plans are constructed for each goal to create the scheduling problem. Then, based on the contention and reliance texture measurements (Section 2.3.1), the activity most reliant on the most contended-for resource is selected. As in KBLPS, the individual demand of each activity is biased by its probability of execution given the alternative process plans that are available to achieve the same goal. The critical activity is used to identify the critical goal, and a single process plan is selected to achieve that goal based on a combination of domain specific rules and the reliance of the critical activity. All the activities in the chosen process plan are scheduled. Temporal and resource propagation is then performed, and any activities that have been made infeasible are removed. The algorithm continues, without backtracking, until all feasible goals have had a process plan scheduled. This technique has been implemented and applied to a number of large-scale scheduling problems involving logistics distribution, medical evacuation, and transportation. These systems are successfully used by various organization; however, a rigorous comparison of these methods to other techniques does not appear in the literature.

A number of observations can be made about the approach used in ISIS, KBLPS, and the work of [Kott and Saks, 1998]:

1. All the activities in the chosen process plan are scheduled at the time that the process plan is selected to achieve a goal. In job shop scheduling, it has been shown that the ability to micro-opportunistically focus on different activities regardless of the job or resource results in improved scheduling performance [Smith et al., 1989; Sadeh, 1991].

---

10. In practice, the KBLPS system used domain dependent preference information to bias the division of probability among alternative routings. It is not necessarily the case, therefore, that each routing in an order is equally likely [Fox, 1999].

2. Nested alternatives cannot be represented or reasoned about [Fox, 1999]: a process plan cannot contain alternative sub-routings. This case arises in particular when activities that are part of an alternative process plan also have alternative resources. Following a least-commitment approach, we may want to commit to a process plan, but delay the commitment on nested alternatives.
3. The propagation techniques such as temporal propagation [Lhomme, 1993] as well as more sophisticated propagation techniques discussed above (Section 2.4) have not been integrated into the reasoning process in the presence of activities that may not execute in a final schedule. These techniques have been shown to significantly extend the reach of constraint directed scheduling techniques and so their full integration with alternative process plans would seem likely to benefit overall scheduling performance.

A higher-level approach to alternative process plan scheduling can be seen in work that integrates the process planning task with scheduling. Rather than having a pre-defined set of process plan alternatives to achieve a goal, systems such as Design-to-Criteria [Wagner et al., 1997; Wagner et al., 1998] and IP3S [Sadeh et al., 1998] dynamically form the actual process plans based on a number of criteria. The Design-to-Criteria work takes into account a sophisticated combination of factors such as uncertainty, cost, and time-to-completion in exploring the space of plans with which a set of goals can be achieved. Based on the high-level evaluation, a set of activities without alternatives is selected and scheduled. Depending on the quality of the resulting schedule, decisions from the planning phase may be revisited in order to achieve a better overall schedule based on the user's criteria. Similarly, IP3S uses a black-board architecture to integrate process planning and scheduling functionality. In an environment where process plans must often be dynamically generated for an order, IP3S achieves this integration by taking into account the resource contention when formulating process plans. As with Design-to-Criteria, non-alternative process plans are generated and scheduled; however, based on the quality of the schedule, alternative process plans may be generated specifically to route process plans away from highly contended-for resources. In both systems, however, the alternative process plans are represented only in the planning phase: no representation of alternatives is present during the actual scheduling.

2.7.2.2 Multiple Alternative Decomposition

[Le Pape, 1994c] extends the activity representation of ILOG Scheduler to allow an activity's duration to be 0 to represent that the activity does not execute. In an application that represents activities with varying durations, all activities are initially present in the problem and the heuristic commitment technique has the choice of setting an activity's duration to 0, setting it to some lower-bound, and/or assigning a start time. This technique allows the representation alternative activities; however, no information is given on what, if any, propagation is done to reason about activity dependencies. For example, it may be the case that as a result of setting the duration of one activity to 0, some other subset of activities must also have their durations set to be 0 as they depend on the former activity. Furthermore, we would like to be able to leverage sophisticated propagators to infer that some activities must have a duration of 0. It is not clear how or if this is done in [Le Pape, 1994c]. In addition, the heuristics discussed are based on random choices and dispatch rules. Again, we would like to be able to make use of more sophisticated heuristics.

2.7.3 Discussion

We draw two general themes from the literature. The first is the constraint programming approach represented by ILOG Scheduler where an alternative is represented by a full variable in the constraint representation of the problem. Much of this work depends on propagation techniques to

significantly limit the heuristic commitments that must be made. However, to our knowledge, none of the sophisticated propagators developed over the past few years have been extended to directly reason about activities that may not exist in a final solution. The second theme is the heuristic approach present in KBLPS and [Kott and Saks, 1998]: resource and start time reservations are made with texture-based heuristics which take into account the fact that alternative choices change the individual demand an activity has for a resource. As discussed above, there are a number of questions about the approach; however, we view it as a significant starting point for the heuristic commitment techniques to address alternative activities.

2.8 Summary

In this chapter, we presented a review of the literature of constraint-directed scheduling concentrating on the three classes of scheduling techniques: heuristic commitment techniques, propagators, and retraction techniques. In addition, we reviewed the literature on the problems of scheduling with inventory and scheduling with alternative activities.

While instances of each class of constraint-directed scheduling technique are typically used together to form a scheduling algorithm, it is not clear from the literature how the classes relate to one another and how a decision to use a particular heuristic commitment technique impacts the performance of a propagator. Furthermore, while propagators are theoretically based on consistency techniques, there does not appear to be much underlying theory regarding heuristic commitment techniques. As a consequence, it is difficult to compare heuristic commitment techniques to one another on a conceptual level.

In the next chapter, we address these issues using the ODO framework for constraint-directed search. The ODO framework makes explicit the relationship among the three classes of techniques introduced in this chapter. In addition, the framework presents the use of texture measurements as a basis for heuristic commitment techniques. This texture-based characterization of heuristic commitment techniques allows us to begin to identify similarities and differences among heuristic commitment techniques.

Chapter 3 The ODO Framework

One of the themes that underlies this dissertation is the development of a deeper understanding of techniques for constraint-directed scheduling. As part of the ODO scheduling project we have formulated a generic constraint-directed scheduling framework that can model a wide variety of existing (and novel) constraint-directed scheduling algorithms. In the previous chapter, we discussed the constraint-directed scheduling literature with respect to three classes of techniques: heuristic commitment techniques, propagators, and retraction techniques. These classes form the functional basis of the ODO framework. In addition to these classes, there are a number of other conceptual components of the ODO framework: the explicit recognition of the constraint graph as the primary representation of scheduling problems, the use of commitments as a unifying basis for constraint-directed search, and the use of texture measurements as a domain-independent foundation for heuristic commitment techniques. After presenting an overview of the ODO framework, we discuss each of the components in depth.

3.1 Overview of the ODO Framework

The ODO framework is a way to understand constraint-directed scheduling algorithms. As such, we believe that the existing constraint-directed scheduling work can be understood within the framework and that this understanding allows a new perspective on that work. We are not necessarily proposing this framework at the implementation level. Depending on the choices made for the various components of the scheduling strategy, close interaction may be required and therefore an implementation may split or merge the components we have identified. That is not to say that the framework cannot also be used at the implementational level: the ODO framework is also the object-oriented architecture of the ODO implementation.

A high-level view of the ODO framework is shown in Figure 13. At this level the objects of the framework are the *constraint graph*, the *scheduling strategy* or *policy*, and *commitments* which are asserted into and retracted from the constraint graph by the policy. This framework for CDS is an extension of the original ODO framework proposed in [Davis, 1994] and [Davis and Fox, 1993].

The constraint graph contains a representation of the current problem state in the form of variables, constraints, and objects built from variables and constraints. A commitment is a set of constraints, variables, and problem objects that the search strategy adds to and removes from the constraint graph. The assertion and retraction of commitments are the only search operators. Figure 14 displays our conceptual model of a constraint graph as the representation of each state in a search tree. As represented in Figure 14, the state transitions are achieved by the modification of the constraint graph through the assertion and retraction of commitments.

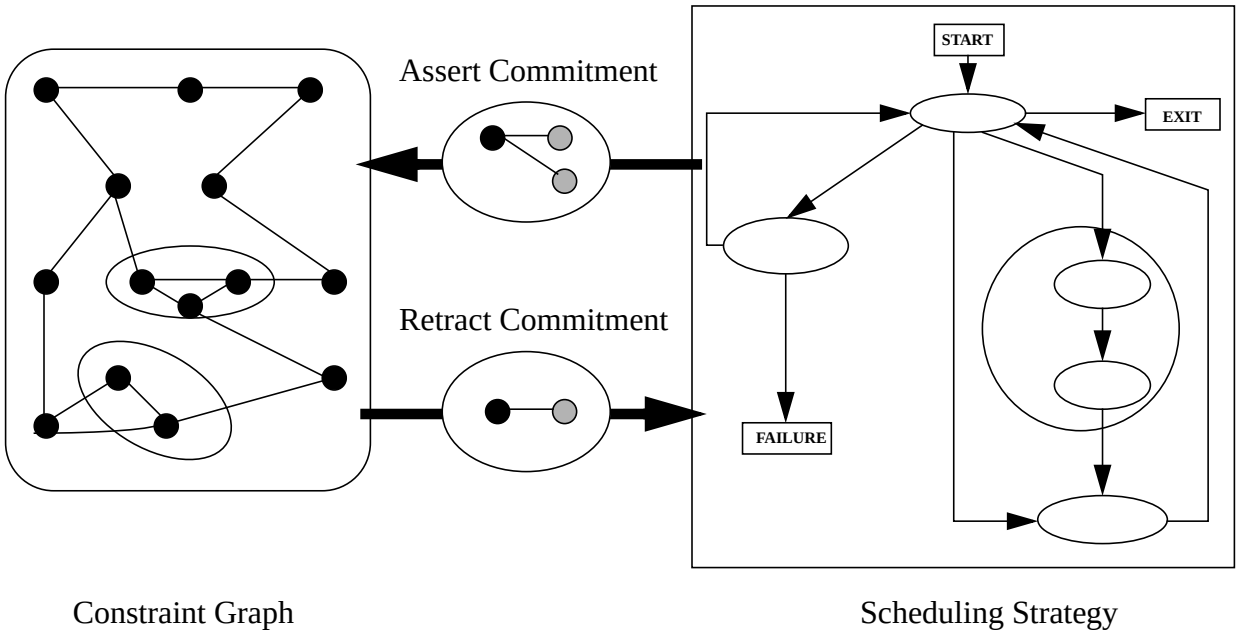


Figure 13. A High-level View of the ODO Framework.

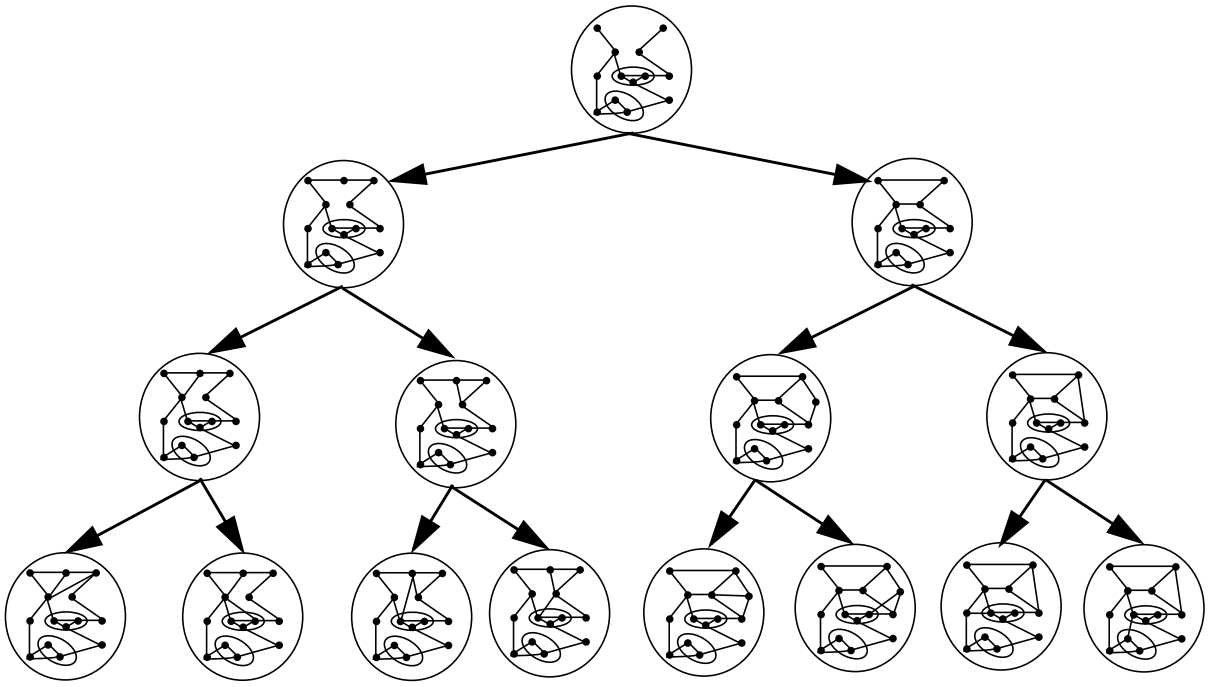


Figure 14. A Conceptual Four-Level Constraint-Directed Search Tree.

A policy contains the components displayed in Figure 15. A pseudo-code representation of a policy is presented in Figure 16. The commitment assertion component is trivial as it requires adding a constraint to the existing graph. The other components may require significant effort.

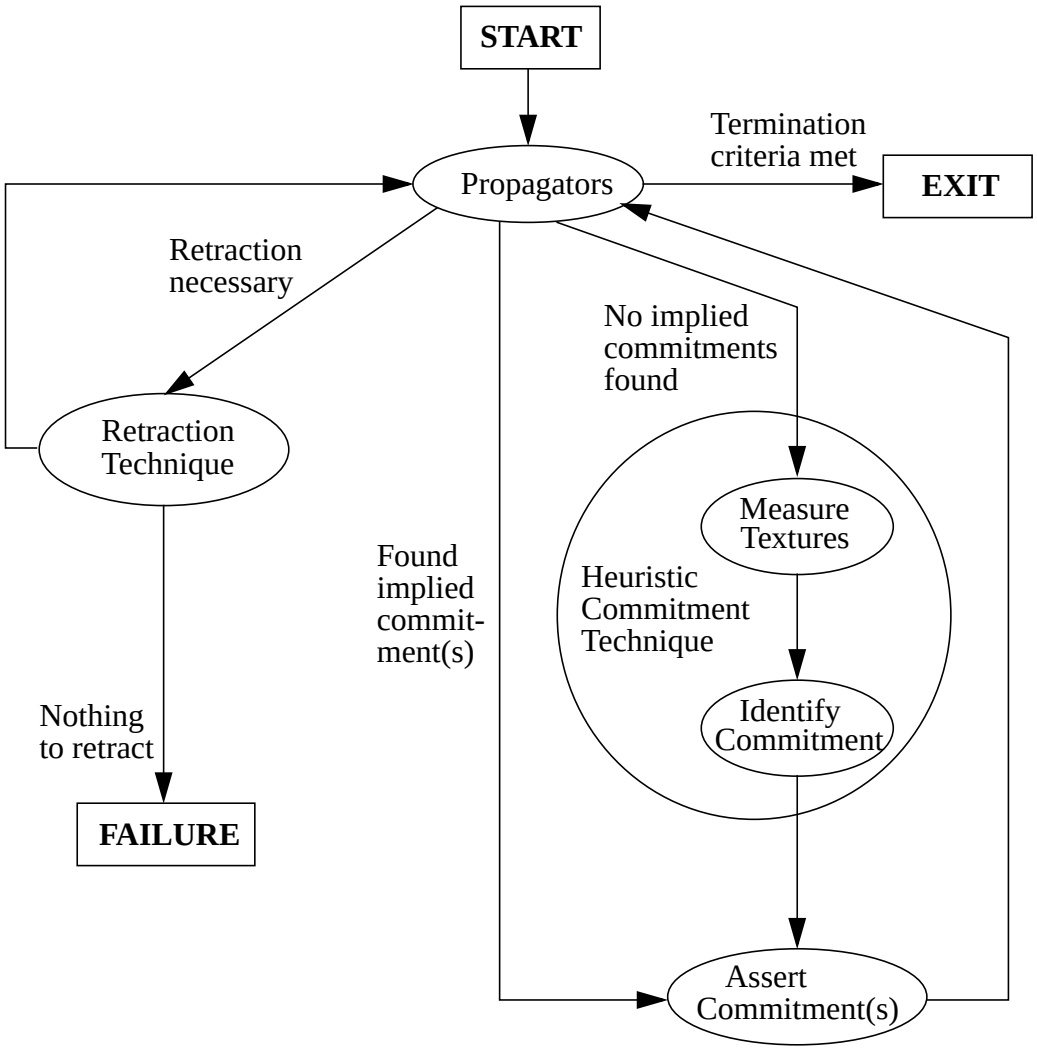


Figure 15. Schematic of a Policy.

A *heuristic commitment technique* is a procedure that finds new commitments to be asserted. It can be divided into two components: the first performs some measurement of the constraint graph in order to distill information about the search state and the second uses this distilled information to heuristically choose a commitment to be added to the constraint graph. A *propagator* is a procedure that examines the existing search state to find commitments that are logically implied by the current constraint graph. A *retraction technique* is a procedure for identifying existing commitments to be removed from the constraint graph. The *termination criteria* is a list of user-defined conditions for ending the search. There may be many criteria including a definition of a solution (*e.g.*, all the activities have a start time and all the constraints are satisfied), determination that a solution does not exist, and limits on the search in terms of CPU time, number of commitments, number of heuristic commitments, and/or number of retractions.

3.1.1 Why the Framework?

The ODO framework provides two important advantages to our research effort. The first is a cognitive model of constraint-directed scheduling: the framework represents a way to *think* about constraint-directed scheduling. Using the framework, we can more quickly understand (and cre-

```
1:  forever
2:     if termination criteria is met
3:        EXIT
4:
5:     while(untried propagators AND
6:           no implied commitments found AND
7:           no retraction necessary)
8:        try next propagator
9:
10:    if (retraction necessary)
11:       retract commitment(s)
12:       if (no commitments to retract)
13:          FAILURE
14:
15:    else
16:       if (no implied commitments)
17:          measure textures
18:          make heuristic commitment
19:       assert commitment
```

Figure 16. Pseudocode for a Policy.

ate) novel scheduling strategies and components, understand the structural similarities and differences among existing strategies, and approach new problem types.

The second advantage of the ODO framework is as an implementational model of constraint-directed scheduling. The high-level concepts of constraint graph, commitment, and scheduling strategy as well as the strategy components all have corresponding objects in our C++ implementation of the ODO scheduling shell. New propagators, for example, can be created by inheriting an interface from an abstract `Propagator` class and then implementing the details of the new propagator. At run-time, then, we can specify that the new propagator (perhaps as one of a set of propagators) is used in a scheduling strategy. The architecture supports rigorous empirical comparison of the components of scheduling algorithms, allowing us to compare, for example, different heuristic commitment techniques while keeping the propagators and retraction techniques constant. It provides the ability to compose novel scheduling strategies simply from the component instances previously implemented. It also allows us to address novel problems with extensions to the constraint graph and commitment representations.

3.2 The Components of ODO

In Chapter 2, we discussed the three main functional components of a scheduling strategy: heuristic commitment techniques (Section 2.3), propagators (Section 2.4), and retraction techniques (Section 2.5). We will not repeat that discussion here. There are, however, three additional aspects of our model constraint-directed scheduling that the ODO framework makes explicit: the constraint graph, the commitment model, and the use of texture measurements as a basis for heuristic commitment techniques. In this section, we examine each of these aspects of the ODO framework.

3.2.1 The Constraint Graph Representation

The constraint graph is the evolving representation of the search state. It is composed of constraints and variables aggregated to form the components of the scheduling problem. Examples of the lower level components include interval variables which can be assigned to a (possibly non-continuous) interval of integer values and constraints expressing various mathematical relationships (*e.g.*, less-than, equal) among interval variables. At the aggregate level, the constraint graph represents, among other scheduling components: activities; Allen's 13 temporal relations [Allen, 1983]; and resources and inventories with minimum and maximum constraints. The components of the constraint graph are not an innovation of the ODO model as most constraint-directed scheduling systems represent these concepts in some way (*e.g.*, [Van Hentenryck, 1989; Le Pape, 1994c; Le Pape, 1994b; Caseau and Laburthe, 1994]).

Neither the implementation-level details of the constraint graph nor the scope of the objects represented are part of the ODO framework. Such prescriptive details and scope would unnecessarily limit the applicability of ODO to scheduling problems with well-understood constraint-based representations. One of the key aspects of constraint-directed search is the extensibility and flexibility of the representation, and therefore the ODO constraint graph is continuously evolving as our research explores new areas of application of constraint-directed scheduling.

3.2.2 The Commitment Model

In traditional, constructive, constraint-directed search, the forward movement through the search space is achieved by the assignment of a value to a variable. Given the heuristic nature of the assignment step, it is likely (in difficult problems) to encounter a dead-end, that is, a state in which at least one variable has no values to which it can be assigned without breaking one or more constraints. At this point some form of backtracking is done: some previously made assignments are undone. Forward search then continues.

An instance of a CSP or COP can also be addressed with a local search procedure (such as tabu search and simulated annealing). Local search techniques work on sets of search states *S*, which may be partial or complete assignments of values to variables. A local search procedure uses a neighborhood function *f* to generate new search states from the current search state: $f(S) \rightarrow S'$. From *S'*, one or more states are selected for exploration. Some analysis is typically done in each new state to decide whether it is acceptable. If so, the neighborhood function is applied to the new state and search continues. If the new state is not accepted, the previous state is returned to and a different neighbor is chosen.

3.2.2.1 Commitments, Assertion, and Retraction

A *commitment* is a variable, a constraint, or a set of variables and constraints, added to the constraint graph representation of the problem during search. *Assertion* of a commitment is the process of adding the problem objects in the commitment to the constraint graph. Thus commitment assertion is a state transition operator. *Retraction* of a commitment is the process of removing a commitment from the constraint graph. Like assertion, retraction is a state transition operator, resulting in a new constraint graph.

Examples of commitments in scheduling include:

- assigning a start time to an activity by posting a unary "equals" constraint (*e.g.*, the start time of activity *A* is equal to 100).

- posting a precedence constraint between activities (*e.g.*, activity *A* executes before activity *B*) [Smith and Cheng, 1993; Cheng and Smith, 1997].
- instantiating a process plan, in response to a demand for some amount of an inventory. A process plan is a set of activities and constraints that together produce some inventory. Assertion of a process plan commitment, like the assertion of any commitment, adds these new objects to the constraint graph.
- adding a new resource to the problem. It may be that part of the scheduling problem is to determine (and perhaps minimize) the number of resources used. Such a problem arises in transportation applications where it is desired to use as few trucks as possible to meet the shipment requirements. A resource, like an activity, is composed of variables and constraints that, with an assertion, are added to the constraint graph.

3.2.2.2 Commitments as a Unification for Constructive and Local Search

Both constructive and local search techniques can be modeled as the assertion and retraction of commitments. This is straightforward in constructive search where the assertion and the retraction are explicit. In local search, the assertion and retraction take place in a single step. Changing the value of a variable can be modeled as an assertion (add the new commitment that assigns the variable to its new value) followed by a retraction (un-assign the variable, that is remove the commitment that assigned it to its current value). A step in the search space, for local search, therefore, is one or more assertions and retractions. Figure 17 shows a schematic example of both styles of search based on assertion and retraction of commitments.

The definition and use of commitments makes no assumptions either about the form of a commitment (other than being a set of variables and constraints), or about why particular commitments are asserted or retracted. The former point is critical in modeling the variety of commitments that are used in constraint-directed scheduling while the latter point is important for the wide applicability of the commitment model. One of the major differences between local search and constructive search styles is in the reasoning that identifies the commitments to be asserted and retracted. By simply modeling the assertion and retraction of commitments without specifying the ways in which the commitments are identified, we are able to account for both constructive and local search techniques.

3.2.2.3 Research Issues

The use of the assertion and retraction of commitments as the sole search operators raises a number of research issues.

1. Constructive and local search are generally viewed as having different strengths and weaknesses. While constructive search, in recent years at least, makes use of both sophisticated propagators and heuristic commitment techniques and is typically systematic and complete, local search generally uses simpler heuristics, appears to scale better in some domains (*e.g.*, hard random 3-SAT [Selman et al., 1992]), and is often easier to understand. An overarching issue is the investigation of hybrid techniques that can combine the strengths of each. For example, can we use propagators with local search techniques?[1] Can we apply the heuristics that are more typical of local search to constructive search or vice versa?

1. Some work has recently been done on this question in the context of vehicle routing [DeBacker et al., 1997].

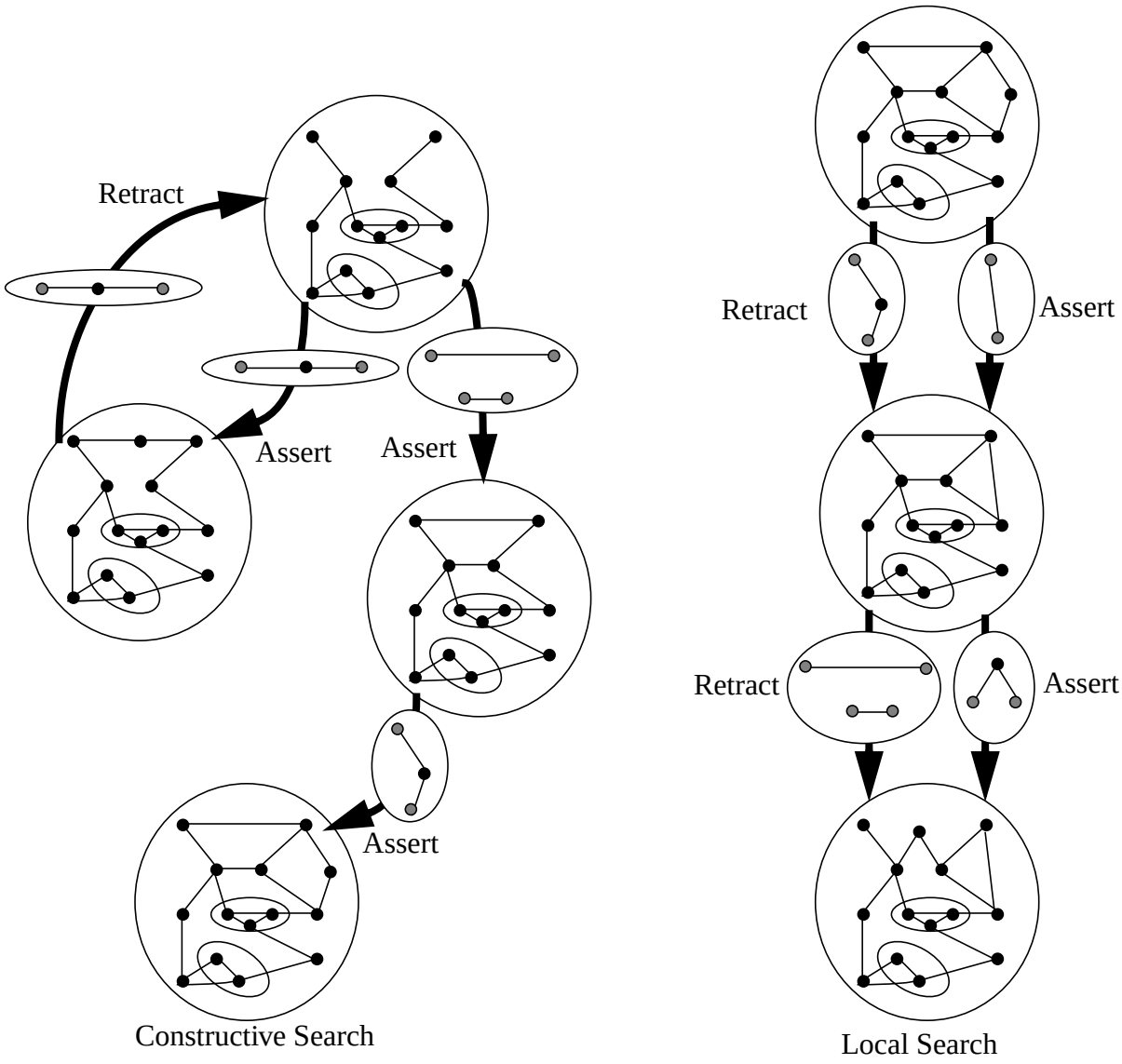


Figure 17. Constructive and Local Search in the Commitment Model.

2. Are there structural characteristics of a problem (and of a sub-problem) that indicate it is better to apply constructive rather than local search techniques or vice versa? Can we move back and forth between constructive and local search while solving a single problem? Can we identify search states where we should change our style from constructive to local or vice versa?
3. Given that the same problem may be solved using different commitments (*e.g.*, job shop scheduling can be solved by assigning start times or by sequencing activities), is there any information that can tell us which types of commitments should be used? Can finding the right commitment type make the search for a solution much easier?
4. Commitments can be of different granularities. “Macro” commitments are large moves in the search space where, for example, all the activities on a unary resource are sequenced (or resequenced) [Smith et al., 1989; Vaessens et al., 1994]. In contrast, “micro” commitments make small steps such as assigning (or reassigning) the start time of a single activity [Sadeh, 1991]. An interesting question therefore is the distinction among commitments of different granularities. Are they all equivalent in some sense (*i.e.*, is a macro commitment simply an agglomera-

tion of micro commitments)? Are there structures in the constraint graph or conditions of the search (*e.g.*, significant backtracking) that suggest that a macro commitment may be of more use? If macro commitments are based on the same information as a micro commitment, are they more cost effective? Is there a trade-off?[2]

3.2.2.4 The Advantages of the Commitment Model

Overall, the use of the commitment model provides two key advantages to our research effort:

1. A new, unifying perspective on local and constructive search that affects how we conceptualize search, and how we address novel problems with CDS techniques.
2. An implementational foundation for the ODO scheduling shell that matches our conceptual model of scheduling. The general commitment model is not only an abstract model of the scheduling work, but also the basis for the object-oriented architecture of the ODO scheduling shell.

3.2.3 Texture Measurements and Heuristic Commitment Techniques

A *texture measurement* is a technique for distilling information embedded in the constraint graph into a form that heuristics can use. A texture measurement is not a heuristic itself. For example, a texture measurement may label some structures in the constraint graph (*e.g.*, constraints, variables, sub-graphs) with information condensed from the surrounding graph. On the basis of this condensed information, heuristic commitments can be made. A relatively small number of texture measurements have been explicitly identified [Fox et al., 1989; Sadeh, 1991]; however, we take the broad view of a texture measurement as any analysis of the constraint graph producing information upon which heuristic commitments are based.

The concept of texture measurements is directly related to the representational and search intuitions discussed in Section 2.1.2. If we are to intelligently search for a solution to a problem and if the problem information is represented in a rich constraint representation, we need to look to the constraints for guidance. We need to distill information from the underlying constraint graph representation of a problem state and base our search commitments on this information.

It is not the case that any possible heuristic decision that can be made in constraint-directed search is necessarily based on texture measurements. However, any heuristic commitment technique that makes use of information in the constraint graph is, at least partially, texture-based. For example, we would not classify a “heuristic” that randomly selects an activity and randomly assigns it a start time as texture-based as it is performing no measurement of the constraint graph.

In general, a texture measurement may be prohibitively expensive (*e.g.*, NP-hard or worse) to compute. Making practical use of texture measurements, then, often requires a polynomial estimation algorithm. For example, the *value goodness* texture is defined to be the probability that a variable, *V*, will be assigned to a particular value, v_a, in a solution [Fox et al., 1989]. To exactly calculate the value goodness we need the ratio of the number of solutions to the problem, where $V = v_a$, to the total number of complete valuations. This is clearly impractical. In practice, therefore, we might estimate the goodness of v_a by examining the proportion of values of connected variables that are consistent with $V = v_a$. We may then base a heuristic commitment on the (estimated) value goodness by choosing to assign the value with greatest (or least) goodness. What information a texture distills, how that information is practically estimated, and what commitment

2. For research that has examined some of these issues see [Clements et al., 1997; Muscettola, 1994; Sadeh, 1991].

is made on the basis of the estimated information form the basic issues surrounding texture measurements.

3.2.3.1 Research Issues

The texture measurement concept produces a number of research issues.

1. What is the information that is to be distilled by a texture? We may not be able to precisely calculate this information in any practical algorithm; however, a firm theoretical basis showing that if we had this information, we could use it to find a solution, allows us to then look to the practical aspects of forming an estimate of this information.
2. Given the impracticality of precisely calculating texture information, can we construct an algorithm that estimates the desired information? It is likely that we can construct a number of algorithms producing estimates of increasing accuracy at the cost of increasing computational complexity. Where is the trade-off in terms of impact on the overall scheduling algorithm? Can we characterize different estimation techniques on the basis of expected error from the true measurements?
3. What are the heuristic commitments that are being made on the basis of the information distilled by the texture measurements? After the texture measurements have been estimated, it is necessary to use that information to make a heuristic commitment. The type of commitment and the heuristic for finding the instance of the commitment, based on the distilled information, may have a significant impact on the search.
4. The *texture hypothesis* is one focus of our research in constraint-directed scheduling. It states that spending a significant, but polynomial, computational effort in measuring textures and making commitments on the basis of the texture information pays off in the reduced need for backtracking and hence results in greater search efficiency. We believe this hypothesis to be true of scheduling though there appear to be CSP domains where it does not hold (*e.g.,* hard random 3-SAT [Selman et al., 1992]).
5. Scheduling problems often require a variety of commitment types: the assignment of activities to resources, the instantiation of process plans, the addition of new resources, etc. Can textures provide a basis for integrating a variety of commitment types in a single search? Based on the constraint graph, can we (heuristically) decide that it is better to make a resource assignment at some point than to sequence activities?

3.3 Scheduling Algorithms as Instances of the Framework

To illustrate the applicability of our framework, we now demonstrate how it can be used to model a number of existing scheduling algorithms.

3.3.1 The ORR/FSS Algorithm

One of the algorithms implemented in the MicroBOSS Scheduler is the Operation Resource Reliance/Filtered Survivable Schedules (ORR/FSS) algorithm [Sadeh, 1991; Sadeh, 1994; Sadeh and Fox, 1996]. It is a constructive algorithm that uses the ORR/FSS heuristic. The texture measurements estimated are the reliance of each activity and the contention for each resource. The heuristic commitment is found by identifying the most critical activity and rating its possible start times in terms of survivability. Survivability is the likelihood that an assignment of the start time will "survive" to participate in a full solution, and is calculated based on the reliance and contention

measures. The commitment made is to assign the most survivable start time to the activity with the highest reliance on the resource and time interval with highest contention.

Two propagators are used: temporal propagation and resource propagation. Temporal propagation (technically, arc-B-consistency) operates on the precedence constraints. If A_i and A_j are activities in the same job such that A_j is a successor of A_i, temporal propagation enforces that: $est_j \geq est_i + dur_i$ and $lft_i \leq lft_j - dur_j$. Resource propagation (arc-consistency) plays a similar role for unary capacity resource constraints. For example, if A_i and B_j require the same unary resource and $lst_j < eft_j$, then for the time interval $[lst_j, eft_j)$, B_j must be the only activity using the resource. Resource propagation will remove the values $[lst_j - dur_i+1, eft_j - 1]$ from the possible start times of A_i. Both propagators remove values from the start time domain of activities by asserting commitments composed of unary disequality constraints.

The retraction technique is chronological backtracking.

3.3.2 The SOLVE Algorithm

The SOLVE algorithm [Nuijten et al., 1993; Nuijten, 1994] is also a constructive algorithm, but takes a different approach than ORR/FSS. In ORR/FSS, the main effort (and computational complexity) is in the heuristic commitment component while the other components are relatively inexpensive. In SOLVE, the heuristic commitment technique is a simpler, less expensive, less powerful technique while the propagators are more expensive and more powerful.

The heuristic commitment technique is the Left-Justified Randomized (LJRand) heuristic. The set of activities that can execute before the minimum earliest finish time of all unscheduled activities is identified. One activity from this set is randomly selected and scheduled at its earliest start time. The heuristic commitment that is made is a unary equals constraint, assigning the start time of the selected activity. The heuristic commitment technique randomly searches the space of “left-justified” schedules which have been shown to form a dominance class in terms of makespan minimization [Baker, 1974]. While it is clearly a measurement of the constraint graph (based on earliest start times and earliest finish times of activities), it is unclear what, if any, underlying constraint graph properties are being estimated by LJRand.

In addition to temporal propagation and resource propagation, SOLVE uses an extensive set of propagators including edge-finding.

The retraction technique is bounded chronological backtracking with restart: chronological backtracking is done until the user-specified bound on the number of backtracks is reached. At that point all commitments are retracted and the search starts over. Clearly, at the root node a different commitment than previously must be made to prevent cycling. With a randomized heuristic commitment technique. the random nature of the heuristic performs this function.

3.3.3 GERRY

GERRY [Zweben et al., 1993; Zweben et al., 1994] is a local search algorithm based on the MinConflicts heuristic [Minton et al., 1992]. Starting from a complete variable assignment with a non-zero cost, GERRY will reschedule some activity in order to reduce the total cost of the schedule. The cost is a weighted sum of the extent to which each constraint is violated. The precedence constraints are always maintained (using temporal propagation); therefore, when applied to job shop, the only constraints that GERRY repairs are the resource constraints. To do this, GERRY reschedules one of the conflicting activities in a MinConflicts fashion, that is, by assigning it to a

new start time that will minimize its conflicts with other activities. GERRY examines moving each conflicting activity to the previous and next time at which the resource is available. Each of these moves is evaluated by a linear combination of factors including the extent to which the size of the activity matches the size of the violation, the number of activities temporally dependent on the activity, and the distance from the current start time of the activity to the new start time. Each move is scored and the score is used to select the heuristic commitment.

In terms of textures, the activity rating procedure estimates contention by calculating the weighted sum of violations for each activity and then estimates the change in contention from moving activities to other start times. Every *l* commitments, a different texture (the overall cost of the schedule) is calculated. If the cost is less than the previous schedule (*i.e.,* from *l* iterations ago), it is accepted as the new schedule. If it is of lower cost than all schedules seen so far, it is also stored as the "best so far" schedule. Even if the schedule is of higher cost than the previous schedule it is accepted at some probability based on a simulated annealing technique [Kirkpatrick et al., 1983].

GERRY can be modeled in our framework by encoding the local search moves as first asserting a new commitment (in this case, start time assignment) and then retracting an existing commitment. Every *l* iterations the retraction is different because the whole schedule is evaluated. If the new schedule is accepted, the usual retraction takes place (after replacing the stored solution with the new one). If the new schedule is not accepted, the previous schedule is put back: the *l* most recent commitments are retracted and the *l* commitments that must be (re)made to return to the previous schedule are asserted.

3.3.4 Tabu Search

In tabu search [Glover, 1989; Glover, 1990; Vaessens et al., 1994] a heuristic *neighborhood function* defines a set of states that can be reached by retracting a set of commitments and then asserting a new set. For example, one neighborhood function swaps the ordering of two adjacent activities (see Section 2.3.6) and another is that used by GERRY. After the neighborhood function has (implicitly or explicitly) defined the set of neighboring states, each state is rated. The state that the search moves to depends not only on the rating, but also on a (potentially) complex mechanism involving a tabu list, an aspiration criteria, and, perhaps, other caches of search information. A *tabu list* is typically a set of states, sub-states, state attributes, or moves that are prohibited as the next state. The form of the tabu list and the criteria for adding and removing elements can be varied and complex, and indeed, are important research issues. A simple example is to have a tabu list of limited length and to insert every move on the list: when the tabu list is full, the oldest moves are removed. While the move is on the tabu list, it cannot be performed. The intuition is that the search should not revisit states too frequently, so as to avoid being trapped in a local optimum of the state-rating function. An *aspiration criteria* is a condition (or set of conditions) for over-riding the tabu list. Like the tabu list, the aspiration criteria can be complex, and a full discussion is beyond our scope. As an example, perhaps a state with an attribute that is on the tabu list is selected anyway because it is more highly rated than any state visited to this point in the search.

Based on the combination of the neighborhood function, the state-rating function, the tabu list, and the aspiration criteria, a search state to move to is selected. The search terminates when a state with the optimal rating is found or when a bound on the number of iterations is reached.

With a conceptualization similar to that used with GERRY, we see that the neighborhood function, the tabu list, the aspiration criteria, and the state-rating function together identify the heuristic commitment that will be made. Once the commitment is identified and asserted, existing

commitments that directly conflict with the new commitment are retracted. Note that the retracted commitments are dictated solely by the heuristic commitment component and in addition there is very little effort spent in identifying conflicting commitments that are to be removed. Because tabu works from a state that does not satisfy all constraints, no sophisticated techniques are used in the retraction technique to identify existing commitments that indirectly conflict with the new heuristic commitment.

The formulation of the neighborhood function, tabu list, and other components of the heuristic commitment techniques are purposely left unspecified in the tabu search definition. In this way, tabu search is more of a framework for local search techniques than a specific search algorithm. In particular, the heuristic commitment technique is free to use any texture measurements to distill information upon which the neighborhood function can be based or as a component of the state-rating function. Propagators are typically not used in conjunction with tabu search (though see [DeBacker et al., 1997] for work on incorporation of propagators into local search), and the retraction technique is wholly driven by the heuristic commitments that are to be asserted.

3.3.5 Genetic Algorithms

Genetic algorithms (GAs) [Holland, 1975; Goldberg, 1989] operate on an evolving population of potential solutions rather than a single solution. An initial population of solutions is created by some (usually randomized) means and each individual schedule is rated. Based on the rating, a subset of existing solutions are allowed to "reproduce" either by cross-over (two new schedules are created from two existing schedules by exchanging some commitments) or mutation (a random sub-set of commitments are changed). The cross-over and mutation operators are usually random, or at least have a random component. After reproduction, potential solutions from the previous generation are discarded, though in some GAs those that are highly rated remain in subsequent populations. The search repeats until a bound is reached on the number of generations or until the optimal solution is found. Clearly, performance of a GA depends highly on the rating function, and on the cross-over and mutation operators.

In terms of our framework, a commitment is an entire schedule (*e.g.*, a set of unary constraints that assign a start time to each activity or, in an alternative formulation, a set of binary precedence constraints that order all activities on each resource). To this point, we have had a single constraint graph to which constraints are added and from which constraints are removed. To represent a population of schedules, we need a set of constraint graphs, one for each population member. A single commitment results in a single population member by the addition of a complete set of constraints to its own constraint graph (*i.e.*, a separate "copy" of the problem).

At each iteration, the heuristic commitment component rates each schedule in the population and generates a new set of heuristic commitments with the reproduction operators. The actual rating function as well as the reproduction operators are left unspecified in the genetic algorithm framework; therefore, texture measurements used as a basis for state-rating or the reproduction operators are left unspecified. As in tabu search, typically no propagators are used (though, of course, it may be possible to incorporate propagators into the unspecified reproduction operators) and the retraction technique is determined by the heuristic component: all those commitments (that is, complete schedules) that will not be present in the subsequent generation are retracted.

Algorithm	Heuristic Commitment Technique		Propagators	Retraction Technique
	Texture Measurements	Commitment Identification		
ORR/FSS	Contention and reliance.	Assign the most survivable start time to the activity with the highest reliance on the {resource, time interval} with highest contention.	Temporal and resource propagation.	Chronological backtracking.
SOLVE	Unknown.	Randomly pick an activity from the identified set and assign it to its earliest start time.	Edge-finding, temporal and resource propagation.	Bounded chronological backtracking with restart.
GERRY	Contention and over-all schedule cost.	Assign activities with highest contention to a start time that minimizes contention.	Temporal propagation.	Retract the previous start time assignment for the activity assigned in the heuristic technique.
Tabu Search	Unspecified.	Unspecified: a combination of neighborhood function, state-rating function, tabu list, and aspiration criteria.	None.	Retract the commitments that directly conflict with the to-be-asserted heuristic commitment.
Genetic Algorithms	Unspecified.	Unspecified: a combination of the state-rating function and reproduction operators.	None.	Retract all commitments that the heuristic technique has determined will not exist in the subsequent generation.

Table 2. **Summary of the ODO Policy Model of Five Example Scheduling Algorithms.**

3.3.6 Summary and Discussion

Table 2 displays a summary of how each of the example scheduling algorithms can be modeled with an ODO policy.

From the above examples, two points, in particular, should be noted:

- In order to account for local search algorithms in our framework, the heuristic commitment technique and retraction technique were functionally combined. The retraction component simply retracts the commitments identified as to-be-retracted by the heuristic commitment component.
- In order to fit GAs into the framework, we needed, in addition, to change a commitment to account for the population of schedules on which GAs operate.

Given these points, the question arises: “Why should we force these disparate algorithms into the framework?”. The answer offered here is that it is not such a leap of faith to conceptualize these algorithms as we have and, even if it were such a leap, the benefits of the unified framework are significant.

We believe the conceptualization of the various algorithms is not forced. To account for local search algorithms, we have created a simple retraction technique that simply carries out the retractions specified by the heuristic commitment component. One could imagine replacing the retraction technique with another that completely, partially, or probabilistically ignores the heuristic commitment component. Alternatively, perhaps the retraction technique could remove all the commitments specified by the heuristic component and then perform further analysis to retract additional constraints. For GAs, our original definition of a commitment noted that it might be a *set* of constraints and other problem objects. The fact that each iteration retracts and asserts a whole population of these commitments does not significantly modify this definition.

Secondly, the benefits from describing these strategies within the framework are significant. The chief benefits are:

1. The ability to perform empirical comparisons of different instances of the same component. As noted in Chapter 2, very little empirical work compares, for example, two heuristic commitment techniques while holding all other components constant. As a result it is difficult to compare strategies, attribute performance observations to specific components, and correlate problem features to performance. The framework allows us to experiment with a variety of combinations, and to generate and test specific hypotheses to explain performance differences.
2. The ability to generate novel strategies and components by combining aspects of existing ones. A case in point is our musing about a retraction component in local search that might retract commitments other than those specified by the heuristic commitment component. In many of the strategies discussed above the main focus of “intelligence” is in only one of the components (*e.g.,* compare ORR/FSS with SOLVE). By making the components more autonomous, we may be able to introduce intelligence into a number of components. By placing GAs within our framework, we can similarly imagine a number of interesting scenarios. To wit:
    - Modify the GA retraction technique to, under certain circumstances (*e.g.,* finding a solution significantly better than all other solutions seen), retract all individual schedules and transfer to, for example, a tabu search on that one solution.
    - Rather than moving to tabu search, perhaps the retraction component could not only retract all but the best solution, but also remove some subset of the constraints from that best solution. The heuristic commitment technique could become ORR/FSS while the retraction component becomes chronological backtracking to try to build a complete schedule from that promising partial schedule.
    - Tabu search, under certain circumstances, may randomly spawn a number of schedules from the best schedule it has found and move to a GA search.

The point is not whether these scheduling strategies will work (that remains to be seen), but rather that being able to conceptualize a variety of strategies within the ODO framework provides a new perspective on these techniques that, in turn, creates a space of novel strategies to be investigated.

3.4 Summary

In this chapter we presented the ODO framework for constraint-directed search. While we introduced the main functional components of the framework (heuristic commitment techniques, propagators, and retraction techniques) in Chapter 2, here we concentrated on the constraint graph representation of problem and search states, the unifying model of assertion and retraction of commitments as the sole search operators, and the use of texture measurements as a basis for heuristic commitment techniques. We demonstrated how a number of existing scheduling strategies could be conceptualized within the ODO framework.

This chapter has emphasized the use of the ODO framework to provide a basis for the conceptual comparison of various scheduling strategies and techniques. While conceptual comparison of scheduling strategies is valuable for the reasons discussed in the chapter, it is also useful to be able to empirically compare the performance of specific instances of scheduling components while holding all the other components of a strategy constant. We turn to such a comparison in the following chapter.

Chapter 4 An Experimental Study of Heuristics for Job Shop Scheduling

In the previous chapter, we highlighted the use of the ODO framework to model a wide variety of constraint-directed scheduling algorithms and to allow their conceptual comparison. Of equal importance is the use of the ODO framework as an experimental tool. The ability to modify scheduling algorithms by substituting one instance of a component with another instance of the same component while holding the rest of the algorithm constant provides us with a tool to conduct rigorous experimental studies.

In this chapter, we examine a number of heuristic commitment techniques for job shop scheduling and formulate a variation of the ORR/FSS heuristic [Sadeh, 1991]. We examine claims that have appeared in the literature concerning the efficacy of texture-based heuristic commitment techniques and re-evaluate texture-based heuristics in light of recent advances in scheduling technology.

4.1 Motivation

Our motivation for the work in this chapter is twofold. First, the work serves as an example of the use of the ODO framework for the experimental evaluation of constraint-directed scheduling components. Second, and more importantly, the central thesis of this dissertation is that, as problems become more complex, knowledge of the problem structure (through, for example, the use of texture measurements) becomes increasingly important in successful heuristic search. While our eventual goal is application of texture-based heuristics to novel constraints (Chapter 6 and Chapter 7), we first to evaluate this thesis in the job shop scheduling paradigm.

The first question to be addressed in examining heuristics for job shop scheduling is the extent to which they are based on a dynamic analysis of a search state and how this analysis correlates with performance. Does sophisticated analysis of the search state by texture measurements correspond to improved overall search performance? Second, a number of criticisms of texture-based heuristics have appeared in the literature. The criticisms call into question our basic hypothesis and therefore, deserve to be addressed before we attempt an extension of texture-based heuristics beyond the job shop model.

4.1.1 Search State Analysis and Scheduling Performance

A central part of the hypothesis regarding texture measurements is that dynamic analysis of the constraint graph representation of the search state can form a superior basis for constraint-directed search heuristics. In this chapter, we investigate three job shop heuristics with varying levels of

dynamic analysis: SumHeight (a variation of ORR/FSS, that is developed in this chapter, see Section 4.3), CBASlack (see Section 2.3.4), and LJRand (see Section 2.3.5).

SumHeight, like ORR/FSS, is based on using the aggregation of the probabilistic individual demand of each activity to estimate the contention for a resource. Of the three heuristics, it makes the most use of dynamic information distilled from the constraint graph in forming heuristic decisions.

CBASlack, though not originally conceptualized as a texture-based heuristic, uses a form of the contention texture to form commitments. CBASlack calculates the biased-slack between all pairs of activities that compete for the same resource and selects the pair with minimum biased-slack as the most critical. We view the pair-wise biased-slack as an estimation of pair-wise contention: the lower the biased-slack the more the two activities contend for the same resource over the same time window. While pair-wise contention distills less information from the constraint graph than the more aggregate measure used in SumHeight, it is nonetheless a dynamic analysis of the search state. Note that pair-wise contention is a measure of how much two activities compete for a resource while the SumHeight texture is a measure of the extent to which all activities compete for a resource. So while SumHeight and CBASlack are both based on forms of contention, they are not based on an identical texture measurement: we expect that the difference in aggregation level will lead to different heuristic performance.

Finally, the LJRand heuristic identifies the unassigned activities that are able to start before the minimum end time of all unassigned activities. One of these activities is randomly selected and assigned to its earliest start time. LJRand performs the least analysis of the search state of the three heuristics tested.

Based on our hypothesis that increased analysis leads to improved scheduling performance, we expect that SumHeight will outperform CBASlack which in turn will outperform LJRand.

4.1.2 Criticisms of Texture-based Heuristics

As noted in Chapter 2, two criticisms of texture-based heuristics have appeared in the literature.

Smith and Cheng. As part of the evaluation of the PCP scheduling algorithm, [Smith and Cheng, 1993] compare Sadeh's ORR/FSS heuristic (Section 2.3.1) (which uses temporal and resource propagation, but no retraction technique) with the CBASlack heuristic (using the CBA propagator (Section 2.4.1), temporal propagation, and no retraction). The results, on a set of job shop scheduling benchmarks, indicated that equal or better performance is achieved with the CBASlack heuristic. Given equal performance and the relative simplicity of the CBASlack heuristic compared with ORR/FSS, the authors conclude that the more complicated implementation associated with texture measurements is not justified.

Nuijten. The SOLVE scheduling algorithm (Section 3.3.2) consists of LJRand, a set of sophisticated propagators including edge-finding exclusion (Section 2.4.2), and bounded chronological backtracking with restart (Section 2.5.1.2) as the retraction technique. On a set of difficult job shop benchmark problems from the Operations Research literature [Beasley, 1990], SOLVE significantly outperformed both Sadeh's ORR/FSS algorithm (using chronological retraction) and the ORR/FSS heuristic augmented with the propagators used in SOLVE but still using chronological retraction [Nuijten, 1994].

Both of these empirical results appear to be evidence against the use of sophisticated search state analyses as a basis for heuristic commitment techniques. The flaw in both pieces of research, however, is the empirical model of scheduling algorithms. Rather than viewing the algorithms as composed of components (as in the ODO framework) the researchers treat them as monolithic wholes. As a consequence, the heuristic commitment technique is not the only component that is varied among algorithms. In the case of [Smith and Cheng, 1993], the CBA propagator was used as part of the PCP algorithm, but not as part of the ORR/FSS algorithm. Similarly, though [Nuijten, 1994] uses the same propagators in SOLVE and in the augmented ORR/FSS, the retraction techniques are different: SOLVE uses bounded chronological backtracking with restart while ORR/FSS uses chronological backtracking. Given these differences, to what should we attribute the observed experimental results? Is it really the case that the CBASlack heuristic achieves equal performance to ORR/FSS, or is the quality of the PCP algorithm due to the combination of the CBASlack heuristic with the CBA propagator? Does LJRand really significantly outperform ORR/FSS, or does the performance difference arise from the different retraction techniques?

In this chapter, we address these questions by creating a number of instances of the ODO framework that vary only the heuristic commitment technique. In particular, the same set of propagators is used in all experiments and two separate experimental conditions for the retraction techniques are set-up: one uses chronological backtracking for all algorithms while the other uses Limited Discrepancy Search (LDS) (Section 2.5.1.2) for all algorithms.

Before presenting the instantiations of the ODO framework, we review the definition of job shop scheduling and then turn again to texture measurements to present details of the SumHeight heuristic.

4.2 The Job Shop Scheduling Problem

One of the simplest models of scheduling widely studied in the literature is the *job shop scheduling problem*. The classical $n \times m$ job shop scheduling problem is formally defined as follows. Given are a set of n jobs, each composed of m totally ordered activities, and m resources. Each activity A_i requires exclusive use of a single resource R_j for some processing duration dur_i. There are two types of constraints in this problem:

- precedence constraints between two activities in the same job stating that if activity A is before activity B in the total order then activity A must execute before activity B;
- disjunctive resource constraints specifying that no two activities requiring the same resource may execute at the same time.

Jobs have release dates (the time after which the activities in the job may be executed) and due dates (the time by which all activities in the job must finish). In the classical decision problem, the release date of each job is 0, the global due date (or *makespan*) is D, and the goal is to determine whether there is an assignment of a start time to each activity such that the constraints are satisfied and the maximum finish time of all jobs is less than or equal to D. This problem is NP-complete [Garey and Johnson, 1979]. A recent survey of techniques for solving the job shop scheduling problem can be found in [Blazewicz et al., 1996].

4.3 Updating Contention and Reliance: SumHeight

The ORR/FSS heuristic [Sadeh, 1991] (Section 2.3.1) identifies the most critical activity in a search state and assigns it to its most "survivable" start time. The estimation of activity criticality and start time survivability is done on the basis of the underlying texture measurements of contention and reliance. Reliance estimates the extent to which an activity depends on being assigned to a particular resource at a particular time in order to participate in a complete schedule. Contention estimates the competition among the activities for a resource and start time.

Most of the intuitions and texture measurements underlying SumHeight are the same as those underlying ORR/FSS. There are, however, two key differences:

1. SumHeight uses an event-based implementation of texture measurements.
2. Rather than assigning a start time to an activity, SumHeight generates a heuristic commitment to sequence the two most critical activities on the most critical resource.

We describe each of these differences in depth in the rest of this section.

4.3.1 An Event-based Texture Measurement Implementation

Recall that the key component of the contention and reliance texture measurements upon which the ORR/FSS heuristic is based is the calculation of an activity's individual demand, $ID(A, R, t)$. The individual demand is (probabilistically) the amount of resource R, required by activity A, at time t and was calculated using Equations (1) and (2) in Section 2.3.1. For simplicity, we repeat those equations here as Equations (14) and (15). (Recall that STD_A is the start time domain of activity A).

$$ID(A, R, t) = \sum_{t - dur_A < \tau \leq t} \sigma_A(\tau) \tag{14}$$

Where:

$$\sigma_A(\tau) = \begin{cases} \dfrac{1}{|STD_A|} & \tau \in STD_A \\ 0 & otherwise \end{cases} \tag{15}$$

In the straightforward implementation of the individual demand, $ID(A, R, t)$ is calculated for each time point, t, $est_A \leq t < lft_A$, creating the "step" curves displayed in Figure 5 in Chapter 2. The time-complexity of this calculation relies not only on the number of activities and resources, but also on the length of the scheduling horizon. To prevent such scaling, we use an event-based representation and a piece-wise linear estimation of the ID curve. The individual activity demand is represented by four (t, ID) pairs:

$$\left(est, \frac{1}{|STD|}\right), \left(lst, \frac{min(|STD|, dur)}{|STD|}\right), \left(eft, \frac{min(|STD|, dur)}{|STD|}\right), (lft, 0) \tag{16}$$

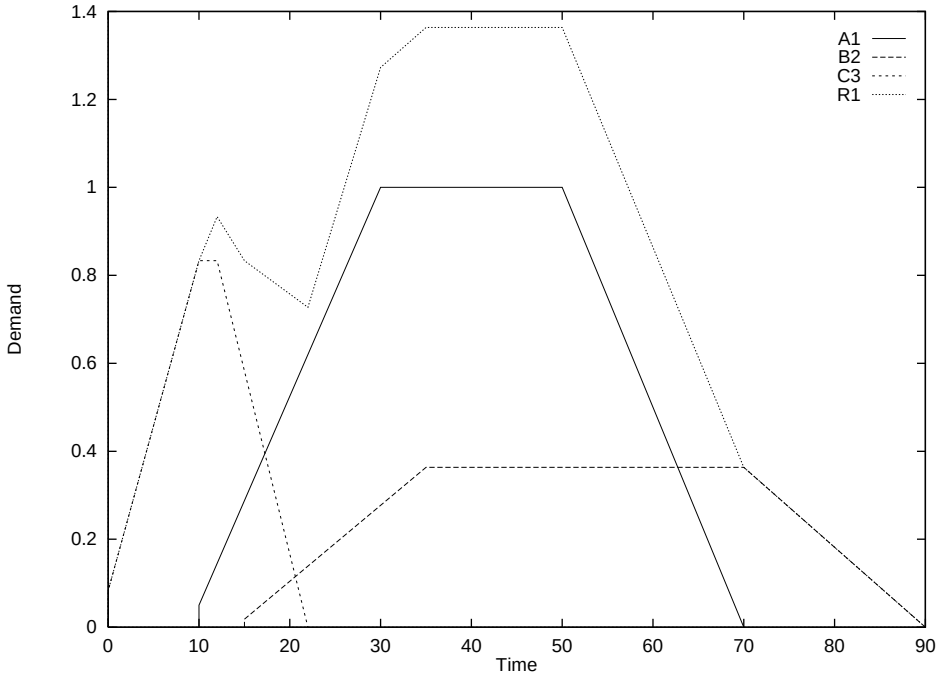


Figure 18. Event-based Individual Demand Curves (A_1, B_2, C_3) and Their Aggregate Curve (R_1).

The individual demand level between any two points on the *ID* curve is interpolated with a linear function.

To estimate contention, the individual demands of each activity are aggregated for each resource. Aggregation is done, as in Sadeh's formulation, by summing the individual activity curves for that resource. This aggregate demand curve is used as a measure of the contention for the resource over time.

Figure 18 shows the individual activity curves and the aggregate resource curve for the activities and resource in Figure 4 of Chapter 2 as implemented for the SumHeight heuristic.

4.3.2 Heuristic Commitment Selection

As in the ORR/FSS heuristic, SumHeight makes a commitment on the activities most reliant on the resource for which there is the highest contention. In more detail, SumHeight does the following:

1. Identifies the resource and time point with the maximum contention.
2. Identifies the two activities, *A* and *B*, which rely most on that resource at that time and that are not already connected by a path of temporal constraints.
3. Analyzes the consequences of each sequence possibility ($A \rightarrow B$ and $B \rightarrow A$) and chooses the one that appears to be superior based on our sequencing heuristics (see Section 4.3.2.2).

The intuition for SumHeight is the same as that for ORR/FSS: by focusing on the most critical resource and activities, we can make a decision that reduces the likelihood of reaching a search state where the resource is over-capacitated. Furthermore, once such critical decisions are made the problem is likely to be decomposed into simpler sub-problems.

4.3.2.1 Finding the Critical Activities

After the aggregate demand curves are calculated for each resource, we identify the resource, *R**, and time point, *t**, for which there is the highest contention (with ties broken arbitrarily). We then examine the activities that contribute individual demand to *R** at *t**. The two critical activities, *A* and *B*, are selected as follows:

- *A* is the activity with highest individual demand for *R** at *t** which is not yet sequenced with all activities executing on *R** (*i.e.*, at least one activity executing on *R** is not yet connected to A via a path of temporal constraints).
- *B* is the activity not yet sequenced with *A* with highest individual demand for *R** at *t**.

Because these two activities contribute the most to the aggregate demand curve, they are the most reliant on the resource at that time.[1]

It can be seen in Figure 18 that one of the critical time points on R_1 is 35. There are only two activities that contribute to this time point, as C_3's latest end time is 22 (see Figure 4 in Chapter 2). Therefore, A_1 and B_2 are selected as the critical activities.

4.3.2.2 Sequencing the Critical Activities

ORR/FSS assigns start times to activities. It identifies the single most reliant activity and determines the start time assignment using the survivability value ordering heuristic.[2] Because we post sequencing constraints between the two most critical activities, rather than assigning start times, we do not calculate survivability or use Sadeh's value ordering. Instead, to determine the sequence of the two most critical activities, we use three heuristics: MinimizeMax, Centroid, and Random. If MinimizeMax predicts that one sequence will be better than the other, we commit to that sequence. If not, we try the Centroid heuristic. If the Centroid heuristic is similarly unable to find a difference between the two choices, we move to Random.

MinimizeMax Sequencing Heuristic. The intuition behind the MinimizeMax (MM) heuristic is that since we are trying to reduce contention, we estimate the worst case increase in contention and then make the commitment that avoids it. MM identifies the commitment that satisfies:

$$MM = min(max_{AD'}(A, B), max_{AD'}(B, A)) \tag{17}$$

Where:

$$max_{AD'}(A, B) = max(AD'(A, A \rightarrow B), AD'(B, A \rightarrow B)) \tag{18}$$

1. The activities with highest individual demand are the most reliant because, in the job shop scheduling problem, there are no resource alternatives. With such alternatives, reliance is not necessarily solely based on individual demand.
2. ORR/FSS also uses a time interval equal to the average activity duration rather than a single time point in identifying the critical resource and activities.

$AD'(A, A \rightarrow B)$ is an estimate of the new aggregate demand at a time point. It is calculated as follows:

- Given $A \rightarrow B$, we calculate the new individual demand curve of A and identify the time point, tp, in the individual demand of activity A that is likely to have the maximum increase in height.[3] This leaves us with a pair: $\{tp, \Delta height\}$.
- We then look at the aggregate demand curve for the resource at tp and form $AD'(A, A \rightarrow B)$ by adding $\Delta height$ to the height of the aggregate demand curve at tp.

The same calculation is done for $AD'(B, A \rightarrow B)$ and the maximum (as shown in Equation (18)) is used in $max_{AD'}(A, B)$. Equation (17) indicates that we choose the commitment resulting in the lowest maximum aggregate curve height.

Centroid Sequencing Heuristic. The centroid of the individual demand curve is the time point that equally divides the area under the curve.[4] We calculate the centroid for each activity and then commit to the sequence that preserves the current ordering (*e.g.*, if the centroid of A is at 15 and that of B is at 20, we post $A \rightarrow B$). Centroid is a variation of a heuristic due to [Muscettola, 1992].

Random Sequencing Heuristic. Randomly choose one of the sequencings.

4.3.3 Complexity

The worst-case time complexity to find a heuristic commitment at a problem state is due to the aggregation of the demand curves for each resource and the selection of the critical activities. By storing the incoming and outgoing slopes of the individual curves at each point, we can sort the event points from all activities and then, with a single pass, generate the aggregate curve. This process has complexity of $O(mn \log n)$ (where n is the maximum number of activities on a resource and m is the number of resources). Selection of the pair of unsequenced activities on the resource requires at worst an additional $O(n^2)$. Thus the overall time complexity for a single heuristic commitment is $O(n^2) + O(mn \log n)$.

The space complexity is $O(mn)$ as we maintain an individual curve for each activity and these individual curves make up the aggregate curve for each resource.

4.4 Instantiations of the ODO Framework

The scheduling algorithms that we evaluate all instantiations of a scheduling strategy of the ODO framework. As our primary purpose in these experiments is to evaluate the efficacy of heuristic commitment techniques, the only difference among the strategies in our experiments is the heuristic commitment technique. Specifically, we hold the set of propagators constant across all experiments and create two experimental conditions based on the retraction technique used.

---

3. Non-local effects resulting from subsequent propagation may have an impact on A's individual demand curve after a heuristic commitment. As a consequence, we do not calculate the actual increase in height of A, but estimate it based on arc consistency of the to-be-posted precedence constraint (*i.e.*, we find A's new *lft* based on local arc consistency propagation and use it to calculate the maximum change in the individual demand).
4. This is a simplification of centroid that is possible because the individual activity curves are symmetric.

Strategy	Heuristic Commitment Technique	Propagators	Retraction Technique
SumHeightChron	SumHeight	All[a]	Chronological backtracking
CBASlackChron	CBASlack	All	Chronological backtracking
LJRandChron	LJRand	All	Chronological backtracking
SumHeightLDS	SumHeight	All	LDS
CBASlackLDS	CBASlack	All	LDS
LJRandLDS	LJRand	All	LDS

a. Temporal propagation, edge-finding exclusion, edge-finding not-First/not-Last, and CBA

Table 3. **Summary of Experimental Scheduling Algorithms.**

Heuristic Commitment Techniques. We use three heuristic commitment techniques in our experiments: SumHeight, CBASlack, and LJRand.

Propagators. Four propagators are used in the following order: temporal propagation (Section 3.3.1), edge-finding exclusion (Section 2.4.2), edge-finding not-first/not-last (Section 2.4.3), and CBA (Section 2.4.1).

Retraction Techniques. We use two retraction techniques in two separate experimental conditions: chronological backtracking (Section 2.5.1.1) and limited discrepancy search (LDS) (Section 2.5.1.2).

Termination Criteria. The termination criteria for all experiments are to find a solution by fully sequencing the activities on each resource, or to exhaust the CPU limit. The CPU time limit for all experiments is 20 CPU minutes on a Sun UltraSparc-IIi, 270 Mhz, 128 M memory, running SunOS 5.6. If an algorithm exhausts the CPU time on a problem, it is counted as a failure to solve the problem.

Table 3 displays a summary of the experimental scheduling strategies.

4.5 Evaluating Scheduling Performance

Given the six instantiations of the ODO scheduling strategy just described, we now want to empirically evaluate these algorithms. While our proximate goals are to investigate the use of texture measurements as a basis for scheduling heuristics and to address the criticisms cited earlier, we have a number of more general goals in conducting an experimental comparison of scheduling algorithms. Included in these goals are:

- An evaluation of the relative usefulness of the scheduling algorithms.
- The ability to generalize results beyond the experimental problem sets.
- Evaluation of differences in the search styles of the algorithms.
- Insight into the reasons for performance differences.

Before discussing the approach used in this dissertation for experimental design and analysis, it is instructive to examine experimental design in the literature.

4.5.1 Competitive versus Scientific Testing

In [Hooker, 1996] the current practice of empirical evaluation of algorithms is termed "competitive testing". This testing consists of running a "track meet" among algorithms, usually on a set of benchmark problems, and using run statistics (often CPU time) to declare a champion among the algorithms tested. This approach is assailed by Hooker as anti-intellectual and counter-productive to the progress of understanding the algorithms in question. The competitive testing paradigm leads not only to researchers spending an inordinate amount of time and resources polishing and optimizing code in order to compete with commercial code but also, given the variety of differences among algorithms, it is difficult to attribute performance differences to any one factor: while competitive testing crowns a champion, no insight into the reasons for the better performance of the algorithm is provided. In particular, the reasons may have more to do with the software engineering abilities of the researchers rather than characteristics of the algorithms tested.

The use of benchmark problems for competitive testing also raises some issues:

- Problem sets become benchmarks because they are solved by some existing algorithm (otherwise they are unlikely to be published) and so existing algorithms have an advantage over new ones.
- Benchmarks exert an evolutionary pressure on subsequent algorithms since if a new algorithm does not perform well on an existing benchmark, it is unlikely to be published.
- It is unclear what problem population benchmarks are representative of, and it is debatable whether a representative set of benchmark problems would even be recognized.

The solution to this dilemma, according to [Hooker, 1996] is to abandon competitive testing in favor of what is referred to as "scientific testing". The point here is controlled experimentation where algorithms are crafted with a single difference and that difference arises out of the desire to test competing hypotheses. The testing then consists of "simulating" the algorithm only to the level of detail necessitated by an algorithmic model. Even though the ultimate goal of the research is a minimized run-time, such data can be estimated from results of the simulation, and furthermore, the simulation provides much deeper insight into *why* an algorithm performs as it does. Problem sets should not be generated randomly or to mirror some "real world" problem, but rather to have varying levels of specific characteristics that are believed to affect performance.

The benefits of scientific testing, it is claimed, are that the algorithms compete on a fair playing field: machine speed, data structure efficiency, coding skill, and parameters are all either irrelevant or specifically part of the algorithmic model.

4.5.2 Empirical Testing in Scheduling

Most, if not all, of the experimental studies in the scheduling literature follow what [Hooker, 1996] describes as the competitive testing paradigm. While frameworks, such as the ODO framework, are designed to allow rigorous empirical testing (by varying a single component of the strategy), there are a number of characteristics of the scheduling domain that make the wholesale adoption of Hooker's "scientific testing" model premature.

The scheduling domain is inherently applied and there are a variety of scheduling problems for which little, if any, research has been done. Given this immaturity, the first goal of research is to

demonstrate that some technology (*e.g.,* constraint-directed scheduling technology) can be of use in some problem domain. This demonstration of usefulness necessarily includes the ability to solve problems in a relatively short amount of time. In other words, CPU time is an important component of evaluation. Until the demonstration of usefulness is made, the question of which problem characteristics can be correlated with the performance of which strategy components is moot. In a new problem domain, the identity of relevant problem characteristics is often unclear. Until we gain experience via less structured experimentation, the investigation of problem characteristics is at best interesting and at worst misguided: much time and energy may be spent on dealing with problem characteristics that are seldom encountered.

This is not to say that the concept of scientific testing is irrelevant to scheduling. Rather, the scheduling research is only beginning to achieve a level of maturity at which such testing can be usefully applied. For example, a number of attempts have been made to investigate difficulty of job shop scheduling problems [Sadeh, 1991; Brasel et al., 1997; Beck and Jackson, 1997]. Furthermore, one of the key motivations for the ODO framework was the ability to rigorously control the scheduling strategies that are tested so that we can attribute any observed performance differences to the component that is varied.

4.5.3 A Compromise Approach

In this dissertation, we have adopted an approach to experimentation that represents a compromise between the competitive and scientific testing paradigms identified by Hooker. While we do perform competitive testing and measure statistics such as CPU time, there are a number of characteristics of our experiments that move toward the scientific testing paradigm.

- We control the scheduling strategies so that only the component of interest is varied.
- We make use of a variety of problem sets: a benchmark set of job shop problems [Beasley, 1990], sets of randomly generated problems, and sets with controlled characteristics (*e.g.,* the number of bottleneck resources).
- We evaluate a variety of measurements. These measurements include CPU time, but also attempt to probe deeper into the search styles of the algorithms. For example, for job shop problems we measure the percentage of total commitments[5] generated by the heuristic commitment technique.

4.5.4 The Reporting of Time-outs

All the experiments in this dissertation make use of a bound on the CPU time. Each algorithm must either find a schedule or prove that no schedule exists for each problem instance. If an algorithm is unable to do so within a limit on the CPU time (in all our experiments in this dissertation the limit is 20 minutes), a time-out is recorded. A time-out indicates that the algorithm was unable to find a solution or prove that no solution exists for a particular scheduling problem instance. Obviously, the fewer problems that an algorithms times-out on, the better the algorithm.

The primary reason for reporting time-out results is that it allows us to use problem sets that contain both soluble and insoluble (over-constrained) problems. The phase transition work in combinatorial problems such as SAT and CSPs [Gent and Walsh, 1994; Gent et al., 1996b] demonstrates that the hardest problem instances are found in locations of the problem space

5. Recall that commitments are generated by both heuristic commitment techniques and propagators.

Source	Problem (lower bound)
[Adams et al., 1988]	abz5(1234), abz6(943)
[Fisher and Thompson, 1963]	ft10(930)
[Lawrence, 1984]	la19(842), la20(902), la21(1046), la24(935), la25(977), la27(1235), la29(1142), la36(1268), la37(1397), la38(1196), la39(1233), la40(1222)
[Applegate and Cook, 1991]	orb01(1059), orb02(888), orb03(1005), orb04(1005), orb05(887)

Table 4. **Test Problems.**

where approximately half of the problems are overconstrained. A “hard instance” is one that cannot be easily shown to be either over-constrained or soluble: significant search effort is required to find a solution or show that none exists. While the space of scheduling problems is not as well-understood as SAT or CSP in terms of phase transition phenomena [Beck and Jackson, 1997], we want to take advantage of this insight in order to generate challenging problem instances. We construct our problem sets so that as the independent variable changes, the problems move from an overconstrained area in the problem space to an under-constrained area. In the former area, proofs of insolubility can often be easily found while in the latter area, solutions can be easily found. It is in the middle range of problems where we expect to find the most difficult problems.

The use of time-outs as a search statistic allows us to integrate search performance on over-constrained problems and soluble problems into a single statistic. The intuition here is that algorithms fail when they can neither find a solution nor a proof of insolubility. By using the number of failures, in this way, we get a clearer picture of the algorithm performance. For example, plotting the number of problems for which a solution is found obscures the fact that some algorithms may be performing very well on over-constrained problems (by finding proofs of insolubility) whereas others are not able to find any such proofs.

4.6 Experiment 1: Operations Research Library

4.6.1 Problem Set

The basic set of problems for the first experiment is a set of 20 job shop scheduling problems (see Table 4) from the Operations Research library of problems [Beasley, 1990]. The set is the union of the problem sets used in [Vaessens et al., 1994] and [Baptiste et al., 1995a] with the exception of one problem (la02) which was removed as it was small and easily solved by all algorithms.

In order to generate a problem set with problems of varying difficulty, we took the set of 20 problems and the known optimal or lower-bound makespan, and created a total of six problem sets by varying the makespan of the problem instances. Recall that, in the classical job shop optimization problem, the goal is to find the minimum makespan within which all activities can be scheduled (see Section 4.2). Rather than adopting an optimization approach, we attempt to satisfy each problem instance at different makespans. This data provides more information on the performance of each algorithm across problems with a range of difficulties. We generate problems with varying makespans by using the *makespan factor*, the primary independent variable in this experiment. It is the factor by which the optimal or lower-bound makespan is multiplied to give the makespan

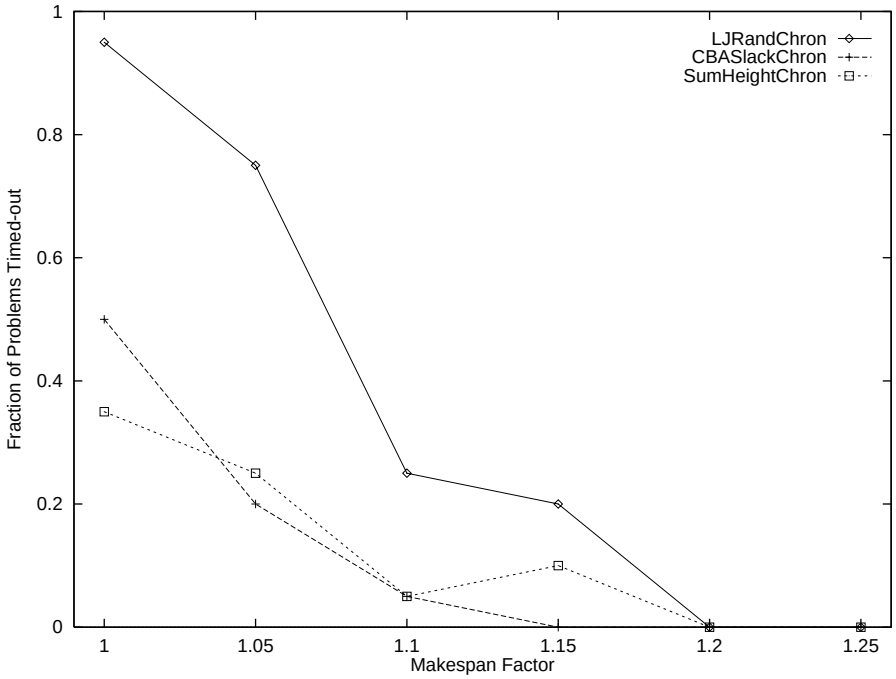


Figure 19. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (Chronological Backtracking).

within which we attempt to solve the problem instances. In this experiment, the makespan factor is varied from 1.0 (the optimal makespan) to 1.25 (25% greater than the optimal makespan) in steps of 0.05, producing six sets of 20 problems each.

4.6.2 Results

Tables of all the results for this experiment can be found in Section B.1 of Appendix B.

4.6.2.1 Problems Timed-Out

Figure 19 displays the fraction of problems in each problem set for which each algorithm (using chronological backtracking) was unable to find a solution. Figure 20 presents the same data for the algorithms using LDS. In terms of the comparison of heuristic commitment techniques, the graphs indicate the same results. Overall, with either retraction technique, LJRand times-out on significantly more problems than either CBASlack or SumHeight, while there is no significant difference between SumHeight and CBASlack.[6] For the individual problem sets, the only significant difference between SumHeight and CBASlack occurs for makespan factor 1.15. For that set CBASlack times-out on significantly fewer problems than SumHeight.

In terms of the comparison between the retraction techniques, overall, the only significant difference was for LJRand which was able to find solutions to significantly more problems when using LDS than when using chronological backtracking. On individual problem sets, LDS significantly outperforms chronological backtracking with SumHeight on the sets with makespan factors 1.05, 1.1, and 1.15, while with CBASlack superior performance of LDS is seen on problem sets 1.05 and 1.1.

6. Unless otherwise noted, statistical tests are performed with the bootstrap paired-t test [Cohen, 1995] with $p \leq 0.0001$.

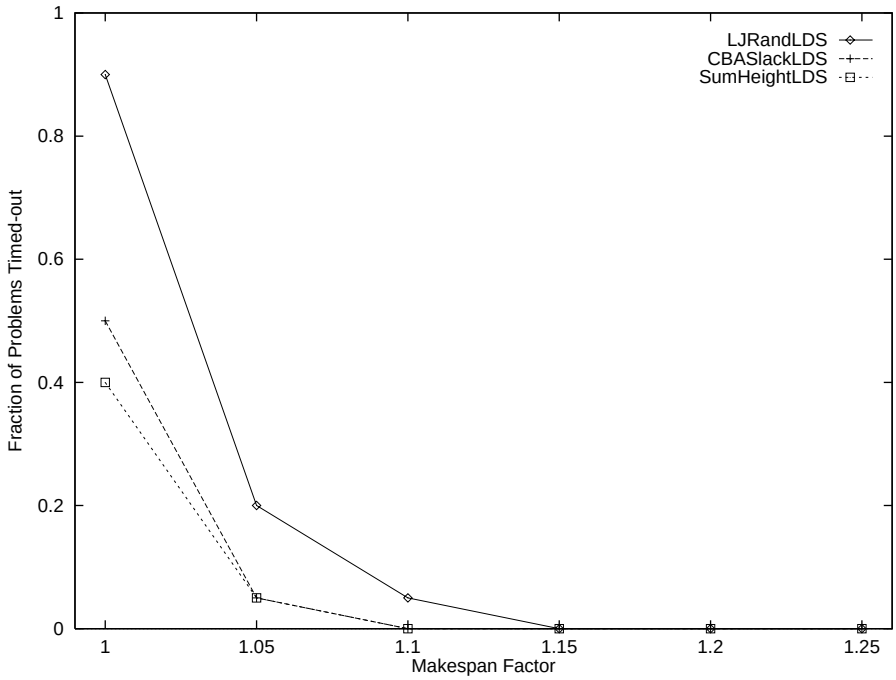


Figure 20. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (LDS).

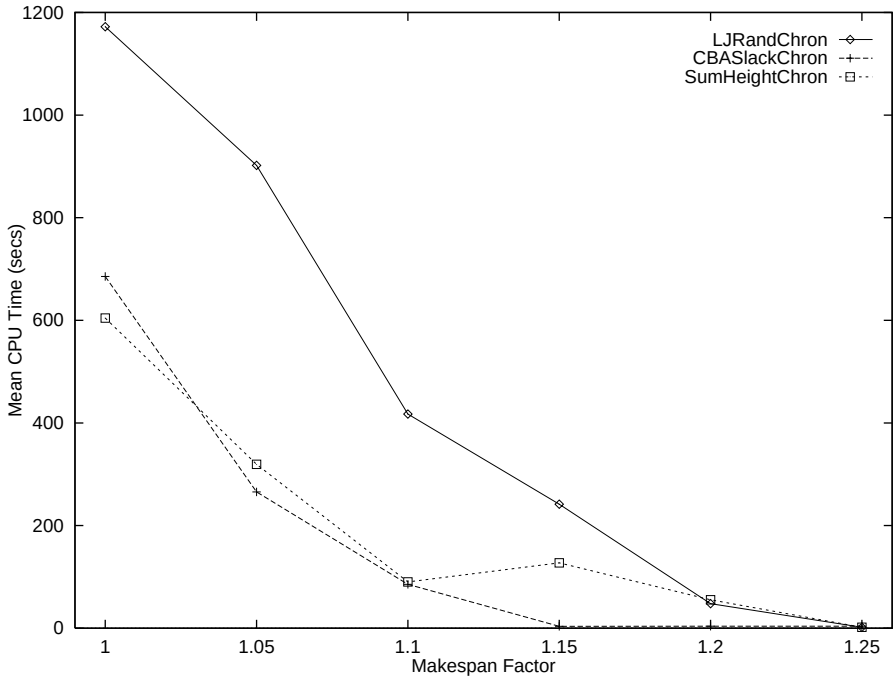


Figure 21. The Mean CPU Time in Seconds for Each Problem Set (Chronological Backtracking).

4.6.2.2 Mean CPU Time

Figure 21 and Figure 22 show the mean CPU time for the chronological backtracking algorithms and the LDS algorithms respectively. As with the number of problems timed-out, overall there is no significant difference between SumHeight and CBASlack while both perform significantly bet-

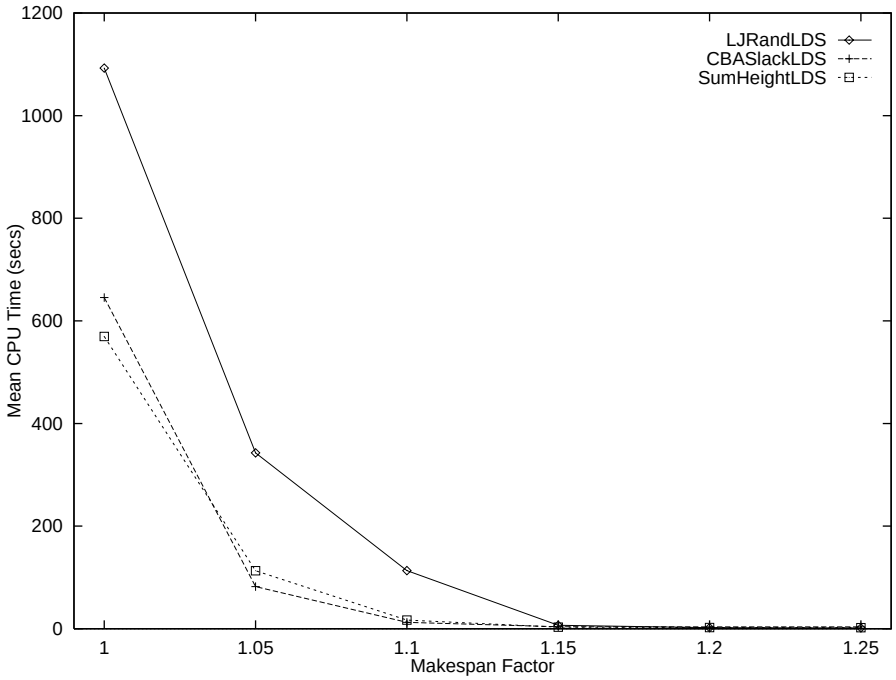


Figure 22. The Mean CPU Time in Seconds for Each Problem Set (LDS).

ter than LJRand. These results hold regardless of retraction technique. On the individual problem sets, there are no significant differences between SumHeight and CBASlack except on problems with makespan factor 1.25 where SumHeight achieves a significantly lower mean CPU time for both retraction techniques. Significantly worse performance by LJRand versus the other two heuristics holds up to the problems with makespan factor 1.1. On problems with larger makespan factors, there is no significant CPU time difference, except on problems with makespan factor 1.25, where LJRand achieves a significantly smaller mean CPU than CBASlack. Again, these results hold with both retraction conditions.

In comparing the retraction techniques, we see significantly larger mean CPU times with chronological retraction than with LDS. The magnitude of the improvement for heuristics indicates that LJRand has a larger improvement in moving to LDS from chronological backtracking than the other two heuristics.

Other measurements of search effort (*i.e.,* mean number of backtracks, mean number of commitments, mean number of heuristic commitments) parallel the CPU time results: no significant differences between SumHeight and CBASlack while both are superior to LJRand. Comparison of the retraction techniques on the other statistics also follow the CPU time results: all algorithms using LDS significantly outperform the corresponding algorithm using chronological backtracking and the magnitude of the improvement appears to be larger for LJRand than for the other two heuristics.

4.6.2.3 Percentage of Heuristic Commitments

Our final search result for Experiment 1 is an attempt to inquire more deeply into the problem solving process of each of these scheduling algorithms. It has been suggested [Baptiste et al., 1995a] that, given the power of propagators in constraint-directed scheduling, a good heuristic is one that makes decisions that result in many subsequent propagated commitments. While the intuition here seems to be primarily pragmatic (*i.e.,* to maximize the contributions from

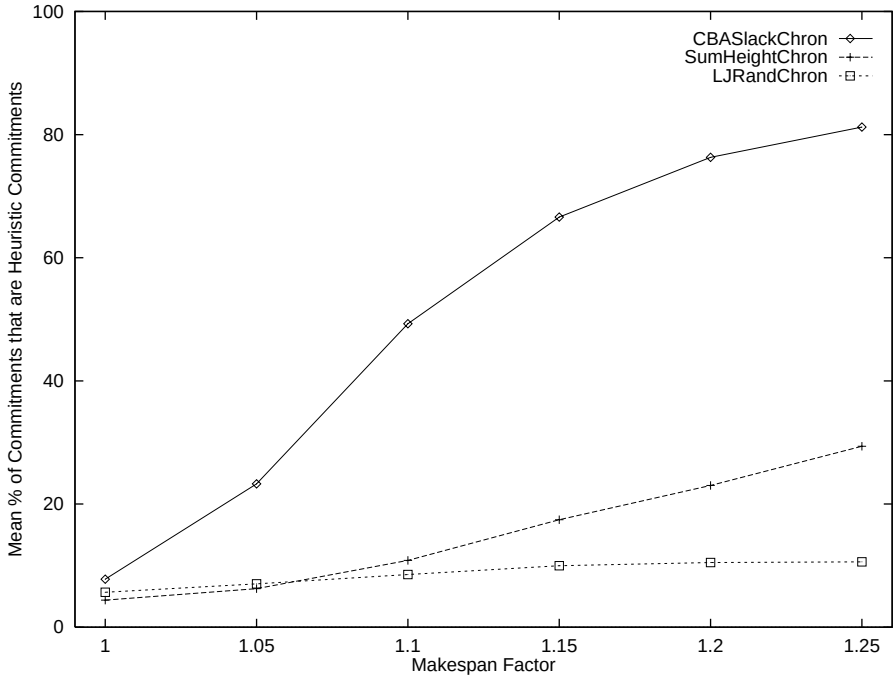


Figure 23. The Mean Percentage of Commitments Made by the Heuristic Commitment Technique (Chronological Backtracking).

sophisticated propagators) there is some more theoretical evidence from SAT heuristics that is consistent with such a suggestion [Hooker and Vinay, 1995]. While further research is necessary to evaluate whether a good heuristic necessarily results in more propagated commitments, it is interesting from the perspective of examining the search behavior to understand the interactions between heuristic commitment techniques and propagators. One measure of this interaction is the percentage of commitments in the search that are made by the heuristic commitment technique: those heuristics that make more use of the propagators will have to make a smaller proportion of heuristic commitments.

The graphs (Figure 23 and Figure 24) display the mean percentage of commitments that are heuristic commitments. Recall that the total set of commitments are composed of heuristic commitments found by the heuristic commitment technique and the implied commitments found by the propagators.[7] The graphs show that LJRand makes a significantly smaller percentage of heuristic commitments than does SumHeight, which in turn makes a significantly smaller proportion of commitments than CBASlack. This holds with both chronological backtracking and LDS.

4.6.3 Summary and Discussion

The basic results from Experiment 1 are that:

- SumHeight and CBASlack are both superior to LJRand while there are no significant differences when SumHeight is compared directly with CBASlack.
- LDS is superior to chronological retraction for all heuristics tested.

7. For this statistic we take into account the implied commitments found by both types of edge-finding and by CBA. We do not count the implied commitments found by temporal propagation as these commitments are typically implemented by removing values from the domains of start time variables rather than by the explicit assertion of a constraint.

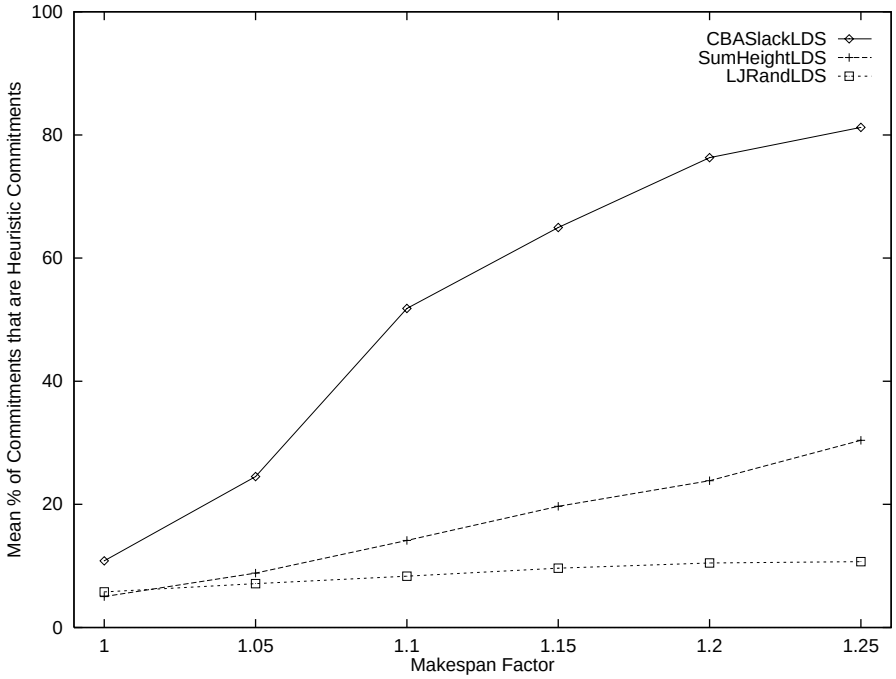


Figure 24. The Mean Percentage of Commitments Made by the Heuristic Commitment Technique (LDS).

The only statistic measured that shows a significant difference among all heuristic techniques is the percentage of commitments that are heuristic. As this is a non-standard statistic, some discussion is in order. The first point to be noted is that LJRand makes a different type of commitment than the other two heuristics. By assigning an activity a start time, LJRand makes a highly constraining commitment from which we might expect a large number of implied commitments to be readily identifiable. This appears to be the case in our experimental results as LJRand demonstrates the lowest percentage of heuristic commitments. Given the difference in heuristic commitment type and the fact that LJRand is significantly worse than the other heuristics on all the problem solving performance measures, it is not very useful to compare the LJRand percentage of heuristic commitments to that of SumHeight and CBASlack. It is interesting, however, to note that the percentage of heuristic commitments for LJRand does not vary much with the makespan factor. This is somewhat surprising: as the makespan factor gets smaller, the problems get tighter and therefore it is expected that the percentage of commitments that are implied should grow.

A comparison of the heuristic commitment percentage between SumHeight and CBASlack is striking. Recall that there were no statistically significant differences in either the number of commitments or the number of heuristic commitments between CBASlack and SumHeight. Nonetheless, SumHeight averaged (non-significantly) more overall commitments and (non-significantly) fewer heuristic commitments than CBASlack. When these results are combined in Figure 23 and Figure 24, significant differences arise. These results show that, as compared with SumHeight, a much larger portion of the CBASlack commitments are heuristic even though exactly the same set of propagators were used in the experiments. We discuss this in Section 4.9.1.3.

4.7 Experiment 2: Scaling with Problem Size

While Experiment 1 used a set of hard instances of job shop scheduling problems, there was no effort to evaluate the performance of the algorithms as the size of the scheduling problems increases. In addition, the use of a variety of problem sets of varying difficulty and characteristics should aid in the differentiation of algorithms [Beck et al., 1997a]. The primary purpose of the second experiment, then, is to evaluate the scaling behaviour of the algorithms.

4.7.1 Problem Set

Four sizes of problems were selected (5×5, 10×10, 15×15, and 20×20) and 20 problems of each size were generated with the Taillard random job shop problem generator [Taillard, 1993].[8] For each problem, a lower bound on the makespan was calculated by running the propagators used for each algorithm. The lower bound is the smallest makespan that the propagators, on their own, could not show was overconstrained. This lower bound calculation is due to [Nuijten, 1994].

Once the lower bounds were calculated, we applied makespan factors from 1.0 to 1.3 in steps of 0.05. For this problem set, however, we only know a lower bound, not the optimal makespan as above. Therefore, the makespan factor is a multiplier of that lower bound, not of the optimal, and we expect most of the problems to be overconstrained at a makespan factor of 1.0. For each size of problem, we have 120 problems divided into six equal sets based on the makespan factor.

4.7.2 Results

All results for Experiment 2 can be found in Appendix B. Tables in the following sections correspond to the various problem sets: overall – Section B.2.1, 5×5 – Section B.2.2, 10×10 – Section B.2.3, 15×15 – Section B.2.4, 20×20 – Section B.2.5.

4.7.2.1 Problems Timed-out

Figure 25 and Figure 26 display the fraction of problems in each problem set that timed-out. Statistically, CBASlack times-out on significantly ($p \leq 0.005$) fewer problems than SumHeight, and SumHeight in turn significantly outperforms LJRand under both the chronological backtracking and LDS conditions. Most of the significance, as indicated by the graphs, is a result of the larger problems. In Figure 27 and Figure 28 we, therefore, display the graphs of the fraction of problems timed-out for the 20×20 problems for chronological backtracking and LDS respectively. Note that CBASlack is significantly better than LJRand in both conditions while CBASlack is only superior to SumHeight using chronological backtracking: the difference in percentage of problems timed-out for LDS is not significant.

Differences between the retraction techniques are similar to the results for Experiment 1. Overall, LDS times-out on significantly fewer problems than chronological retraction when LJRand is used as the heuristic. The overall differences between retraction techniques when using SumHeight or CBASlack as the heuristic are not significant, and, in fact, there are no significant differences between chronological backtracking and LDS on any problem set when CBASlack is used.

8. The duration of each activity is randomly selected, with uniform probability from the domain [1, 100]. Further details, sufficient to generate similar problem sets (varying only based on the random seed) can be found in [Taillard, 1993].

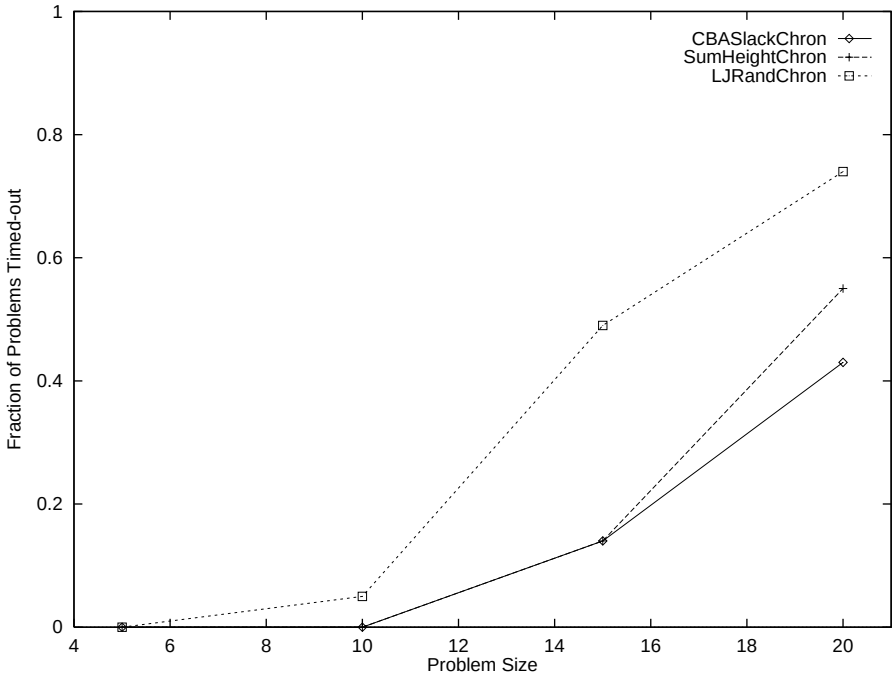


Figure 25. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (Chronological Backtracking).

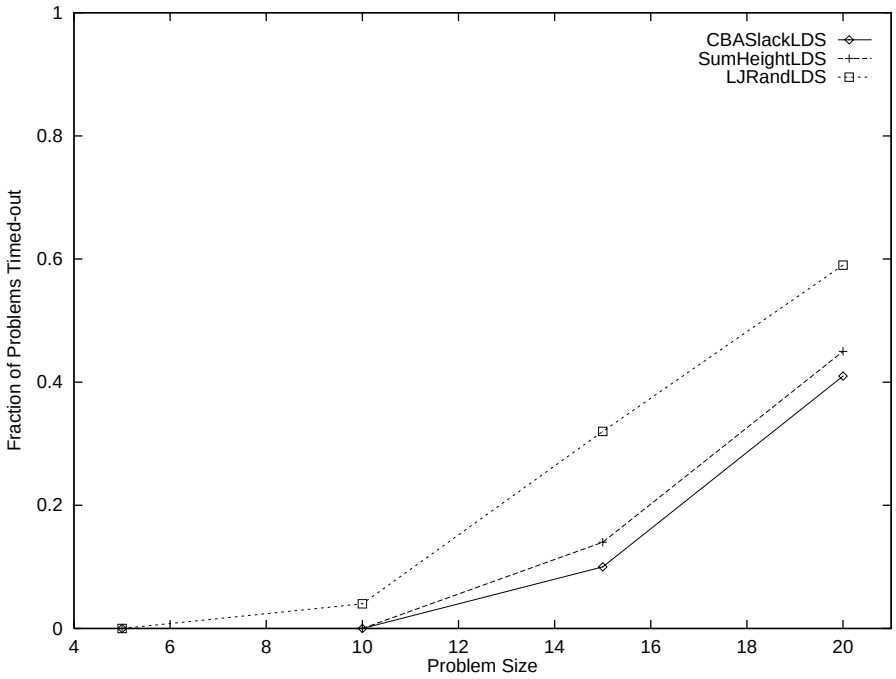


Figure 26. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (LDS).

For the 20×20 problems, LDS significantly outperforms chronological backtracking with Sum-Height and LJRand, while for the 15×15 problems, LDS is significantly better than chronological backtracking only with LJRand. These results confirm the observation from Experiment 1 that LDS appears to improve the weaker heuristics (based on their performance with chronological backtracking) more than the stronger ones.

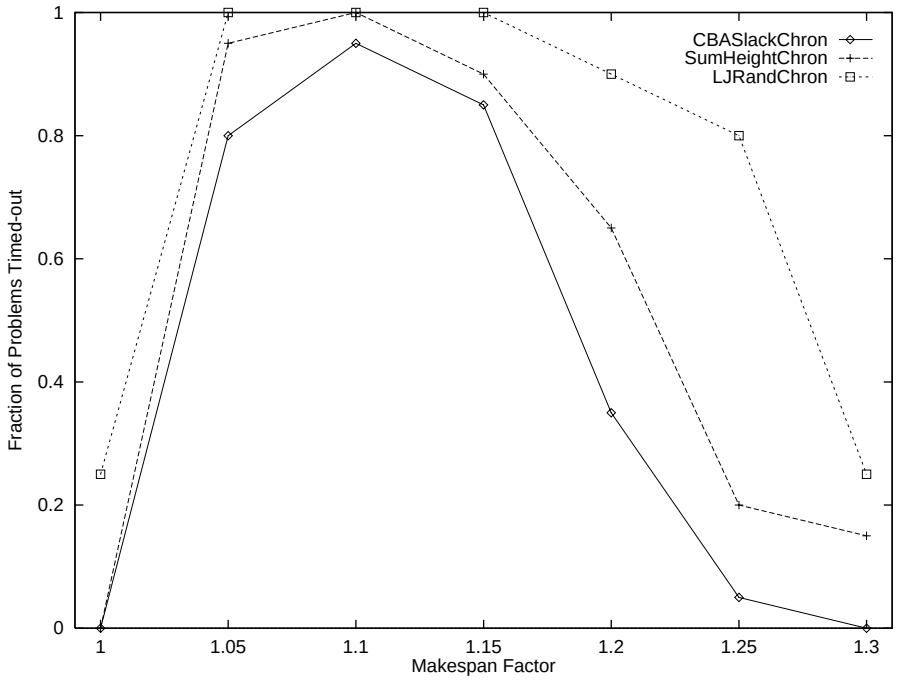


Figure 27. The Fraction of the 20×20 Problems at Each Makespan Factor for which Each Algorithm Timed-out (Chronological Backtracking).

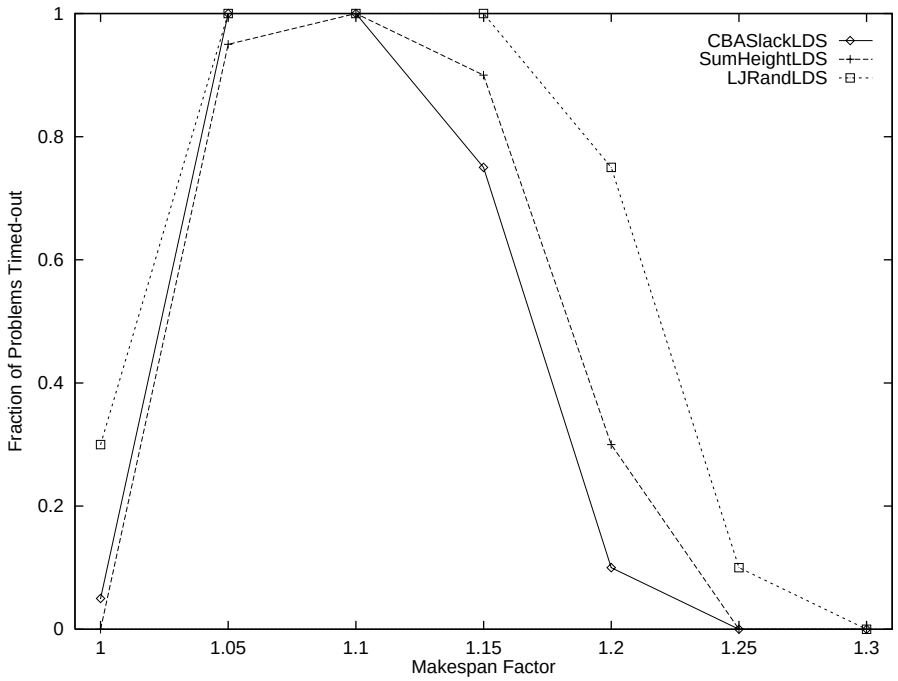


Figure 28. The Fraction of the 20×20 Problems at Each Makespan Factor for which Each Algorithm Timed-out (LDS).

4.7.2.2 Mean CPU Time

The mean CPU time results are shown in Figure 29 for chronological backtracking and Figure 30 for LDS. These graphs are quite similar to the results for the percentage of problems timed-out as the CPU time on the unsolved problems tends to dominate the mean results. Therefore, the overall

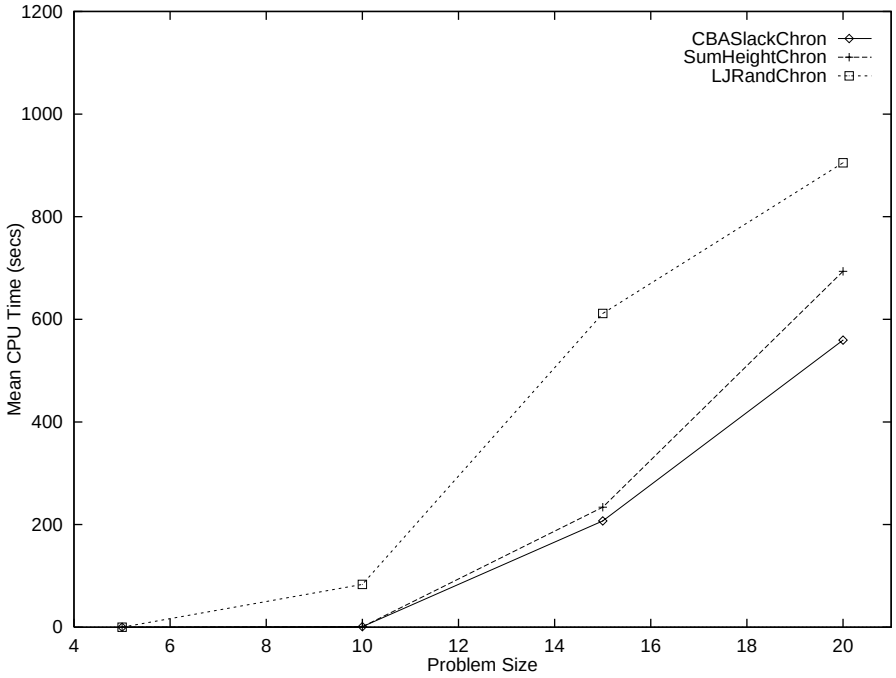


Figure 29. The Mean CPU Time in Seconds for Each Problem Set (Chronological Backtracking).

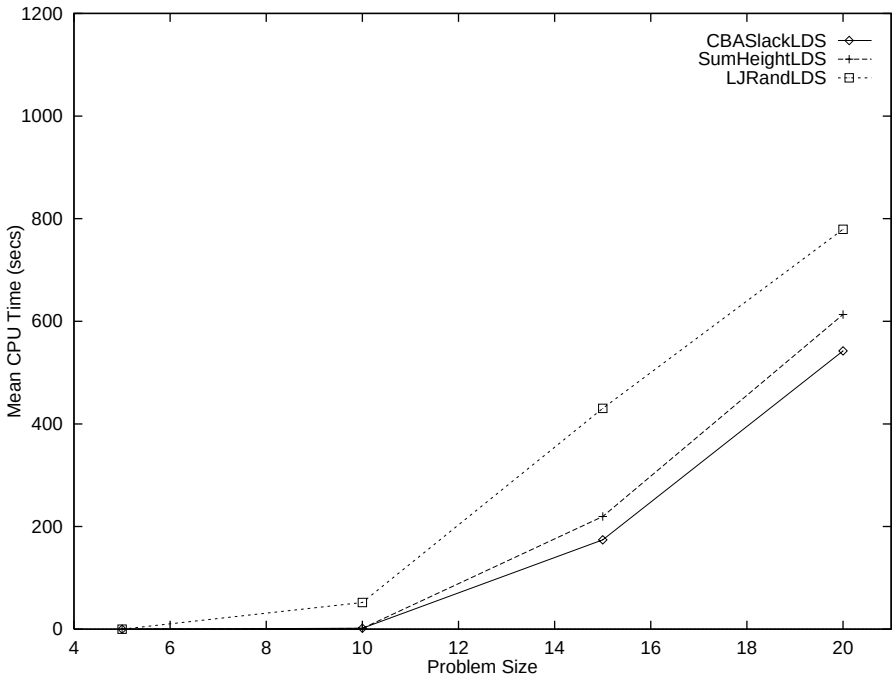


Figure 30. The Mean CPU Time in Seconds for Each Problem Set (LDS).

CPU results tend to mirror the timed-out results: CBASlack achieves significantly lower mean CPU time than SumHeight ($p \leq 0.001$) which in turn achieves a significantly lower mean CPU time than LJRand. In terms of comparing the results for different problem sizes, CBASlack significantly outperforms SumHeight only on the 5×5 and 20×20 problem sets. There is no significant difference at 15×15 and SumHeight achieves a significantly lower mean CPU time than

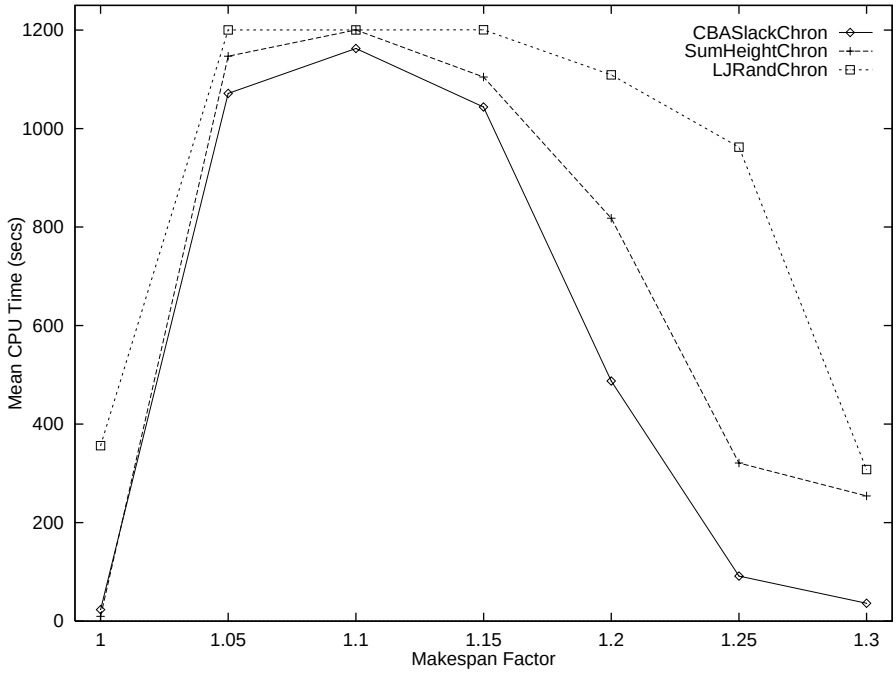


Figure 31. The Mean CPU Time in Seconds for the 20×20 Problems at Each Makespan Factor (Chronological Backtracking).

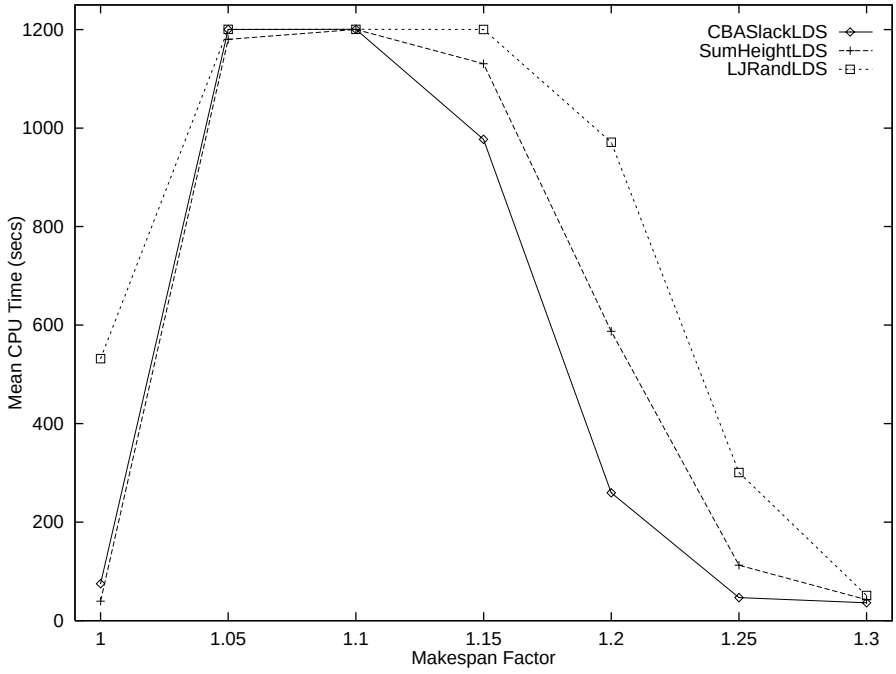


Figure 32. The Mean CPU Time in Seconds for the 20×20 Problems at Each Makespan Factor (LDS).

CBASlack on the 10×10 problems. LJRand is significantly worse than CBASlack for all problem sizes and significantly worse than SumHeight for all sizes except 5×5, where there is no significant difference. Again, since the largest difference among the algorithms comes on the 20×20 problems, we display the mean CPU time results in Figure 31 and Figure 32.

Comparing the retraction techniques, we observe that, overall, LDS with LJRand is significantly better than chronological backtracking with LJRand. The overall differences with the other two heuristics are not significant, though they are in favor of LDS over chronological backtracking. Interestingly, on the 10×10 problems both CBASlack and SumHeight use significantly less CPU time ($p \leq 0.005$) when using chronological backtracking than with LDS. As well, on the 5×5 problems LJRand with chronological backtracking significantly outperforms LJRand with LDS. On the two larger problem sizes, LDS significantly outperforms chronological retraction with LJRand while there is no significant difference with the other two heuristics.

With other search statistics, CBASlack makes significantly fewer commitments, significantly more heuristic commitments, and about the same number of backtracks as SumHeight with both retraction techniques. Both CBASlack and SumHeight significantly outperform LJRand on these statistics. Comparing LDS and chronological backtracking, we see significantly fewer backtracks for LDS, and significantly more commitments and heuristic commitments for LDS (regardless of heuristic used) than for chronological backtracking. As with our other experiments, it appears that the magnitude of the improvement in using LDS rather than chronological backtracking is greater with heuristics that perform worse with chronological backtracking.

4.7.3 Summary

The results of Experiment 2 show that:

- CBASlack significantly outperforms SumHeight which in turn significantly outperforms LJRand.
- LDS significantly outperforms chronological backtracking.

The primary difference between these results and those of Experiment 1 is that we now see differences between CBASlack and SumHeight. These differences, though significant in a number of cases, are not as clear cut as the differences between LJRand and the other two heuristics. On almost every problem set and every performance heuristic, LJRand is significantly worse than SumHeight and CBASlack.

4.8 Experiment 3: Bottleneck Resources

In our final experiment, we follow the recommendations of [Hooker, 1996] in crafting problems with specific characteristics that we believe will affect the performance of the heuristics.

It has long been a practice in industrial scheduling applications to identify the resource for which there is the greatest number of contending activities. This *bottleneck* resource would then be scheduled first before moving to other less utilized resources. Similarly, in the constraint-directed scheduling research, one of the intuitions behind heuristics, such as the contention-based heuristics we have been investigating, is to identify the bottleneck resource and focus the heuristic commitments on it [Adams et al., 1988; Smith et al., 1989; Sadeh, 1991; Muscettola, 1994]. It is generally accepted by industrial schedulers that the existence of bottleneck resources makes scheduling more difficult.[9]

9. Whether this difficulty can be quantified in the spirit of the phase transition research [Hogg and Williams, 1994; Gent et al., 1996b] remains as future work (see however [Beck and Jackson, 1997] for some preliminary efforts in this direction).

4.8.1 Problem Sets

The goal in this experiment is to create problems with specific bottleneck resources, and evaluate the performance of the heuristic commitment techniques and retraction techniques as the number of bottlenecks in a problem increase.

Our starting point is the 10×10, 15×15, and 20×20 problems, all with makespan factor 1.2, used in Experiment 2. Recall that the makespan factor is the factor by which a lower bound on the makespan of each problem is multiplied to generate the makespan of the problem instance. The makespan factor was chosen so that many, and perhaps all, of the problem instances in the starting set is not overconstrained. This final choice was made because by adding bottlenecks we are further constraining the problem instances and we do not want the problems without bottlenecks to already be overconstrained.

The bottlenecks were generated for each problem and each problem set independently by examining the base problem and randomly selecting the bottleneck resources. On each bottleneck resource, we then inserted five new activities of approximately equal duration to reduce the slack time on that resource to 0. The five new activities (on each bottleneck resource) are completely ordered by precedence constraints.

For example, for problem *A* with one bottleneck we may select resource R_2 as the bottleneck. In the base problem the overall makespan is, perhaps, 200 and the sum of the durations of all activities on R_2 is 120. We insert five new activities, each of which executes on R_2 and each with a duration of 16 time units. The sum of the duration of activities on R_2 is 200, the same as the makespan. This has the effect of reducing the slack time on R_2 to 0. To continue the example, when generating the problem set with three bottlenecks, we take problem *A* and randomly select three resources to be bottlenecks. These resources may be R_5, R_8, and R_{10}.[10] For each of these resources five completely ordered activities are added to reduce the slack time on the resource to 0.

With this technique, problem sets are generated for each problem size. For the 10×10 problems, six problem sets, each containing 20 problems, were generated with the number of bottlenecks ranging from 0 to 10 inclusive, in steps of 2. For the 15×15 problems seven problem sets of 20 problems each were generated with the number of bottlenecks ranging from 2 to 14 inclusive, in steps of 2. Finally for the 20×20 problems, six problem sets of 20 problems each were generated with the number of bottlenecks ranging from 0 to 20 inclusive, in steps of 4.

4.8.2 Results

Complete results for the algorithms in Experiment 3 can be found in Appendix B, Section B.3. Tables in the following sections correspond to the various problem sets: 10×10 – Section B.3.1, 15×15 – Section B.3.2, 20×20 – Section B.3.3.

4.8.2.1 10×10 Problems

For the 10×10 problems, the fraction of the problems that each algorithm timed-out on are displayed in Figure 33 for the algorithms using chronological backtracking and in Figure 34 for those algorithms using LDS. Slightly obscured by the plot, in Figure 33, is the fact that Sum-Height does not time-out on any problems. Regardless of the retraction technique used, statistical

10. R_2 may be selected again as a bottleneck. However, R_2 is no more likely to be selected than any other resource.

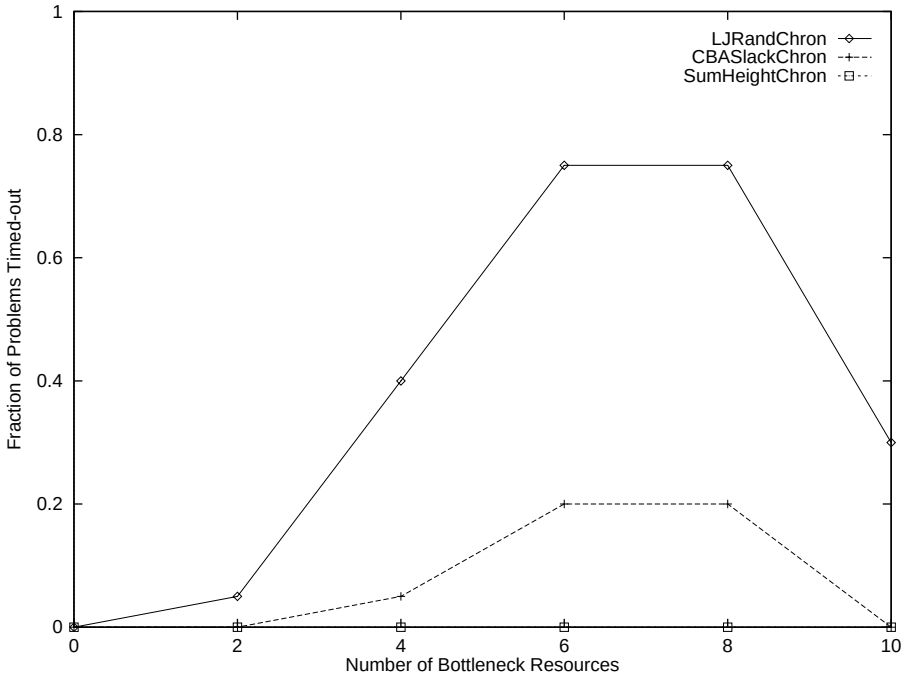


Figure 33. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (10×10 Problems – Chronological Backtracking).

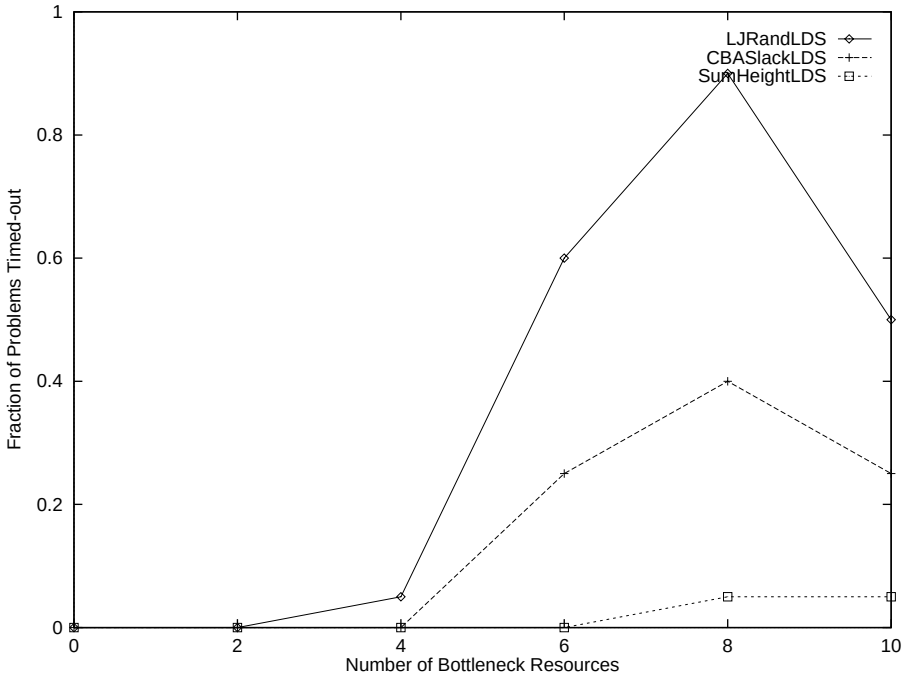


Figure 34. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (10×10 Problems – LDS).

analysis indicates that SumHeight times-out on significantly fewer problems than CBASlack ($p \leq 0.005$ for the comparison using chronological backtracking) and LJRand, while CBASlack in turn times-out on significantly fewer problems than LJRand.

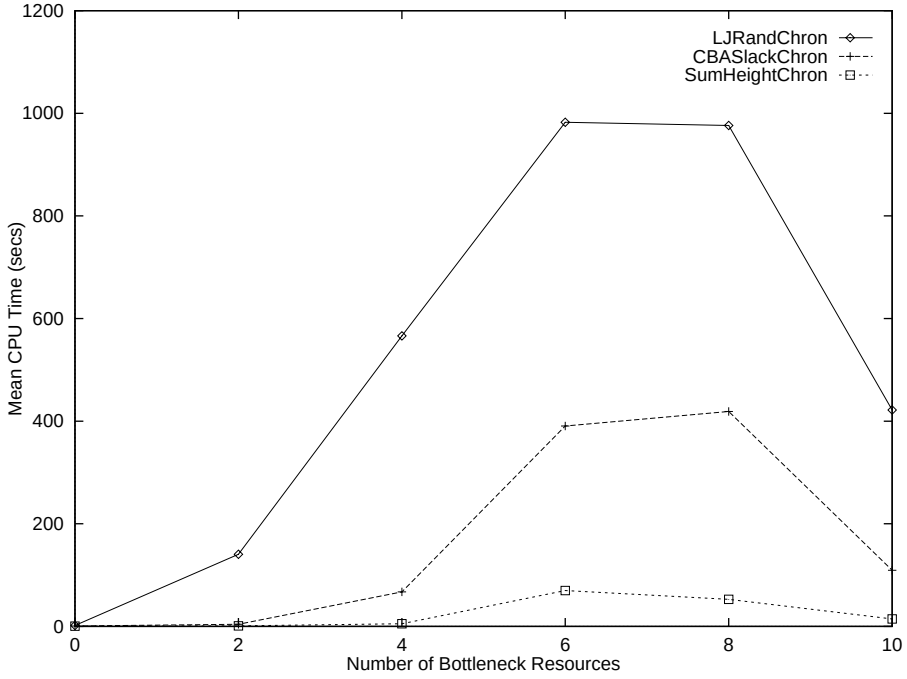


Figure 35. The Mean CPU Time in Seconds for Each Problem Set (10×10 Problems – Chronological Backtracking).

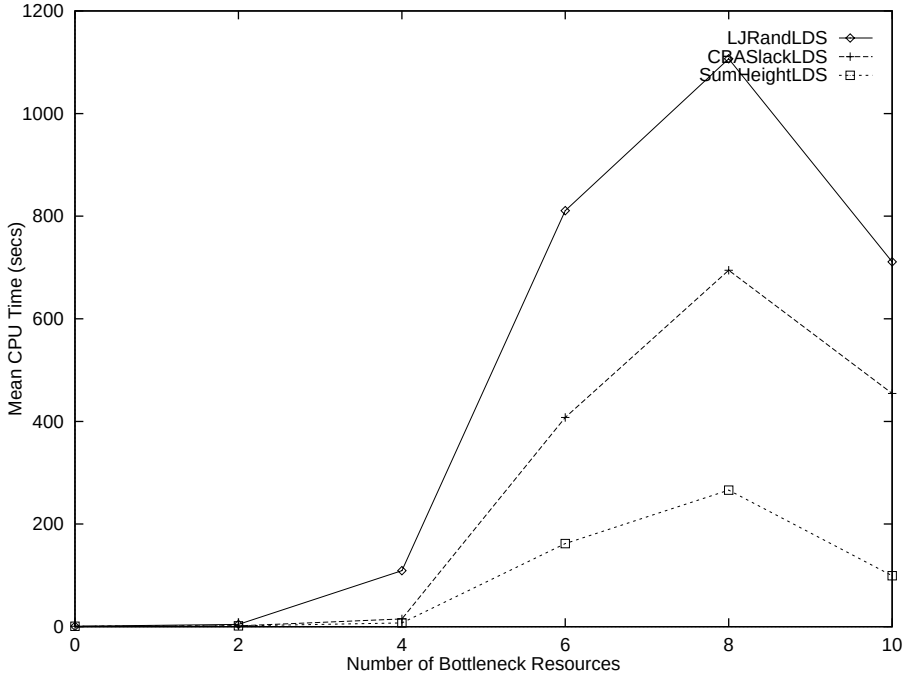


Figure 36. The Mean CPU Time in Seconds for Each Problem Set (10×10 Problems – LDS).

The mean CPU time results are shown in Figure 35 (chronological backtracking) and Figure 36 (LDS). These results are consistent with the timed-out results, as, regardless of retraction technique SumHeight incurs significantly less mean CPU time than either CBASlack or LJRand while CBASlack incurs significantly less mean CPU time than LJRand.

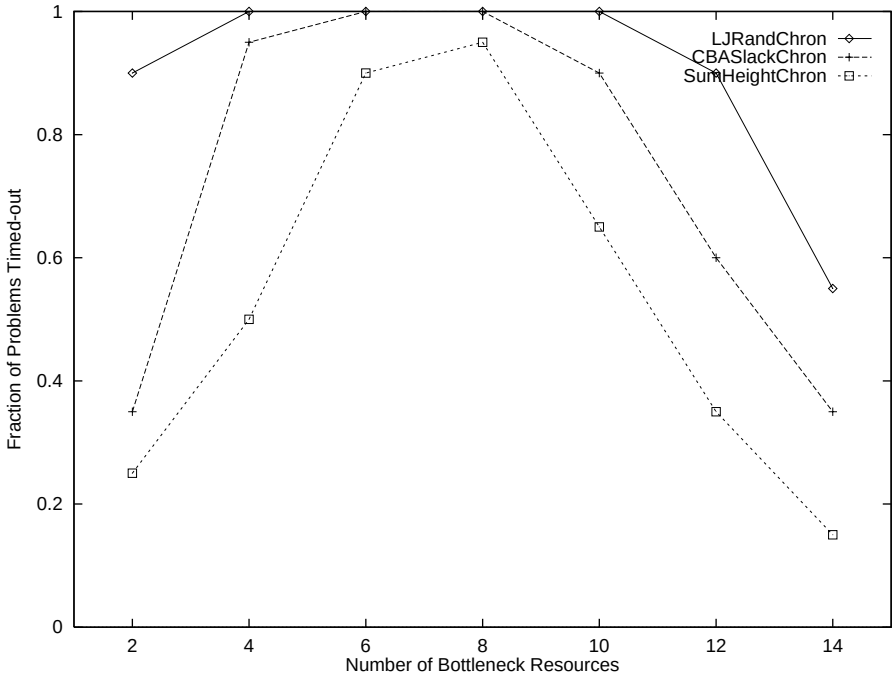


Figure 37. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (15X15 Problems – Chronological Backtracking).

Comparison of the retraction techniques show that there is no significant difference in terms of the number of problems timed-out between the algorithms using chronological backtracking and those using LDS. However, both SumHeight and CBASlack incur a lower mean CPU time when used with chronological backtracking rather than LDS.

Results with other search statistics are as follows:

- For the algorithms using chronological backtracking, all statistics tested (the number of backtracks, the total number of commitments, and the number of heuristic commitments) indicate that SumHeight significantly outperforms CBASlack and LJRand. CBASlack in turn significantly outperforms LJRand. These results are repeated when LDS is used as the retraction technique except in the case of the number of backtracks. For that statistic, while SumHeight is significantly better than all other heuristics, there is no significant difference between CBASlack and LJRand.
- Comparison of the retraction techniques shows that, for the number of backtracks, SumHeight incurs significantly fewer when used with chronological backtracking than with LDS, CBASlack shows no significant differences, and LJRand incurs significantly fewer backtracks with LDS than with chronological backtracking. For both the overall number of commitments and the number of heuristic commitments, SumHeight and CBASlack make significantly fewer with chronological backtracking while there is no significant difference for LJRand.

4.8.2.2 15X15 Problems

The fraction of problems in each problem set for which each algorithm was unable to find a solution (or show that the problem was over-constrained) is shown in Figure 37 for those algorithms using chronological backtracking and in Figure 38 for those algorithms using LDS.

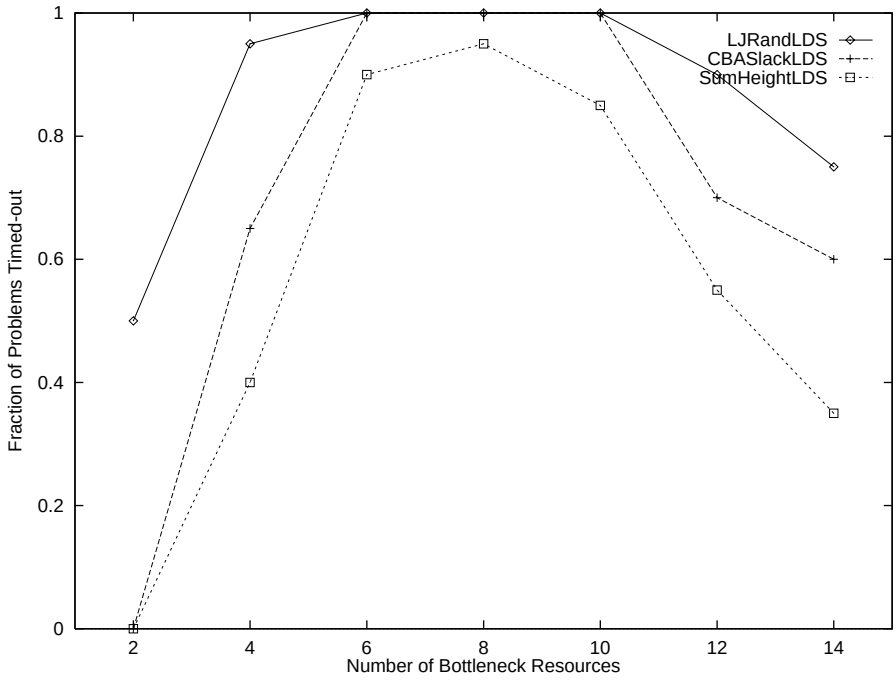


Figure 38. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (15×15 Problems – LDS).

Statistical analysis mirrors the informal impression of these graphs: SumHeight times-out on significantly fewer problems than CBASlack which, in turn, times-out on significantly fewer problems than LJRand. These results hold in both retraction component conditions.

Comparing the retraction techniques while holding the heuristic commitment technique constant shows no overall statistically significant differences between chronological backtracking and LDS. For the problem sets with a low number of bottlenecks (two and four), we see that LDS solves significantly more problems than chronological backtracking for each heuristic (with the exception of SumHeight on the problems with four bottlenecks where there is no significant difference). For problem sets with higher number of bottlenecks (*e.g.,* 12 and 14) we see the reverse: the algorithms with chronological backtracking time-out on fewer problems; however, these differences are not statistically significant.

The results for the mean CPU time for each problem set are displayed in Figure 39 (chronological backtracking) and Figure 40 (LDS).

These results reflect the timed-out results: with either retraction technique SumHeight incurs significantly less mean CPU time than CBASlack which in turn incurs significantly less mean CPU time than LJRand.

Comparison of retraction techniques shows no overall significant differences in mean CPU time. Again, however, the pattern of chronological retraction being inferior at low bottlenecks and superior at high bottlenecks is observed. Chronological backtracking with both CBASlack and LJRand on the problem set with two bottlenecks incurs significantly more CPU time than the corresponding algorithms using LDS. For problem sets with 12 and 14 bottlenecks, chronological backtracking with both SumHeight and CBASlack incurs significantly less CPU time than the corresponding algorithm with LDS.

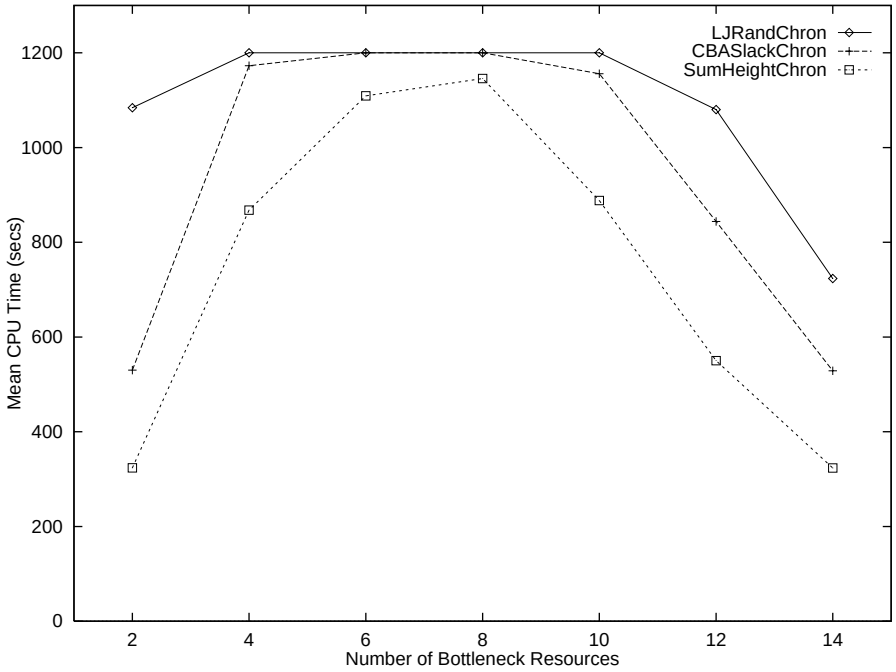


Figure 39. The Mean CPU Time in Seconds for Each Problem Set (15X15 Problems – Chronological Backtracking).

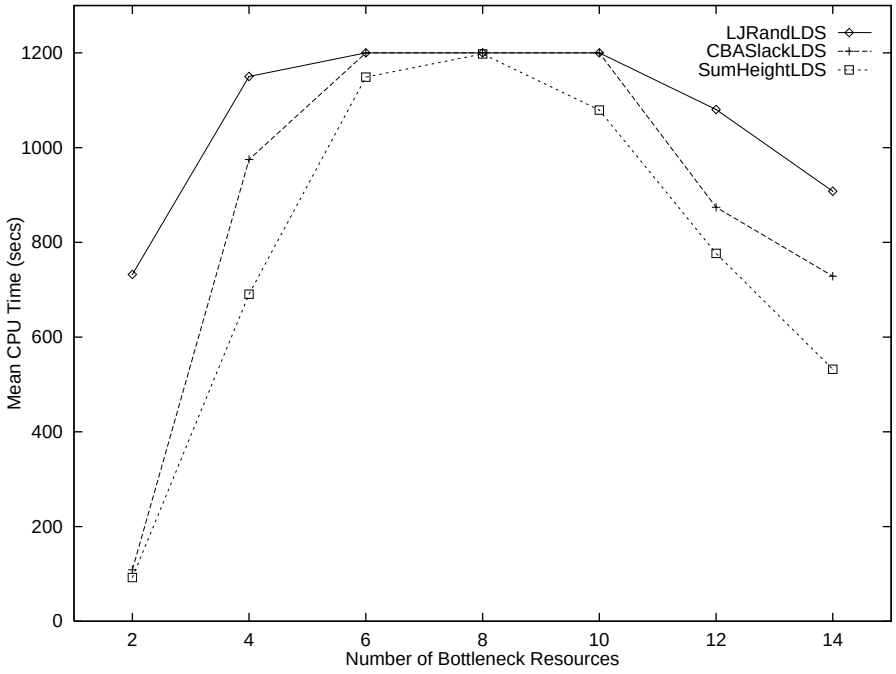


Figure 40. The Mean CPU Time in Seconds for Each Problem Set (15X15 Problems – LDS).

Results with other search statistics are as follows:

- With chronological backtracking, SumHeight makes significantly fewer backtracks than either LJRand or CBASlack. There is no significant difference between CBASlack and LJRand. With

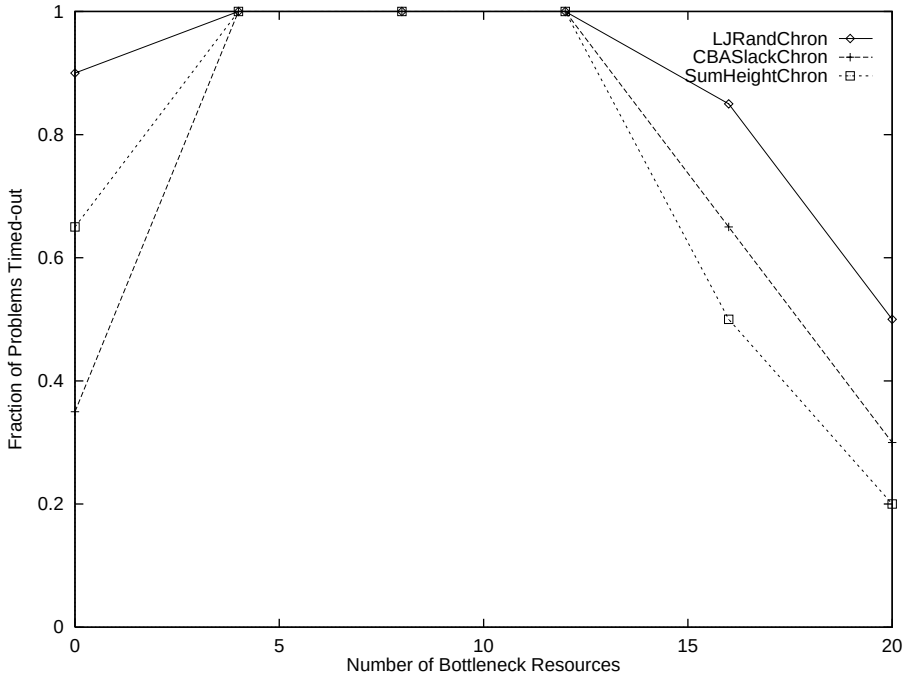


Figure 41. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (20×20 Problems – Chronological Backtracking).

LDS, SumHeight uses significantly more backtracks than CBASlack; however, LJRand incurs fewer backtracks than both SumHeight ($p \leq 0.005$) and CBASlack.

- In terms of the number of commitments, there is no significant difference between SumHeight and CBASlack (in either retraction condition) while LJRand makes significantly more commitments than the other algorithms, regardless of retraction technique.
- SumHeight makes significantly fewer heuristic commitments than either of the other algorithms with both LDS and chronological backtracking. CBASlack makes significantly more heuristic commitments than LJRand using LDS.
- Finally, in the comparison of the retraction techniques we see that the algorithms using chronological backtracking make significantly more backtracks, significantly fewer commitments, and significantly fewer heuristic commitments than the corresponding algorithms using LDS. These results hold regardless of the heuristic.

It may seem inconsistent that SumHeight makes fewer backtracks and heuristic commitments than CBASlack but not significantly fewer overall commitments. This anomaly can be understood based on the fact that, for SumHeight, the percentage of the total commitments that are heuristic commitments is significantly smaller than for CBASlack. While CBASlack backtracks more and makes more heuristic commitments, because it makes fewer propagated commitments for each heuristic commitment, the overall number of commitments is not significantly different from that of SumHeight. Given the differing computational expense of heuristic commitments, propagated commitments, and backtracking, the existence of such a comparison is one reason we use mean CPU time as one of our primary search statistics.

4.8.2.3 20×20 Problems

The fraction of problems timed-out for each algorithm on the 20×20 problems are shown in Figure 41 and Figure 42 for chronological backtracking and LDS respectively. Regardless of the

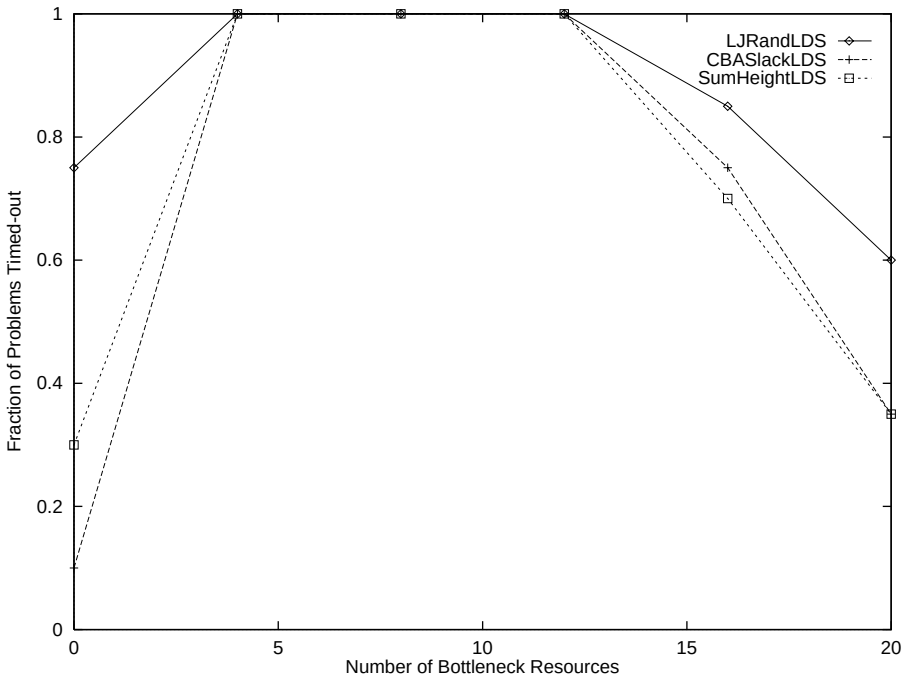


Figure 42. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (20×20 Problems – LDS).

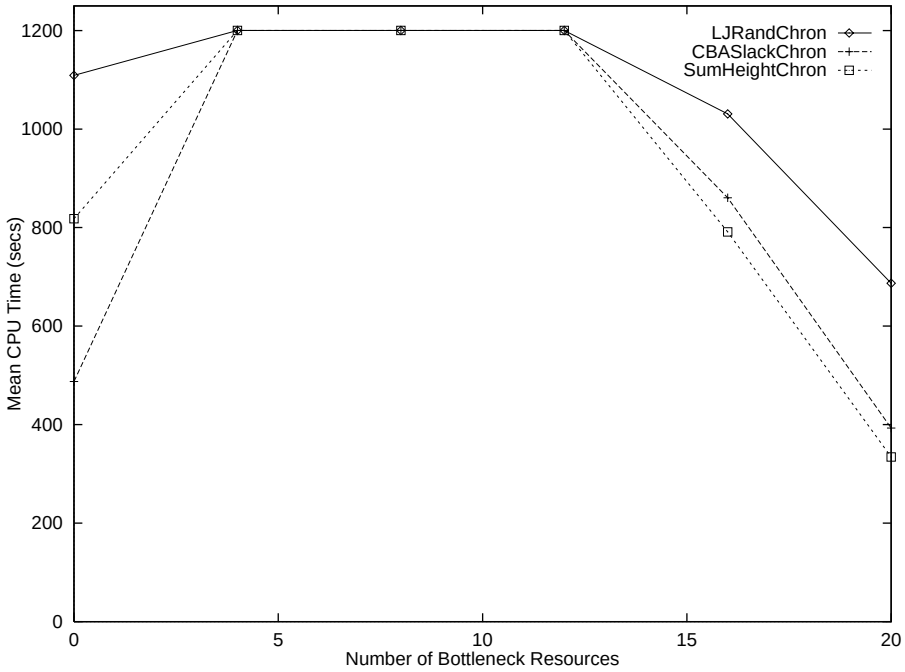


Figure 43. The Mean CPU Time in Seconds for Each Problem Set (20×20 Problems – Chronological Backtracking).

retraction condition, statistical analysis indicates no significant difference between SumHeight and CBASlack while both time-out on significantly fewer problems than LJRand.

The mean CPU time results are shown in Figure 43 for chronological backtracking and Figure 44 for LDS. As with the timed-out results, the mean CPU time results indicate that there is no significant differences between SumHeight and CBASlack, regardless of retraction technique, while

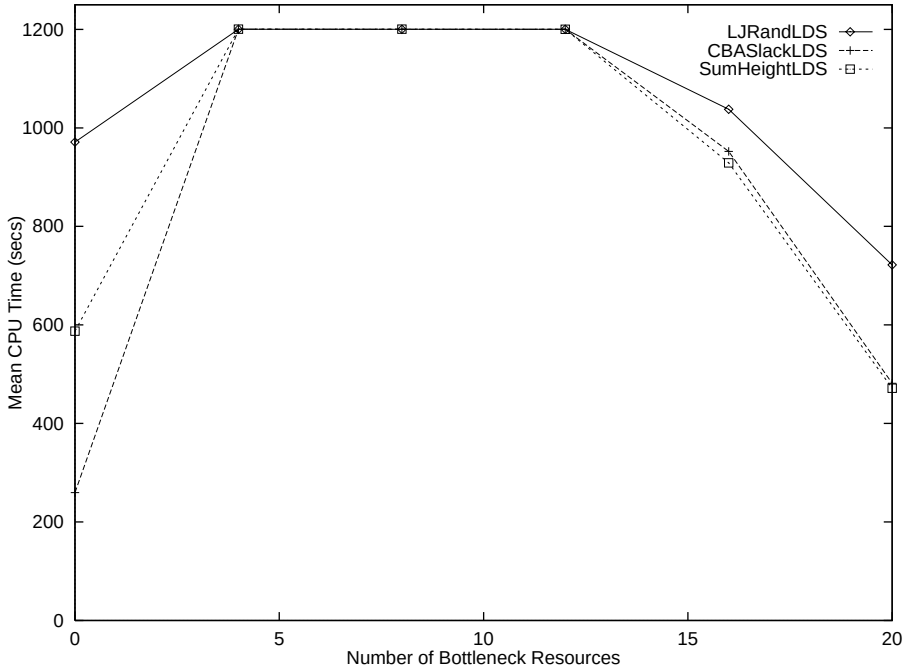


Figure 44. The Mean CPU Time in Seconds for Each Problem Set (20X20 Problems – LDS).

both SumHeight and CBASlack incur significantly less mean CPU time than LJRand, again, regardless of retraction technique.

In comparing the retraction techniques themselves, we see that there are no significant differences between the chronological backtracking algorithms and the LDS algorithms in terms of the fraction of problems timed-out or the mean CPU time. This result holds for each heuristic commitment technique.

Results with other search statistics are as follows:

- In terms of the number of backtracks, there are no significant differences among the heuristics except when LDS is used as the retraction technique. In that condition, SumHeight and CBASlack both incur significantly more backtracks than LJRand while there is no significant difference between SumHeight and CBASlack.
- Regardless of the retraction technique used, CBASlack makes significantly fewer overall commitments than SumHeight or LJRand while SumHeight makes significantly fewer than LJRand.
- For the number of heuristic commitments, SumHeight makes significantly fewer than all other heuristics using either chronological backtracking or LDS. Similarly, LJRand makes significantly fewer heuristic commitments than CBASlack regardless of retraction technique used.
- Finally, in comparing the retraction techniques, we see that algorithms using chronological backtracking make significantly more backtracks, significantly fewer commitments, and significantly fewer heuristic commitments than corresponding algorithms using LDS.

4.8.3 Summary

The results from Experiment 3 indicate that:

- With the exception of the 20×20 problems, SumHeight outperforms CBASlack and LJRand while CBASlack in turn outperforms LJRand. On the 20×20 problems there was little difference between SumHeight and CBASlack while both outperformed LJRand.
- The comparison of chronological backtracking with LDS is less clear. With the exception of the 10×10 problems, there is no difference in terms of CPU time or the fraction of problems timed-out. On the 10×10 problems, chronological backtracking is able to achieve a lower mean CPU time when used with SumHeight or CBASlack. The other statistics show that, typically, chronological backtracking incurs more backtracks than LDS but fewer overall commitments and fewer heuristic commitments. An exception is observed in the 10×10 results where SumHeight with chronological backtracking incurs significantly fewer backtracks than SumHeight with LDS.

4.9 Discussion

Two goals motivated the three experiments performed in this chapter. The primary goal was a comparison of heuristic commitment techniques under a rigorous experimental model. The secondary goal was an investigation of the differences between the chronological backtracking and LDS retraction techniques. In this section, we look at each of these in turn.

4.9.1 Heuristics

4.9.1.1 LJRand

The most obvious result from the experiments in this chapter is that LJRand does not perform as well as either of the other two heuristics. This comparison holds regardless of the retraction technique and across all the experiments. This result contradicts work by [Nuijten, 1994] who showed that LJRand was able to outperform the ORR/FSS heuristic on the Operations Research library problems (a subset of which were used in Experiment 1). As noted above, however, for the experiments performed by Nuijten, LJRand was run with a different retraction technique (chronological backtracking with restart (see Section 2.5.1.2)) than the ORR/FSS heuristic was (chronological retraction). Based on the experiments here, we attribute Nuijten's results to the retraction techniques rather than the heuristics.[11]

4.9.1.2 CBASlack and SumHeight

The comparison between SumHeight and CBASlack is more complicated as we see no significant difference in Experiment 1, while CBASlack performs better in Experiment 2 and SumHeight performs better in Experiment 3.

---

11. The ORR/FSS heuristic does not have a random component; therefore, it cannot be directly used with a retraction technique like chronological backtracking with restart which depends on such a component to explore different paths in the search tree. Schemes for adaptation of heuristics such as SumHeight and CBASlack to include a random component have been shown to result in gains in overall search performance [Oddi and Smith, 1997] [Gomes et al., 1998].

Recall (Section 4.1.1) that SumHeight and CBASlack are different methods of measuring similar underlying characteristics of a constraint graph: the contention for a resource. Where SumHeight measures this contention by aggregating probabilistic demand of each activity, CBASlack identifies the pair of activities that compete most with each other.

The results of our experiments can be understood based on the relative sensitivity of the two heuristics to non-uniformity at the resource level. To understand this concept, it is helpful to examine the properties of our problem sets. In Experiment 2, the duration of each activity and the sequence of resources used by each job were randomly generated. It is reasonable to suppose, therefore, that there is little difference in resource utilization: these problems should have a uniform resource utilization. In Experiment 3, we augmented the randomly generated problems by selecting a subset of resources and adding activities such that their utilization was 100%. A scheduling problem with a wide ranging resource utilization is said to have a non-uniform resource utilization.

Imagine two scheduling problems, $P_{uniform}$ and $P_{non\text{-}uniform}$, with a uniform and a non-uniform resource utilization respectively. Assume that $P_{non\text{-}uniform}$ was generated from $P_{uniform}$ by the method used to generate bottleneck problems. $P_{non\text{-}uniform}$ has a superset of the activities in $P_{uniform}$.

The CBASlack calculations (before any commitments have been made) (see Section 2.3.4) on $P_{uniform}$ find the biased-slack for each pair of activities. The calculation depends wholly upon the time-windows of each activity which in turn depend on the precedence constraints within a job: there are, as yet, no inter-job constraints. When the same calculations are done on $P_{non\text{-}uniform}$ the pairs of activities that the problems have in common will have exactly the same biased-slack values. CBASlack only estimates the contention between pairs of activities and so it will calculate the same biased-slack value for a pair of activities regardless of the other activities contending for the same resource. Therefore, the biased-slack value of two activities in $P_{uniform}$ is the same as for those same two activities in $P_{non\text{-}uniform}$. Gradually, as commitments are made, the activities on a bottleneck resource will tend to have lower biased-slack values; however, for the first few commitments, CBASlack does not detect the non-uniformity. As has been noted elsewhere [Harvey, 1995], the first few commitments are often critical to finding a solution. SumHeight, in contrast, immediately focuses on one of the bottlenecks as it aggregates probabilistic demand information from all activities on the resource. Assuming that it is truly more important to make decisions on tighter resources, the ability to focus on bottleneck resources, *when they exist*, is an explanation of why SumHeight is able to outperform CBASlack on non-uniform problems as in Experiment 3.

When problems have a uniform resource utilization, however, no bottleneck resources exist. SumHeight identifies a critical resource and time point; however, it is likely that other resources and time points have criticality measurements that are close (or even equal) to the one selected. Only the activities on the “critical” resource are then examined in order to post a commitment. In contrast, CBASlack looks at all pairs of activities. It may well be the case that though the resource utilization is uniform, there are pairs of activities that have a very low biased-slack value. For example, imagine a pair of activities such as *A* and *B* in Figure 45. The wide time windows on each activity lead to a relatively low contention as measured by SumHeight, while CBASlack will identify these activities as critical. While SumHeight may identify some other resource as critical and make a commitment, CBASlack will focus on a critical pair regardless of the resource utilization and, therefore, it is likely to make a commitment at a more critical area of the search space than SumHeight.

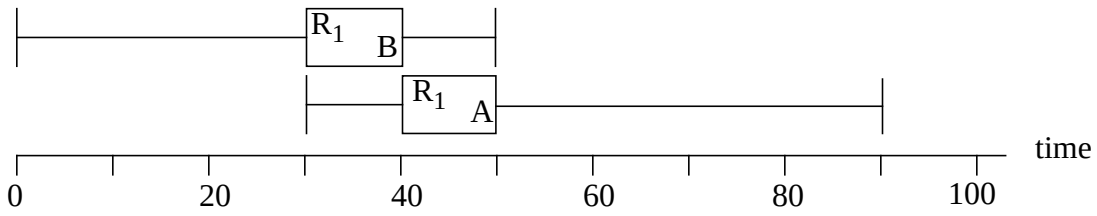


Figure 45. Activities A and B.

An underlying assumption in this explanation is that when a resource-level non-uniformity exists, it is more important to make a commitment on activities that execute on a highly contended-for resource even though there may be an activity pair with a lower biased-slack value on another resource. While we believe this assumption to be reasonable, at this time we have no empirical evidence to support it (without begging the question we are trying to explain).

To lend support to this explanation, we examined the resource uniformity of the problems in Experiment 2 and Experiment 3. The resource usage of a resource, *R*, is the fraction of the total scheduling horizon during which a resource is used by some activity. We represent the resource usage by *RU*(*R*) which can be found by summing the durations of the activities on *R* and dividing by the length of the scheduling horizon. Given *RU*(*R*) for each resource in a problem, *RU*(*P*) is defined to be the mean resource usage across all resource in problem, *P*. Assuming that *RES*(*P*) denotes the set of resources in problem *P*, the standard deviation for the resource usage in a problem, $\sigma(RU(P))$, can be calculated as shown in Equation (19).

$$\sigma(RU(P)) = \sqrt{\frac{\sum_{R \in RES(P)} (RU(R) - RU(P))^2}{|RES(P)| - 1}} \tag{19}$$

The standard deviation of resource usage in a problem is a measure of its resource non-uniformity. The higher the standard deviation, the more varied is the resource usage in the problem.

Figure 46 plots the difference in CPU time between SumHeight and CBASlack (both using chronological backtracking) against the standard deviation of resource usage in each problem. Points above zero on the y-axis indicate problems where SumHeight incurred greater CPU time than CBASlack, while those below zero indicate problems where SumHeight incurred less CPU time. The horizontal axis is the standard deviation of the percent resource utilization for each problem. A larger standard deviation indicates a larger difference in utilization among resources in a problem and therefore a greater resource-level non-uniformity.

Figure 46 indicates that on problems with more resource-level non-uniformity, SumHeight tended to outperform CBASlack. The reverse tends to be true with problems with more uniform resource usage. These results do not prove that the difference in resource-level uniformity causes the performance difference. The results do, however, indicate a correlation which we see as lending credence to the resource-level uniformity explanation.

Uniformity and Non-uniformity in Problem Structure

Our explanation of the results of SumHeight and CBASlack is a single instance of a broader concept: a productive tactic of a heuristic commitment technique is the exploitation of non-uniformity

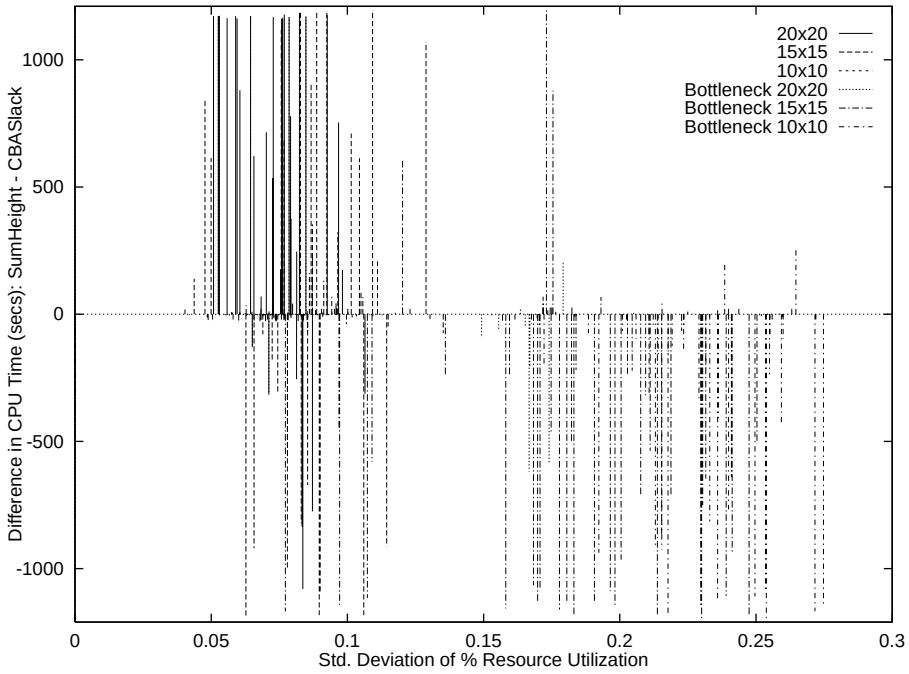


Figure 46. The Standard Deviation in Resource Usage for Each Problem in the 10X10, 15X15, and 20X20 Problem Sets of Experiment 2 and Experiment 3 versus the Difference in CPU Time in Seconds between SumHeight and CBASlack (Chronological Backtracking).

in problem structure. SumHeight identifies and exploits non-uniformity among resources while CBASlack identifies and exploits non-uniformity among activity pairs. When non-uniformity at a resource level exists SumHeight outperforms CBASlack. When such a non-uniformity is lacking CBASlack performs better.

While we have not seen this intuition about heuristics stated in the literature, as such, we do not expect its explicit identification to be controversial: it can be seen as underlying much of the research into heuristics for constraint-directed search (*e.g.*, [Gent et al., 1996a]).

4.9.1.3 Heuristic Commitments versus Implied Commitments

In Experiment 1, where there was no significant difference between of CBASlack and Sum-Height, we examined the percentage of commitments in the search that were heuristic commitments (Figure 23 and Figure 24). Results showed that a significantly larger percentage of the commitments for CBASlack are heuristic commitments.

There are two requirements for such a comparison to be meaningful. First, it is important that the heuristics make the same type of commitment. Clearly, if one heuristic completely sequences a resource in a single heuristic commitment, it will incur a lower percentage of heuristic commitments than a heuristic that posts a sequencing constraint between a pair of activities. The second requirement is similar search performance. A search state for a problem is more constrained if it occurs deep in a search tree (after a number of commitments have been added to the graph) than early in the tree. A chronological backtracking algorithm that is not performing well is likely to spend a great deal of time near the bottom of a search tree where each heuristic commitment results in many implied commitments. In contrast, an algorithm that easily finds a solution spends

a relatively small portion of the search at the bottom of the tree and therefore will have a higher percentage of heuristic commitments.

With these caveats, our results indicate that SumHeight has a much lower percentage of heuristic commitments than CBASlack. Based on how the heuristics choose a pair of activities to sequence, this is a reasonable result. CBASlack chooses the activity pair with minimum biased-slack regardless of the presence of any other activities competing with the pair. Therefore, after a CBASlack commitment, it may be the case that few other activities are affected and, therefore, there is little propagation. SumHeight, on the other hand, selects a pair of activities precisely because they are part of a larger subset of competing activities. Because SumHeight explicitly looks for areas where there are a number of activities, it is reasonable to expect that a single commitment will, though the propagators, lead to a greater number of implied commitments than CBASlack.

Intuitions on Heuristic Performance

A more important consideration of the difference in heuristic commitment percentage is the meaning, if any, that it has for the quality of the heuristic. Here we present intuitions about the nature of heuristic commitment techniques. These intuitions are speculative and their investigation forms the basis for our future work on heuristics.

We believe that there are two characteristics that correlate highly with the success of a heuristic commitment technique.

1. The Short Dead-end Proof Intuition: the ability of a heuristic to quickly discover that the search is at a dead-end. Imagine a search that progresses through a series of states to state *S*. Assume that there are no solutions in the subtree below *S*, but that we are unable to prove this at state *S*. Some heuristic commitments must be made to explore the subtree below *S* in order to prove that *S* is a dead-end.[12] A heuristic that is able, through its heuristic commitments and subsequent propagation, to prove a dead-end while exploring a small number of states in the subtree will be superior to one that must explore a larger subtree.
2. The Few Mistakes Intuition: a high likelihood of making a commitment that leads to a solution. A heuristic that has a higher likelihood of making a commitment that leads to a solution, that is one that does not result in a dead-end, is likely to be a higher quality heuristic than one with a lower probability of making the correct decision.

These two characteristics are independent from the perspective that if either characteristic were infallible, search performance would be independent of the other characteristic. If a heuristic never made a commitment that resulted in a dead-end state, the ability to detect such a state is irrelevant. Similarly, if a search could immediately detect that it was at a dead-end, then the quality of the commitments is irrelevant: any mistake is immediately detected and repaired.

With real, fallible heuristics, these two characteristics interact: it may be the case that a heuristic with a high probability of making the right decision results in a relatively large effort in discovering when it has made a mistake. Conversely, a heuristic with a lower probability of making a right decision may be able to find a shorter proof of a dead-end (on average) and therefore recover from mistakes quickly. The best heuristic will be the one that minimizes the size of the overall search tree and this may well be achieved by some trade-off between the two characteristics.

12. We are assuming a provable retraction technique like chronological backtracking.

Implications for SumHeight and CBASlack

Returning to SumHeight and CBASlack, it is our intuition that the lower percentage of heuristic commitments for SumHeight than for CBASlack indicates that SumHeight is able to find shorter proofs of a dead-end. More commitments in the search tree below an unknown dead-end will be implied commitments and therefore the size of the subtree that must be investigated will tend to be smaller.

In Experiments 1 and 2, this ability to find short proofs (if our intuitions are accurate) does not result in superior performance. A possible explanation for this is a higher probability for CBASlack to make the correct decision. CBASlack identifies an activity pair, such as the one shown in Figure 45, where the sequence is almost implied by the existing time windows. SumHeight on the other hand selects a pair of activities that may have significantly more overlapping time windows. We speculate, therefore, that in Experiment 2, the "more obvious" commitment made by CBASlack has a higher probability of being in a solution than the commitment made by SumHeight and further, that this higher probability is enough to overcome the shorter-proof characteristics of SumHeight.

In Experiment 3 we believe that the presence of bottleneck resources reduces the probability that CBASlack will make the correct commitment to such a point that the short proof characteristic of SumHeight results in superior overall performance. Imagine two problems, P and $P_{bottleneck}$, with the only difference between them being that extra activities are added to resource R_1 in $P_{bottleneck}$ to make it a bottleneck resource. A heuristic commitment in P can have greater impact on the activities of R_1 without resulting in an eventual dead-end than the same commitment would have in $P_{bottleneck}$. Because CBASlack ignores the presence of a bottleneck, the likelihood that it will make a correct commitment is smaller in $P_{bottleneck}$, where a bottleneck is present, than in P. We speculate that this reduction in the likelihood that a commitment will be correct can account for the results of Experiment 3.

It should be noted again that much of this section is speculative. Many of our intuitions do not, as yet, have strong empirical support, and we are not claiming that our "explanation" for the heuristic comparison is the correct one. We believe that we have expressed a viable explanation that accounts for the current results; however, it should be regarded primarily as a starting point for further research.

4.9.2 Retraction Techniques

For Experiments 1 and 2, LDS was clearly superior to chronological backtracking in terms of ability to solve more problems and the mean CPU time incurred. There was little statistically significant overall difference in Experiment 3. One pattern seen in almost all experiments is that LDS significantly outperforms chronological backtracking on the looser problems (*e.g.*, problems with higher makespan factor or fewer bottlenecks) while there is no significant difference on the tighter problems. Our intuition as to the explanation for this pattern is the presence of overconstrained problems in our problem sets (see Section 4.9.2.1).

In general, these results are consistent with previous work and our expectations. LDS makes larger jumps in the search space than chronological backtracking, undoing a number of commitments in one backtrack. The larger jumps mean that there will often be more effort for each backtrack of LDS than of chronological backtracking. Often, as in Experiments 1 and 2, the extra effort pays off in terms of solving more problems in less time. In Experiment 3, however, the extra effort at each backtrack did not result in better overall performance: the effort incurred by making

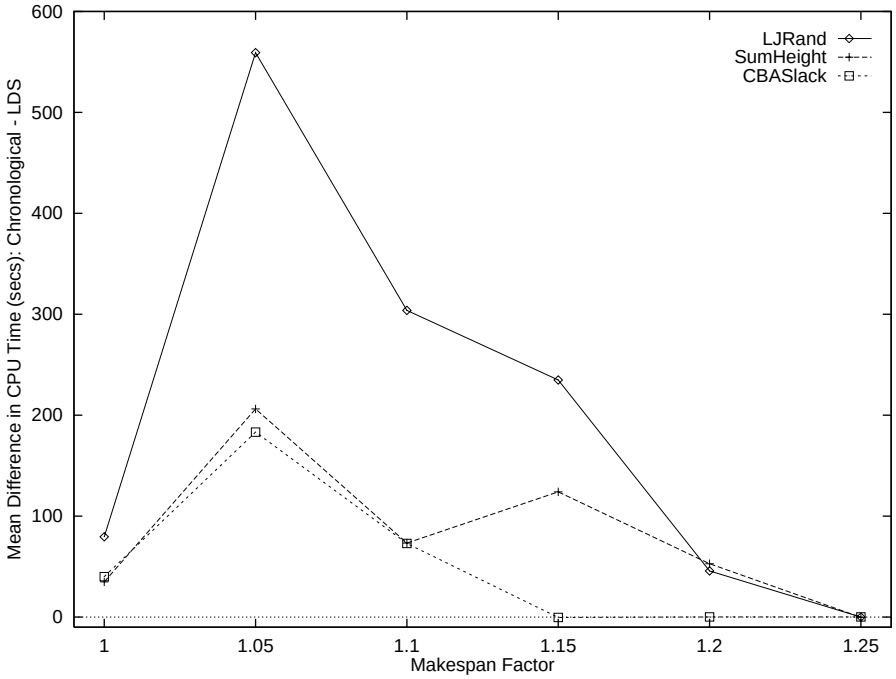


Figure 47. The Mean Reduction in CPU Time in Seconds When Using LDS Instead of Chronological Backtracking for Each Heuristic.

significantly more commitments (while making fewer backtracks) was not reflected in solving more problems.

4.9.2.1 The Presence of Overconstrained Problems

LDS significantly outperforms chronological backtracking on the looser problems (*e.g.*, problems with higher makespan factor or fewer bottlenecks) while there is no significant difference on the tighter problems. This pattern can be understood by the presence of overconstrained problems in our problem sets. In an overconstrained problem, an algorithm using LDS will expend more resources proving a problem is overconstrained than the corresponding algorithm using chronological backtracking. This is due to the fact that LDS must revisit some search states in performing a complete search. In our experiments, the problems on which chronological backtracking outperforms LDS can be attributed to the presence of overconstrained problems. Examples of this are the problem sets with low makespan factor in Experiment 2 (*e.g.*, makespan factor 1 and 1.05 in Figure 27 and Figure 28) and the problem sets with a high number of bottleneck resources in Experiment 3 (*e.g.*, sets with 12 and 14 bottlenecks in Figure 37 and Figure 38).

4.9.2.2 Improving Heuristics

In terms of comparing the retraction components, we see that on all the search performance measures, the LDS algorithms tend to outperform their counterparts using chronological backtracking. The size of the improvement when using LDS rather than chronological backtracking is also an interesting statistic. It has been observed [Beck et al., 1997b; Le Pape and Baptiste, 1997] that when moving from chronological backtracking to LDS, the heuristics that performed worse with chronological backtracking are improved more than the heuristics that performed better: the differences among the heuristics are less with LDS than with chronological backtracking. If we examine the magnitude of improvement, we also observe this trend. Figure 47 plots the mean difference in CPU time between LDS algorithms and chronological backtracking algorithms for

each problem set in Experiment 1. The greatest improvement is seen with LJRand which is improved significantly more ($p \leq 0.005$) by using LDS than either SumHeight or CBASlack. No significant difference in the magnitude of the improvement is observed between CBASlack and SumHeight.

While the trend of a greater improvement for weaker heuristics has been observed before, it remains at the level of observation. It may be that some or all of this phenomenon is due to a ceiling effect:[13] because LJRand does so poorly with chronological backtracking there is much more room for improvement when using LDS than is the case for the other, better heuristics. If the effect is real, rather than an artifact of the experimentation, we do not have and are not aware of any explanation for it. It remains an interesting observation, but whether it has any implications for retraction techniques or heuristics remains future work.

It should be noted that despite improving the weaker heuristics more than the stronger ones, LDS did not change the relative heuristic performance. In many cases where there was a significant difference among heuristics, that difference was manifest with both chronological backtracking and LDS. In some cases, a significant difference disappeared; however, we were not able to find results where there was a significant difference between heuristics with one retraction technique and the opposite significant difference with the other technique.

4.10 Conclusions

This chapter examined three heuristic commitment techniques across three experiments. The purpose of these experiments was to investigate the analysis of each search state as a basis for heuristic commitment techniques and address criticisms in the literature with respect to the efficacy of texture-based heuristics.

Our results support the use of texture measurements as a basis for heuristic commitment techniques as the two heuristics (SumHeight and CBASlack) based on an analysis of a characteristic of the search state outperformed the less informed heuristic (LJRand) across all experiments. The comparison of SumHeight and CBASlack suggests that an explanation of their relative performance depends upon the presence (or absence) of non-uniformities in problem structure. This result also supports the use of texture-based heuristics as one of the key motivations for texture measurements is to distill non-uniformities in a search state.

Comparison of chronological backtracking and LDS retraction techniques show that LDS is, in general, superior to chronological backtracking. The only exception to this is in problem sets with overconstrained problems, where LDS, due to its larger search space, is inferior.

13. A ceiling effect is when two techniques are judged to be not significantly different due to the fact that they both perform very well on an experimental problem set. It may be the case that with a more difficult set of problems a significant difference would be detected. In this context, we speculate that a ceiling effect may contribute to the smaller improvement of SumHeight over LJRand when using LDS instead of chronological backtracking. SumHeightChron may perform too well on the experimental problem sets to be outperformed by SumHeightLDS to the same extent that LJRandChron is outperformed by LJRandLDS.

Chapter 5 The Criticality of Constraints

The goal of this chapter is to develop a measurement for the criticality of constraints together with practical algorithms for its estimation. Based on the results of Chapter 4, contention is a successful measurement of criticality of resource constraints in job shop scheduling. Unfortunately, contention, as we argue in this chapter, has limited applicability: there are constraints that exist in real-world scheduling problems for which contention does not appear to provide a meaningful measure of criticality. Therefore, we generalize from contention to criticality itself, examine the requirements of a general measure of criticality, and propose the use of the probability of breakage of a constraint as such a measure. We create three estimation techniques for the probability of breakage of a constraint and, in this chapter, empirically evaluate each in the context of job shop scheduling. This initial empirical evaluation provides a basis for analysis of our new techniques in a well-understood and well-studied problem model. This is important, before addressing novel constraints in Chapter 6, in order to gain insight into the behavior and quality of heuristic search techniques based on estimations of the probability of breakage of constraints.

5.1 From Contention to Criticality

The heuristic proposed by [Sadeh, 1991] and its variation, SumHeight, investigated in Chapter 4 equate the criticality of a resource constraint with contention. The more intense the competition for a resource, the higher the contention, and, therefore, the more critical is the constraint. Such a model of criticality is reasonable for constraints which represent some maximum limit, *M*, on the number of variables that can have the same value. Such maximum limit constraints are common, and extremely important, in scheduling (*e.g.*, in the representation of unary and multi-capacity resources). However, it is desirable to be able to compare the criticalities of all constraints in our graph. For example, in scheduling, we encounter minimum and maximum storage constraints on raw materials, work-in-process, and finished goods inventories as well as unary and multi-capacity resource constraints. Our goal, then, is to be able to estimate and compare the criticalities of *all* types of constraints so that, in a search state, we can find the most critical constraint and make a heuristic commitment to reduce its criticality.

5.1.1 The Criticality of a Constraint

Before proceeding further, it is necessary to be more definitive with respect to our use of the term *criticality* for a constraint. [Sadeh, 1991, p. 43] postulates that a critical variable is "one that is expected to cause backtracking." Applying the same intuition to constraints, we define a critical constraint as one that is expected to be unsatisfied given the current search state. This definition is

both firmly grounded in the context of combinatorial search and widely applicable. The grounding of criticality in search is due to the fact that the crucial application of constraint criticality is as a basis for heuristic search techniques. The information that we believe is most useful from that perspective is the expectation of breakage. The wide applicability of the definition of criticality is also important. We do not want to limit, *a priori*, the types of information that can be used as a measure of criticality. It is an open question as to what measures of criticality are most effective as a basis for heuristic search and we do not want to confine the investigation of such measures with an overly restrictive definition.

5.1.2 Contention and Aggregate Demand

Given this definition of criticality of a constraint, we now return to the contention texture measurement. We claimed above that contention is a measure of criticality for a maximum resource constraint. It is useful, therefore, to examine precisely what characteristic is estimated by contention in order to understand how that characteristic correlates with criticality.

It is our belief that contention estimates the true aggregate demand (over time) for a resource. Given a search state, *S*, a resource, *R*, and a resource constraint, *RC*, expressing that the maximum capacity of *R* is *M*, we can exactly calculate the aggregate demand for *R* in state *S* as follows: remove *RC* from the scheduling problem and enumerate the set of solutions, *SOLS*(*S*), that can be developed from *S*. The aggregate demand, *AD*(*R*, *t*, *S*), for *R* at some time, *t*, on the scheduling horizon can be found by calculating the mean number of activities executing on *R* at time *t* over all solutions, $sol \in SOLS(S)$ as shown in Equation (20). *NumActs*(*R*, *t*, *sol*) is the number of acts executing on resource *R* at time *t* in solution *sol*.

$$AD(R, t, S) = \frac{\sum_{sol \in SOL(S)} NumActs(R, t, sol)}{|SOLS(S)|} \tag{20}$$

In a scheduling problem where all resource constraints have the same maximum value (such as in job shop scheduling), an estimate of the aggregate demand can be used directly as a measure of criticality. The constraint with the highest aggregate demand (at some time point) is the most likely to be broken. Extending the scheduling problem by allowing multi-capacity constraints prevents the direct comparison of aggregate demand for criticality: obviously an aggregate demand of three (at some time) on a resource with capacity of one is more critical than an aggregate demand of 20 (at some time) on a resource with capacity 100. In such a situation, perhaps, the aggregate demand can be transformed into a measure of criticality by normalizing it with the capacity of each resource (such an approach is taken in the KBLPS system [Saks, 1992]). It is not clear, however, that normalization necessarily results in a measure that is well correlated with criticality. Is it truly the case that a resource of capacity two with an aggregate demand (at some time) of ten is as likely to be broken as a resource of capacity one with an aggregate demand (at some time) of five? It might be argued that there are more activities that must be scheduled elsewhere in the former case than in the latter (*i.e.*, eight versus three) and therefore the resource with capacity two is actually more critical. Further investigation is necessary to evaluate the correctness of this argument and the intuitions behind normalization. Moving one step further, however, we also want to be able to estimate the criticality of a minimum constraint. What is the likelihood that the resource usage, at some time, is less than some minimum capacity, *m*? Simple normalization of the aggregate demand with *m* does not seem to provide a solution.

5.1.3 Requirements for a Measure of Criticality

The above discussion of the use of aggregate demand (and its estimate, contention) as a measure of criticality for a wider variety of constraints raises the issue of the characteristics that we require and desire for such a measure. Based on experience with contention, we have identified the following four requirements for any measure of constraint criticality:

1. Wide conceptual applicability – A measure of criticality should be applicable, at a conceptual level, to a very wide range of constraint types. By conceptual applicability, we are specifically ignoring issues of practical application. Rather, we want the meaning of the criticality measure to be reasonable given the intended semantics, if any, of the constraint. For example, in the above discussion of the use of aggregate demand, it was not clear what meaning it had for a minimum capacity constraint.
2. Syntactic comparability – At a practical level, we must be able to compare the actual values representing the criticalities of different constraints. For example, it is not reasonable to base criticality on the direct comparison of the aggregate demand for resources with different maximum capacities. We require that a measure of criticality define a completely ordered, closed interval from which criticality values must be drawn.
3. Semantic comparability – A criticality value of *k* for one type of constraint must denote the same level of criticality as a value of *k* for some other type of constraint. While this is an obvious requirement from the perspective of wanting to compare criticalities among different types of constraints, it is not clear, as noted above, that aggregate demand when normalized for multi-capacity resources meets this requirement.
4. Practical utility – A measure of criticality must have some practical utility as a basis for heuristic commitment techniques. While such utility is not solely a result of the criticality measurement itself, but rather the measure in combination with estimation algorithms, we nonetheless believe that, in order to ground the research in empirical evidence, a measure of criticality should be able to form the basis for heuristic commitment techniques that are as good as or better than other, non-criticality-based, heuristic approaches.

These requirements represent our starting place for investigations into criticality measures. It may be the case that further research will add or even remove requirements.

Given the practical goals of this dissertation in terms of applying measures of criticality to constraints that exist in real-world scheduling problems, there are also a number of desirable characteristics for a measure of criticality and estimations of such a measure. Ideally, we would like efficient algorithms to be able to exactly compute our measure of criticality. This desire is not very likely to be met; therefore, we want efficient algorithms that are able to estimate a measure of criticality. We expect that the computational complexity of an estimation technique will have to be traded-off against the accuracy of its estimates. Finally, we would like estimation algorithms to have a wide applicability across a variety of constraint-types. As discussed below (Section 5.2), we do not expect that one estimation technique will be applicable to all types of constraint. Nonetheless, the ability to apply the same estimation across some subset of constraint-types (provided it does not have negative implications to computational complexity or accuracy) is desirable.

5.2 Probability of Breakage of a Constraint

Based on the intuitions provided from the analysis of the contention texture measurement and the requirements identified for a measure of criticality, we propose to measure the criticality of a con-

straint by its probability of breakage. In a search state, *S*, the probability of breakage of constraint, *C*, can be calculated by the ratio of the number of complete valuations of the variables that fail to satisfy *C* to the total number of complete valuations. Given that, in state *S*, some variables may have been assigned values or their domains may have been pruned by previous commitments, the probability of breakage of *C* in *S* can be very different from that of *C* in *S'*, $S' \neq S$. Exact calculation of the probability of breakage of constraint is impractical and we need to look to techniques for its estimation.

From the perspective of our requirements for a measure of criticality, we note that the probability of breakage meets the first three: it is widely applicable on a conceptual level, and is both syntactically and semantically comparable. The final requirement, that of practical utility, remains to be investigated through the implementation of estimation algorithms and experimentation. The balance of this chapter conducts this investigation.

5.2.1 Estimation of the Probability of Breakage

We do not believe that the formulation of a single, widely applicable estimation for the probability of breakage of a constraint is likely to be useful in extending the power of heuristic search techniques. Conceptually, the probability of breakage of any constraint, *C*, in a problem can be found by removing *C* from the problem and enumerating all solutions. The ratio of these solutions that fail to satisfy *C* to the total number of solutions is the probability of breakage of *C*. While this formulation is universally applicable, it is not useful in the formation of heuristics for solving the problem in the first place as it assumes the solutions can be enumerated.

Less general is an estimation of the probability of breakage of a constraint at the generic constraint level in the spirit of the texture measurements defined in [Fox et al., 1989] and the κ measurement in [Gent et al., 1996b]. Such an estimate might be based on the number of values in the domain of a variable, simple estimates of the total number of solutions, and/or the total number of remaining variable valuations. While heuristics based on these estimates have been shown to be useful for general CSPs [Gent et al., 1996b], one of the key lessons from early work in the constraint programming (CP) community is that such generic techniques are not able to successfully address the complexities inherent in scheduling problems [Beck et al., 1998]. The CP community began making significant advances to the power of scheduling techniques through the development of specific propagation techniques for different types of constraints found in scheduling (*e.g.*, [Van Hentenryck, 1989; Caseau and Laburthe, 1995; Caseau and Laburthe, 1996; Nuijten, 1994; Le Pape, 1994c; Le Pape and Baptiste, 1996]).

Based on this reasoning, the approach we follow is to define the criticality of a constraint as its probability of breakage and to allow that different types of constraints may require different estimation algorithms. If possible, we would like to find estimation algorithms that both lead to efficient estimation of criticality and are applicable to a number of types of constraints; however, we do not require such algorithms. The development of estimation algorithms is not necessarily a straightforward process: we expect different approaches to the estimation of criticality for a constraint to display different characteristics in terms of computational complexity and the quality of estimation. Investigation of such characteristics for a variety of constraints is, therefore, a key challenge. The eventual goal is to have estimation techniques for the types of constraints relevant to the problems we wish to solve and so to be able to integrate them all into the dynamic focusing abilities of a texture-based heuristic commitment technique.

In the following three sections, we begin the investigation of estimation algorithms for the probability of breakage of a constraint. We present three estimations of the probability of breakage of a

unary capacity resource constraint. The first estimation (JointHeight) is the most principled as it avoids the assumptions of the other two techniques; however, it also incurs a higher complexity and is not readily generalizable beyond the unary capacity resource constraint. The other two estimations (TriangleHeight and VarHeight) incur the same complexity as SumHeight (Section 4.3) while being fully generalizable to multi-capacity resource constraints as well as inventory minimum and maximum constraints. We investigate such generalizations in Chapter 6.

5.2.2 The JointHeight Texture

Following the format of the contention texture measurement, given a constraint *C*, a set of variables constrained by *C*, and the individual demand of each variable, we want to calculate the probability of breakage of the constraint by aggregation of the contributions from individual variables.

The JointHeight algorithm is based on the idea that a unary resource constraint is broken if two activities execute at the same time point. To calculate the probability that two independent activities will execute at a time point, we can simply multiply their individual demands (which we interpret as a measure of probability) for the time point. Activities are independent unless they are connected by a directed path of precedence constraints, in which case their joint probability is 0. This suggests that we calculate the probability of breakage, *pb*, as follows:

> Let Ω be an ordered set of activities on the resource, *R*, and *C* be the resource capacity constraint. The function $connected(A_i, A_j)$ is true if there is a directed path of precedence constraints connecting activity A_i with A_j.

$$pb(C, t) = \sum_{\forall A_i \in \Omega, \forall A_j \in \Omega, i < j} joint(A_i, A_j, t) \tag{21}$$

$$joint(A_i, A_j, t) = \begin{cases} 0 & connected(A_i, A_j) \\ ID(A_i, R, t) \times ID(A_j, R, t) & otherwise \end{cases} \tag{22}$$

The JointHeight algorithm is identical to SumHeight until the point where the aggregate curve is calculated. Where SumHeight sums the individual demand for the resource from each activity at each time point, JointHeight evaluates each pair of activities on the resource using Equations (21) and (22).

There are two weaknesses with JointHeight:

1. The time complexity at a single search state is $O(mn^3)$ (see below).
2. The formulation depends strongly on the fact that two activities cannot co-occur. It is difficult, therefore, to see how JointHeight can be efficiently extended to multi-capacity resource and inventory constraints.

5.2.2.1 Complexity

We cannot use the mechanism used in SumHeight (Section 4.3.3) to find the aggregate curve. At a single time point, we must examine all pairs of activities on the resource using Equation (22). This alone incurs a time complexity of $O(n^2)$. There are $O(n)$ time points (due to our event repre-

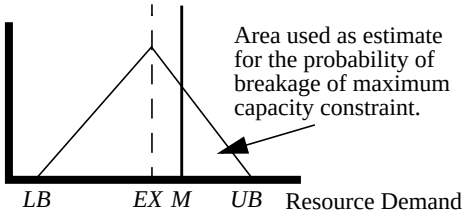


Figure 48. Calculating the Probability of Breakage at Event *t* with TriangleHeight.

sentation) and *m* resources; therefore, the total complexity at one search state is $O(mn^3)$. The space complexity is the same as SumHeight: $O(mn)$.

5.2.3 The TriangleHeight Texture

TriangleHeight estimates the probability of breakage based on the expected value of the aggregate demand and a distribution around the expected value. The aggregate curve on a resource as calculated by SumHeight provides the expected value at each event point. In addition, we can find the upper bound and lower bound on the aggregate demand (see below, Section 5.2.3.1). By assuming a triangular distribution around the expected value, the probability of breakage at *t* can be estimated. Given the lower bound, the upper bound, and the fact that the area of the triangle is 1, the height of the triangle can be easily found (see Figure 48). For a maximum capacity constraint, *M*, the area under the curve to the right of the line representing *M* is interpreted as the probability that the maximum capacity constraint will be broken.

For each event point, *t*, we use the expected value and distribution as in Figure 48. We calculate the area of the triangle that is greater than the maximum constraint and use it as an estimate of the probability that the constraint will be broken at *t*.

The primary disadvantage of TriangleHeight is that we have no justification for a triangular distribution. We expect, therefore, that the estimation of probability will be worse than JointHeight, and thus that our heuristic decisions based on TriangleHeight will be more error prone than those based on JointHeight. In addition, for TriangleHeight we have to maintain the upper and lower bound curves as well as the aggregate curve.

The advantages of this method, however, may outweigh the disadvantages. For example:

1. We can estimate the probability of breakage of a minimum capacity constraint by examining the area under the curve to the left of the constraint line (see Figure 48).
2. We can generalize this method easily for non-unary capacity resources and inventory constraints.
3. We can use the same event-based technique used in SumHeight.

5.2.3.1 Calculating the Bounds on Aggregate Demand

One of the requirements of the TriangleHeight texture is the calculation of upper and lower bounds on the aggregate demand on a resource. We use an event-based curve to represent each bound and so, as with the aggregate demand, each activity has an individual contribution to each bound. All individual contributions are summed, using the same technique as for aggregate demand in SumHeight (see Section 4.3.3), to form the corresponding bound curve.

The lower bound contribution to the individual resource demand for activity, *A*, is 0 unless, $lst_A < eft_A$. In such a situation, *A* must execute on the resource on the interval $[lst_A, eft_A)$ and therefore the contribution of *A* is represented by two event points: $(lst_A, 1)$ and $(eft_A, -1)$.[1]

For the upper bound, an activity can potentially use a resource at any time point between its earliest start time and latest finish time. Therefore, the upper bound contribution of an activity *A* is represented by two event points of the form: $(est_A, 1)$ and $(lft_A, -1)$.

5.2.3.2 Complexity

The upper and lower bound curve can be maintained in much the same way and with the same complexity as the aggregate demand curve. As in SumHeight, we sort all the individual demand elements and, with a single pass through the list, calculate the probability of breakage using the expected value and the triangular distribution. Overall, TriangleHeight has the same complexity measurements as SumHeight: time complexity of $O(mn \log n) + O(mn)$ and space complexity of $O(mn)$.[2]

5.2.4 The VarHeight Texture

VarHeight is similar in form to TriangleHeight, but uses a different estimate of the distribution around the expected value. The probability of breakage is estimated based on the expected value and the distribution created by the aggregation of the variance of the individual demands.

In considering resource *R*, a time point *t*, and an activity *A*, we can associate a random variable *X* with the demand that *A* has for *R* at time *t*. For unary resources the domain of *X* is {0, 1}. The expected value for *X*, *EX*, assuming a uniform distribution for the start time of *A*, is *ID*(*A*, *R*, *t*) as calculated in Equation (14) (Section 4.3.1). We calculate the variance of *X*, *VX*, as follows:

$$VX = EX \times (1 - EX) \tag{23}$$

Derived as follows:

(1) $VX = EX^2 - (EX)^2$ and $EX^2 = \sum x^2 p(x)$ by definition.

(2) *x* can take on only the values in {0, 1} and $p(x) = ID(A, R, t)$.

(3) Therefore, $EX^2 = \sum x^2 p(x) = \sum xp(x) = EX$.

(4) So $VX = EX^2 - (EX)^2 = EX - (EX)^2 = EX \times (1 - EX)$.

---

1. We are assuming that an activity uses one unit of a resource. The bound calculations can be easily extended for the non-unary case by substituting the amount of the resource used by each activity.
2. This complexity is for the creation of the texture curves. A heuristic based on the texture measurement may incur a higher complexity in forming a commitment.

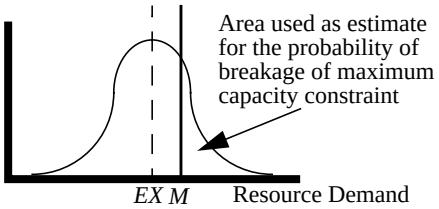


Figure 49. Calculating the Probability of Breakage at Event *t* with VarHeight.

Given this definition, we can calculate EX_i and VX_i for all activities, A_i, at time point *t*. We would now like to aggregate these individual measures to find a measure of the total expected value and variance at event *t*. To do this we use the following two theorems:

Theorem 1. The expected value of a sum of random variables is the sum of their expected values.

Theorem 2. If *f* and *g* are independent random variables on the sample space *S*, then $V(f + g) = V(f) + V(g)$, where $V(x)$ is the variance of random variable *x*.

Proofs of these theorems can be found in statistics texts (*e.g.*, [Bogart, 1988] p. 573 and p. 589, respectively).

These theorems allow us to sum the expected values and variances from each activity, provided we make the following assumption:

Egregious Assumption 1. The random variables associated with each activity are mutually independent.

This assumption is false. We will be sequencing these activities by posting precedence constraints and therefore their random variables will become increasingly interdependent. Reality notwithstanding, we make this assumption and form the aggregate expected value and variance by summing the constituent expected values and variances.

The aggregate expected value and variance form a distribution that can be used to represent the aggregate demand for resource *R* at time point *t*. By comparing this distribution with the capacity of *R* at *t* we can find an estimate for the probability that the capacity constraint will be broken at *t*. To do this we will make our second assumption:

Egregious Assumption 2. The aggregate random variable is normally distributed around the expected value.

Given this assumption (which, based on anecdotal simulation evidence appears more well-founded than our first), we can compare the expected value and the standard deviation of the random variable with the capacity at *t*. This is illustrated in Figure 49. The area under the curve greater than the maximum capacity constraint is used as an estimate of the probability of breakage.

The disadvantages of the VarHeight method center around the assumptions. Since these assumptions do not actually hold, we expect that VarHeight, like TriangleHeight, will be inferior to JointHeight in terms of the quality of its heuristic commitments. Another disadvantage is that because of the shape of the individual variance curve over time, we need to use six event points in our individual activity curve representation. Furthermore, each event must contain not only the

expected value and variance, but also the incoming and outgoing slope of these curves at that event. In addition to the four points in Expression (16) (Section 4.3.1), we add the following two points to the individual activity curve (calculation of the expected values and variances is left as an exercise to the reader, based on Equations (14) and (23)):

$$t_5 = \frac{est + min(lst, eft)}{2}, t_6 = \frac{lft + max(lst, eft)}{2} \quad (24)$$

The advantages of VarHeight are the same as those of TriangleHeight: we can generalize it easily to minimum constraints, we can generalize it beyond unary capacity resource constraints, and we can use the same event-based formulation as used with SumHeight.

5.2.4.1 Complexity

As in SumHeight, we sort all the individual demand elements and, with a single pass through the list, calculate the probability of breakage using the expected value and variance curves. VarHeight, therefore, has the same complexity as SumHeight and TriangleHeight: time complexity of $O(mn \log n) + O(mn)$ and space complexity of $O(mn)$.

5.3 Empirical Studies

The purpose of the experimental studies in this chapter are to evaluate the three estimation algorithms for the probability of breakage. Before evaluating the heuristics on a wider variety of constraints, we use the job shop scheduling problem as an evaluation tool in order to ensure that the generalization of contention to the probability of breakage does not seriously damage the performance of texture-based heuristic commitment techniques for scheduling.

5.3.1 Instantiations of the ODO Framework

With the exception of the heuristic commitment techniques, the components of the ODO framework strategy are identical to those used in the experiments in Chapter 4. In particular, the termination criteria (20 minute time limit) and machine characteristics are identical (see Section 4.4).

Heuristic Commitment Techniques. We use three new heuristic commitment techniques in our experiments: JointHeight, VarHeight, and TriangleHeight. Each of these heuristics are based on their correspondingly named probability of breakage estimations. The heuristic commitment algorithm that makes commitments on the basis of the estimations is identical to that used with SumHeight in Section 4.3: the critical activities are the top two unsequenced contributors to the resource and the time point with highest probability of breakage. We post a sequencing constraint between the two selected activities according to the MinimizeMax, Centroid, and Random heuristic (see Section 4.3.2).

To provide a basis of comparison for the three new heuristic commitment techniques, we also include SumHeight and CBASlack as described in Chapter 4.

Table 5 displays a summary of the strategies that we experiment with below.

Strategy	Heuristic Commitment Technique	Propagators	Retraction Technique
JointHeightChron	JointHeight	All[a]	Chronological backtracking
VarHeightChron	VarHeight	All	Chronological backtracking
TriangleHeightChron	TriangleHeight	All	Chronological backtracking
SumHeightChron	SumHeight	All	Chronological backtracking
CBASlackChron	CBASlack	All	Chronological backtracking
JointHeightLDS	JointHeight	All	LDS
VarHeightLDS	VarHeight	All	LDS
TriangleHeightLDS	TriangleHeight	All	LDS
SumHeightLDS	SumHeight	All	LDS
CBASlackLDS	CBASlack	All	LDS

a. Temporal propagation, Edge-finding Exclusion, Edge-finding Not-First/Not-Last, CBA

Table 5. **Summary of Experimental Scheduling Algorithms.**

5.4 Experiment 1: Operations Research Library

The experiments to evaluate the new texture-based heuristics are identical to those run in Chapter 4 (see Section 4.6). In particular, the same problem sets are used.

5.4.1 Results

Complete results for Experiment 1 can be found in Section B.1 of Appendix B.

5.4.1.1 Problems Timed-out

Figure 50 and Figure 51 display the fraction of problems in each problem set that each algorithm was not able to solve or show to have no solution within the 20 minute time limit. Figure 50 shows the results for the algorithms that used chronological backtracking and Figure 51 shows the results for the algorithms using LDS.

With chronological backtracking, there are, overall, no significant differences in terms of the number of problems solved among VarHeight, CBASlack, JointHeight, and SumHeight. TriangleHeight solves significantly fewer problems than each of the other heuristics.[3]

The results when LDS is used as the retraction technique are that VarHeight, JointHeight, and TriangleHeight all time-out on significantly more problems than SumHeight ($p \leq 0.005$, for the TriangleHeight comparison). There are no significant overall differences among the other heuristics.

3. Unless otherwise noted, statistical tests are performed with the bootstrap paired-t test [Cohen, 1995] with $p \leq 0.0001$.

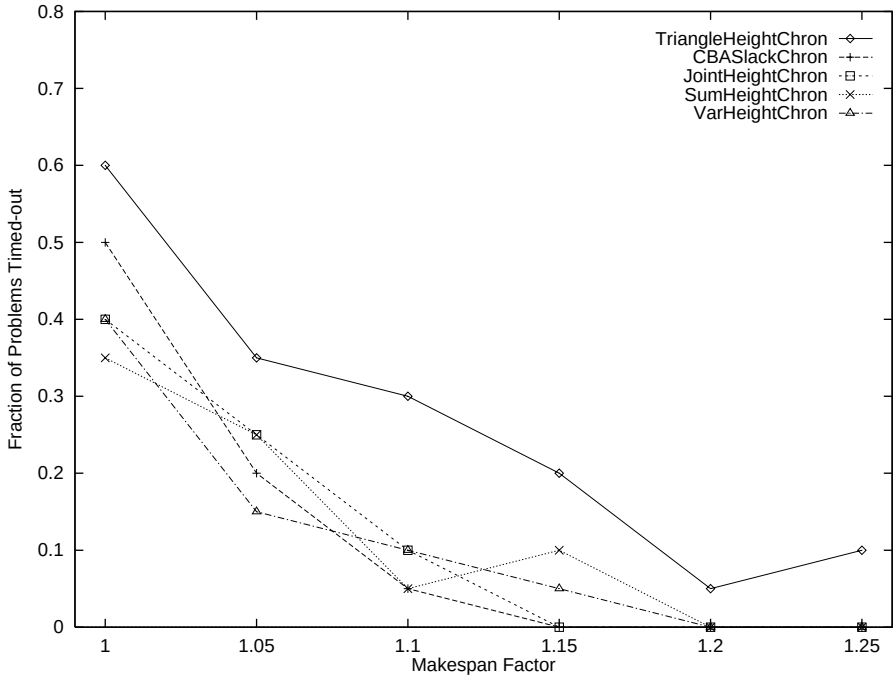


Figure 50. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (Chronological Backtracking).

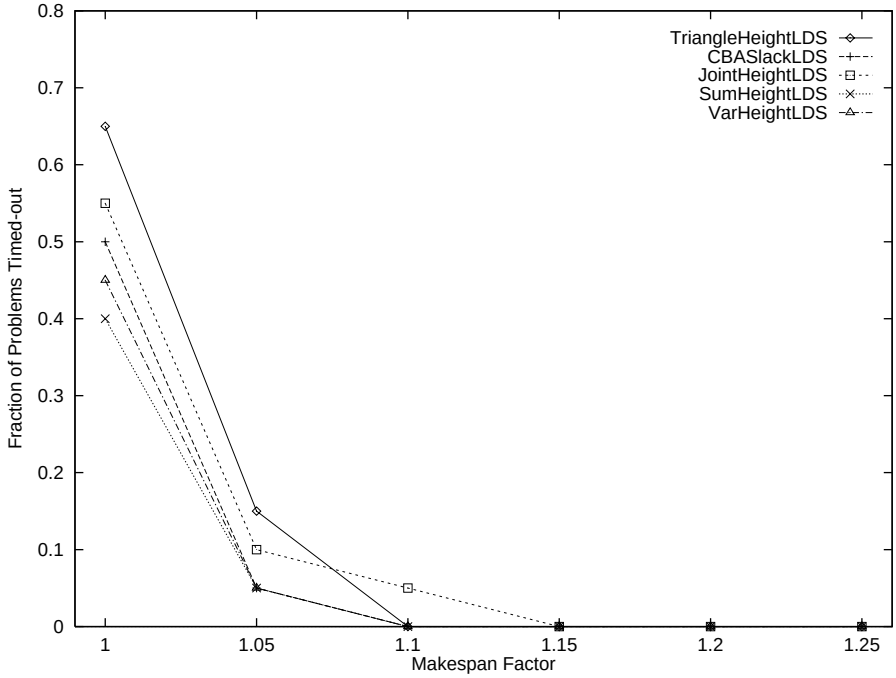


Figure 51. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (LDS).

When we compare the algorithms using chronological backtracking to those using LDS, we see that only TriangleHeight is significantly better with LDS than with chronological backtracking. The other heuristics show a non-significant improvement when using LDS rather than chronological backtracking.

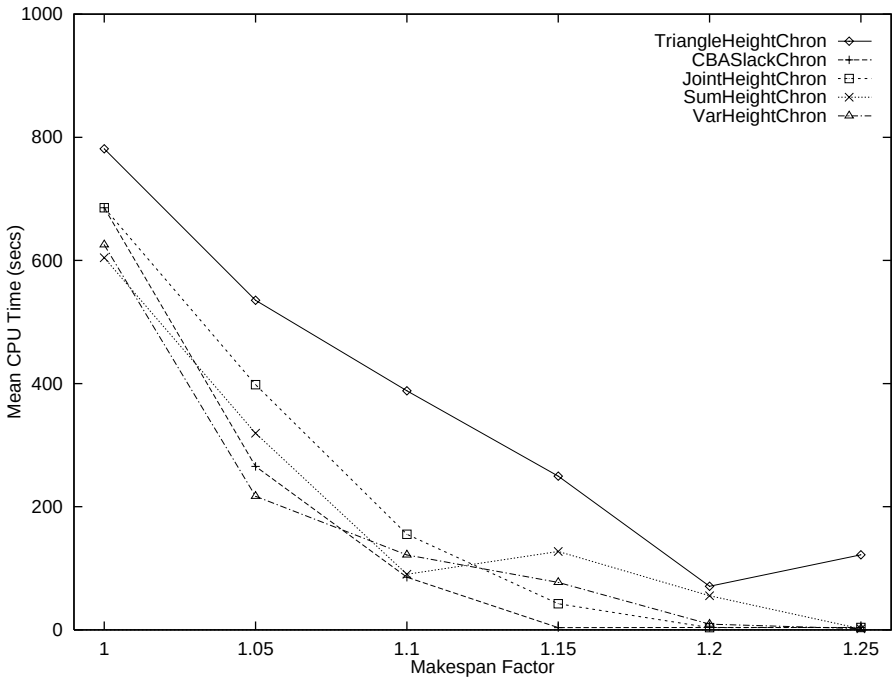


Figure 52. The Mean CPU Time in Seconds for Each Problem Set (Chronological Backtracking).

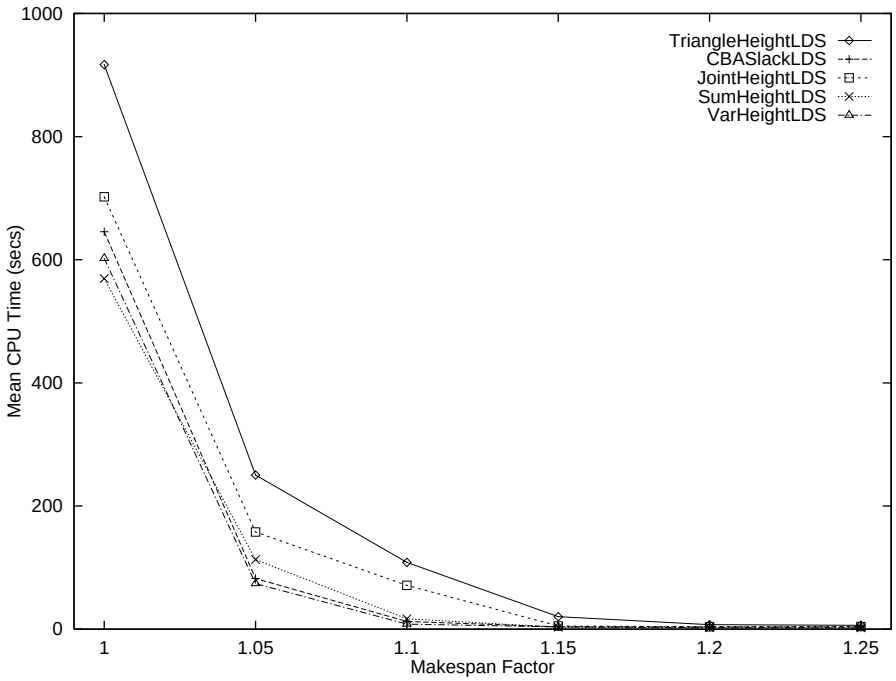


Figure 53. The Mean CPU Time in Seconds for Each Problem Set (LDS).

5.4.1.2 Mean CPU Time

The mean CPU time for each problem set are shown in Figure 52 and Figure 53 for algorithms using chronological backtracking and LDS, respectively.

The CPU results with chronological backtracking mirror the results for the number of problems timed-out: no significant differences among the heuristics except for TriangleHeight. All other heuristics are significantly better (lower CPU time) than TriangleHeight. These results are also seen when LDS is used as the retraction technique, with the only difference being that in addition to the significant differences between TriangleHeight and all other heuristics, VarHeight uses significantly ($p \leq 0.005$) less CPU time than JointHeight.

In the comparison of retraction techniques, we see significantly lower CPU times for LDS only with TriangleHeight. All other heuristics also incur a lower CPU time with LDS, but not significantly.

5.4.1.3 Other Search Performance Statistics

The other performance measurements tend to reflect the results demonstrated above and in Experiment 1 of the previous chapter.

- The mean number of backtracks indicates no significant differences among the heuristics except TriangleHeight which is significantly worse than all others. The algorithms using LDS make significantly fewer backtracks than those using chronological backtracking.
- JointHeight and TriangleHeight make significantly more commitments than CBASlack. In addition, TriangleHeight makes significantly more commitments than VarHeight. In comparing the retraction techniques, the tendency is for those algorithms with chronological backtracking to make fewer commitments; however, these differences are not significant. The only significant difference is with JointHeight which make significantly more commitments ($p \leq 0.005$) with chronological backtracking than LDS.
- TriangleHeight also makes significantly more heuristic commitments than any of the other heuristics, while there are no significant differences among the other heuristics. As with overall commitments, the trend with heuristic commitments is that those algorithms using LDS make more than those using chronological backtracking. None of these differences are significant except the one with CBASlack ($p \leq 0.005$) which follows this trend (*i.e.,* fewer heuristic commitments with chronological backtracking than with LDS).

5.4.2 Summary

The results for Experiment 1 indicate that:

- TriangleHeight is significantly worse than all other heuristics.
- There are few differences among SumHeight, VarHeight, JointHeight, and CBASlack.
- The comparison of the retraction techniques indicate that, on almost all problem sets, LDS outperforms chronological backtracking. However, this difference is only statistically significant when TriangleHeight was used as the heuristic commitment component.

5.5 Experiment 2: Scaling with Problem Size

As with Experiment 1, the same problem sets and experiments in Experiment 2 of Chapter 4 are used again here (see Section 4.7).

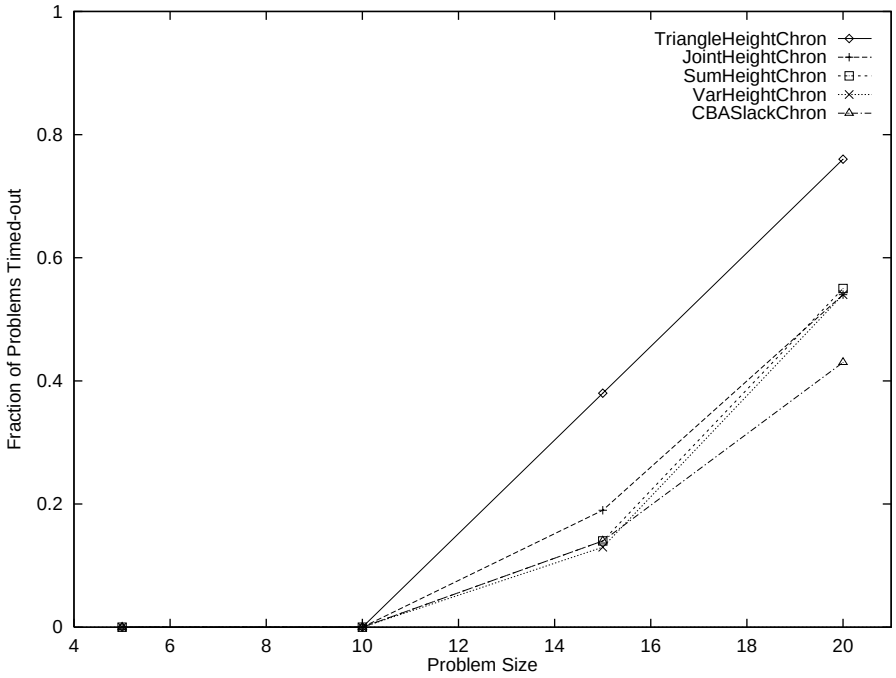


Figure 54. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (Chronological Backtracking).

5.5.1 Results

All results for Experiment 2 can be found in Appendix B. Tables in the following sections correspond to the various problem sets: overall – Section B.2.1, 5×5 – Section B.2.2, 10×10 – Section B.2.3, 15×15 – Section B.2.4, 20×20 – Section B.2.5.

5.5.1.1 Problems Timed-out

The number of problems that each algorithm timed-out on are displayed in Figure 54 and Figure 55. The former displays the results for the algorithms using chronological backtracking while the latter, the results for the algorithms using LDS.

The results in Figure 54 show that TriangleHeight solves significantly fewer problems than all other algorithms. JointHeight, while being significantly better than TriangleHeight and showing no significant differences from SumHeight or VarHeight, is significantly worse than CBASlack. While SumHeight is significantly worse than CBASlack ($p \leq 0.005$), there is no significant difference between VarHeight and CBASlack, or between VarHeight and SumHeight. Turning to the LDS results in Figure 55, TriangleHeight is again significantly worse than all other algorithms, CBASlack solves more problems than JointHeight ($p \leq 0.005$), and there are no other significant differences.

Figure 54 and Figure 55 indicate that the largest differences among the algorithms occur at the largest problems sizes. Therefore, Figure 56 displays the number of problems timed-out for each makespan factor of the 20×20 problems for the algorithms in the chronological backtracking condition. The same results for the algorithms in the LDS condition are shown in Figure 57.

For the 20×20 problems with chronological backtracking, CBASlack solves significantly more problems than all other algorithms while TriangleHeight solves significantly fewer problems than all other algorithms. Among JointHeight, VarHeight, and SumHeight there are no significant dif-

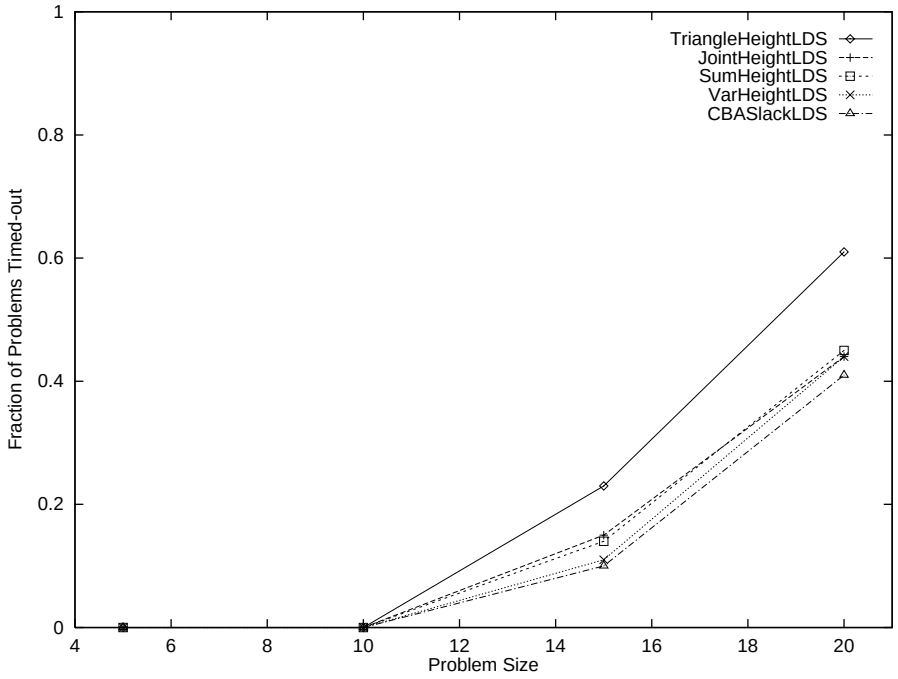


Figure 55. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (LDS).

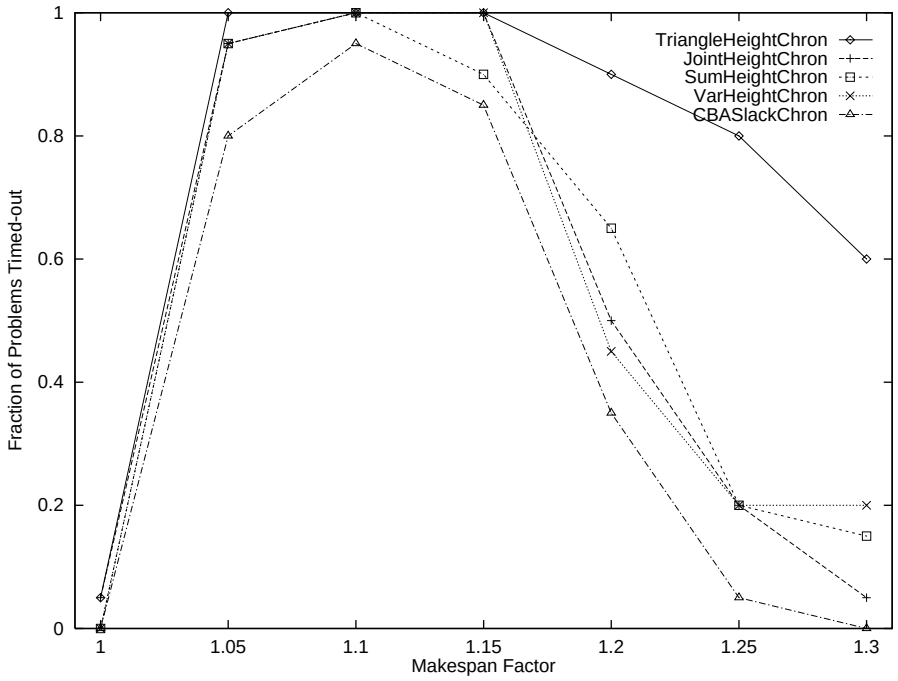


Figure 56. The Fraction of the 20×20 Problems at Each Makespan Factor for which Each Algorithm Timed-out (Chronological Backtracking).

ferences. With the LDS retraction technique, TriangleHeight is again significantly worse than all other algorithms, but there are no other significant differences.

The comparison between the retraction techniques shows that, overall, for the texture-based algorithms, those using LDS were able to solve significantly ($p \leq 0.005$) more problems than the cor-

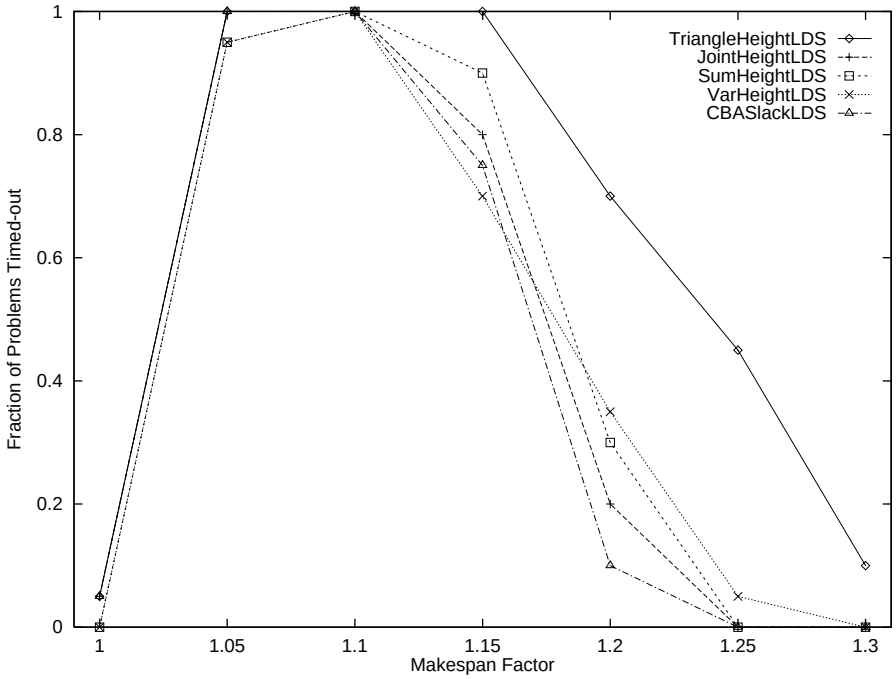


Figure 57. The Fraction of the 20×20 Problems at Each Makespan Factor for which Each Algorithm Timed-out (LDS).

responding algorithms using chronological backtracking. The difference for CBASlack, as discussed in the previous chapter, is not significant.

When just considering the 20×20 problems, the same pattern of significant differences is observed ($p \leq 0.0005$): the texture-based algorithms all solve significantly more problems with LDS than with chronological backtracking and there are no significant differences when CBASlack is used.

5.5.1.2 Mean CPU Time

The mean CPU times in seconds for each algorithm are shown in Figure 58 (chronological backtracking) and Figure 59 (LDS). When using either chronological backtracking or LDS, the statistical differences among the algorithms are the same: CBASlack achieves a significantly lower mean CPU time than all other algorithms ($p \leq 0.005$ for the comparison between CBASlack and VarHeight) while TriangleHeight has a significantly higher mean CPU time than all other algorithms. There are no significant differences among the other three texture-based algorithms.

The CPU time results for the 20×20 problems for the algorithms using chronological backtracking and those using LDS are presented in Figure 60 and Figure 61, respectively. The statistical results for the chronological backtracking algorithms for mean CPU time on the 20×20 problems match those of the overall statistical results: CBASlack achieves a significantly lower mean CPU time than all other algorithms while TriangleHeight incurs a significantly higher mean CPU time than all other algorithms. There are no significant differences among the other algorithms. The LDS results are similar with the only variation being that there are no significant differences in mean CPU time among CBASlack, VarHeight, and SumHeight.

Direct statistical comparison of the retraction techniques on the overall mean CPU time factor shows that LDS achieves significantly lower mean CPU time ($p \leq 0.0005$) than chronological backtracking for all algorithms except SumHeight and CBASlack. For the 20×20 problems, these

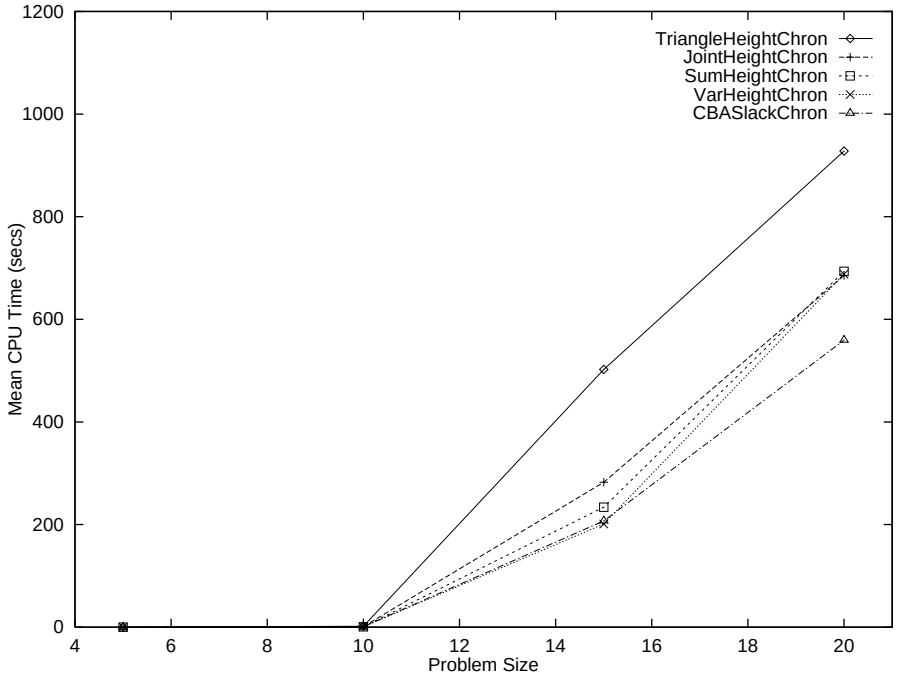


Figure 58. The Mean CPU Time in Seconds for Each Problem Set (Chronological Backtracking).

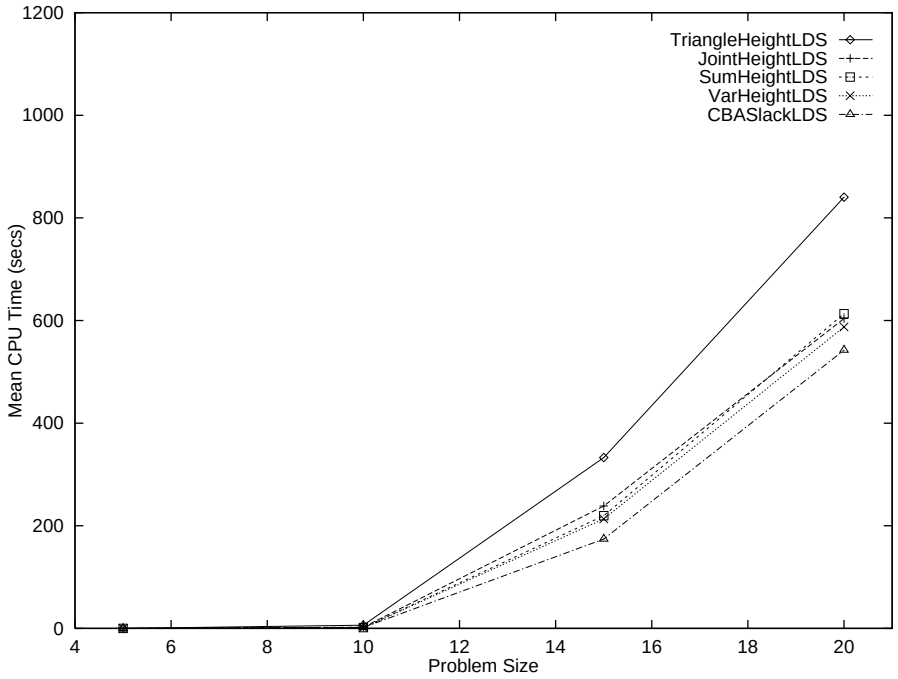


Figure 59. The Mean CPU Time in Seconds for Each Problem Set (LDS).

differences are repeated with the addition of SumHeight incurring significantly ($p \leq 0.005$) more mean CPU time with chronological backtracking than with LDS.

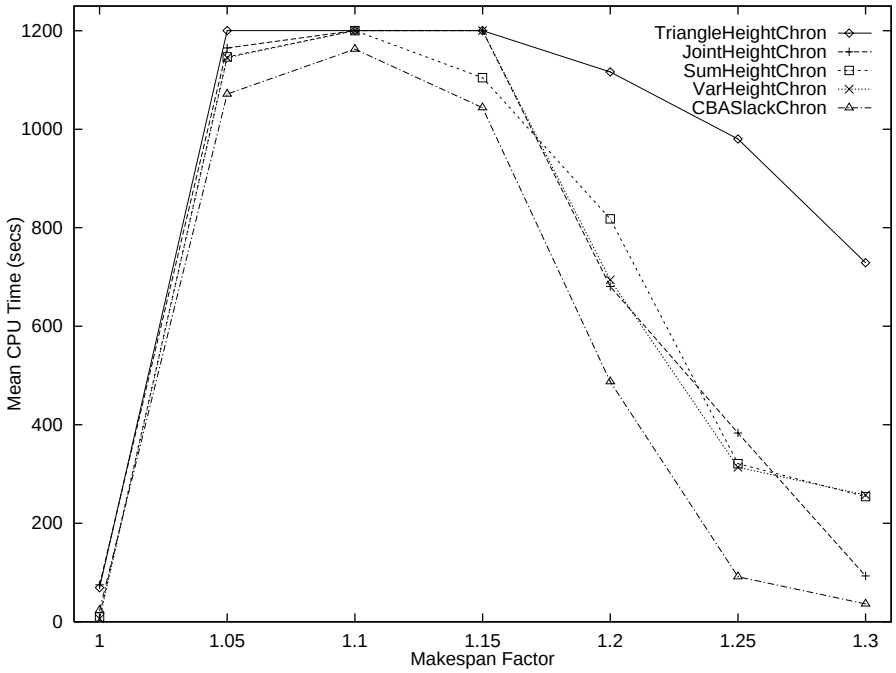


Figure 60. The Mean CPU Time in Seconds for the 20×20 Problems at Each Makespan Factor (Chronological Backtracking).

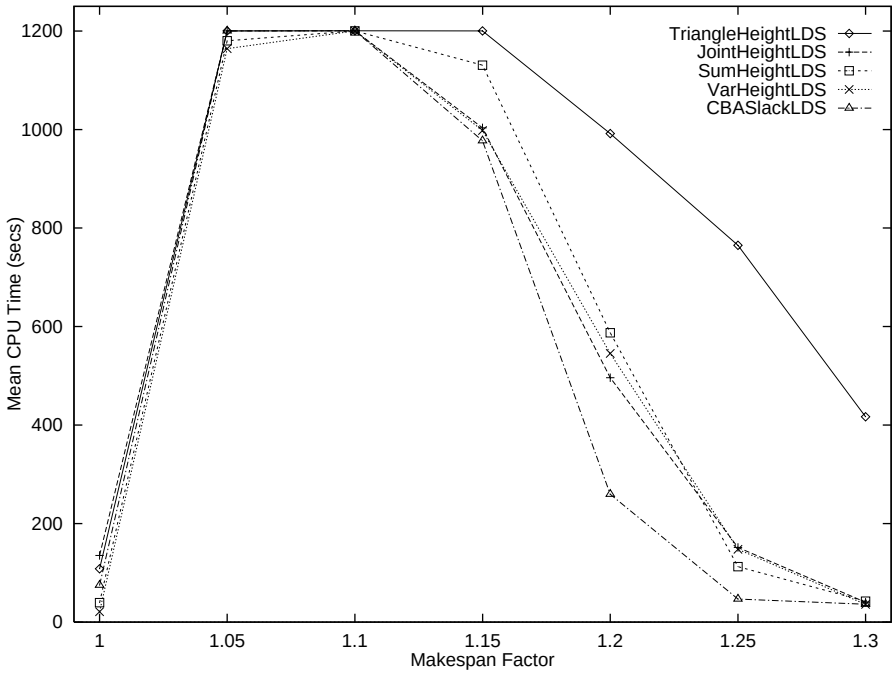


Figure 61. The Mean CPU Time in Seconds for the 20×20 Problems at Each Makespan Factor (LDS).

5.5.1.3 Other Search Performance Statistics

The other search performance statistics reflect the trends seen in the ones presented so far: CBASlack tends to be superior to all other algorithms, TriangleHeight tends to be inferior to all other algorithms, and LDS outperforms CBASlack. Exceptions to this trend are:

- Regardless of retraction technique, VarHeight, SumHeight, and JointHeight each use significantly fewer heuristic commitments than CBASlack.
- With LDS, JointHeight uses significantly ($p \leq 0.005$) fewer mean backtracks than CBASlack, SumHeight, and VarHeight. VarHeight uses significantly fewer mean backtracks than SumHeight ($p \leq 0.005$).
- For each heuristic, chronological backtracking makes significantly fewer commitments and significantly fewer heuristic commitments than the corresponding algorithms using LDS.

5.5.2 Summary

The results of Experiment 2 indicate that:

- CBASlack tends to outperform the texture-based heuristics regardless of retraction condition. Differences were slight or non-existent on the smaller problems, but were manifest in the larger problems.
- The algorithm based on the TriangleHeight texture estimation algorithm is inferior to all the other algorithms tested.
- There is little difference in performance among VarHeight, SumHeight, and JointHeight.
- Algorithms using LDS outperform their counterparts using chronological backtracking.

5.6 Experiment 3: Bottleneck Resources

The final experiment in this chapter is a repeat of Experiment 3 in Chapter 4 with the new heuristic commitment techniques (see Section 4.8).

5.6.1 Results

Complete results for the algorithms in Experiment 3 can be found in Appendix B, Section B.3.

5.6.1.1 10×10 Problems

The fraction of each problem set for which each algorithm timed-out can be found in Figure 62 (for the algorithms using chronological backtracking) and Figure 63 (for the algorithms using LDS). With chronological backtracking, SumHeight and JointHeight both time-out on significantly fewer problems ($p \leq 0.005$) than all other algorithms while there is no significant pair-wise difference. There are no significant differences among VarHeight, TriangleHeight, and CBASlack when chronological backtracking is used. With LDS, SumHeight times-out on significantly fewer problems ($p \leq 0.005$) than all other heuristics, including JointHeight, while TriangleHeight times-out on significantly more problems ($p \leq 0.005$) than all other heuristics. Of the remaining pairs of algorithms, JointHeight is superior to CBASlack ($p \leq 0.005$) while there are no differences between VarHeight and JointHeight or between VarHeight and CBASlack.

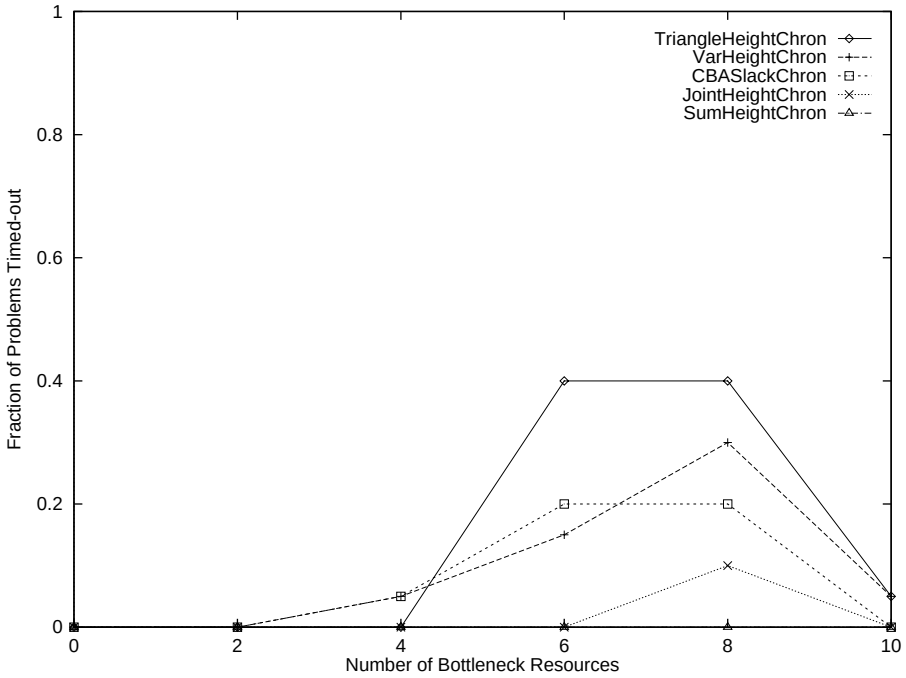


Figure 62. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (10×10 Problems – Chronological Backtracking).

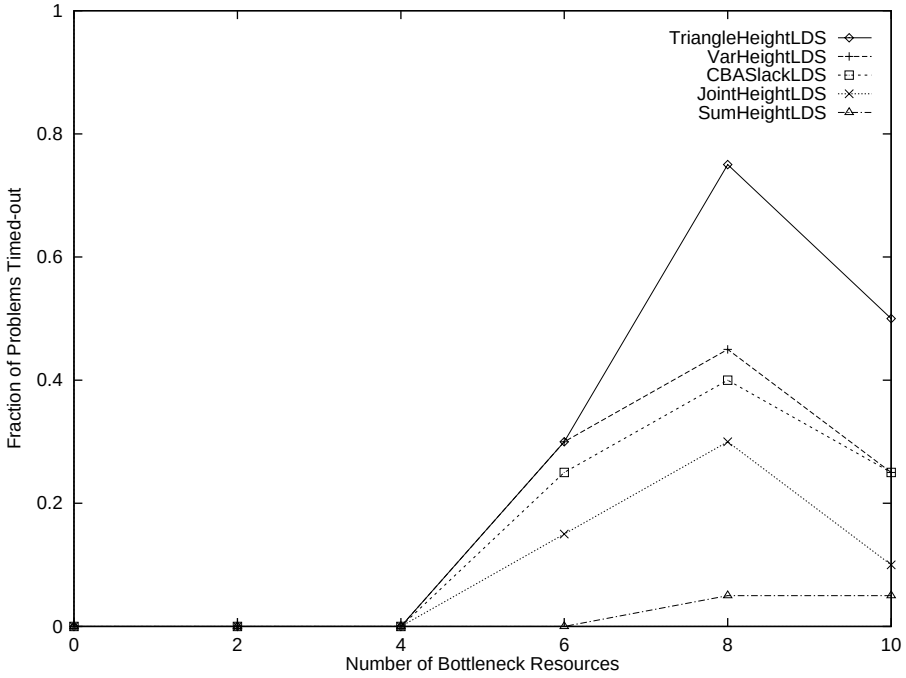


Figure 63. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (10×10 Problems – LDS).

The mean CPU time results for the 10×10 problems are displayed in Figure 64 for the algorithms using chronological backtracking and in Figure 65 for those using LDS. In both retraction conditions, SumHeight incurs a significantly lower mean CPU time than any other heuristic. JointHeight incurs the second lowest mean CPU time, significantly lower ($p \leq 0.005$) than all

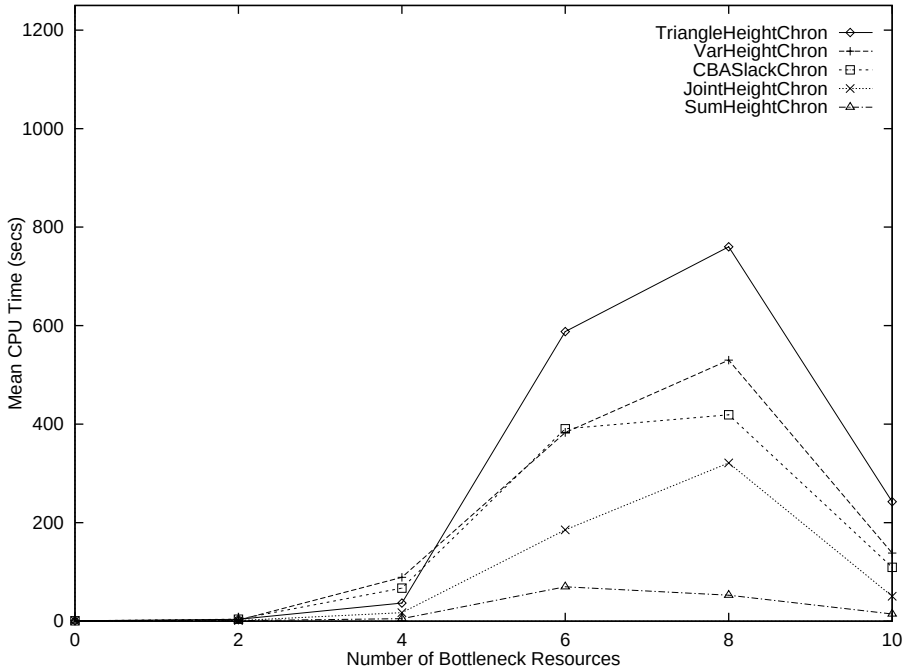


Figure 64. The Mean CPU Time in Seconds for Each Problem Set (10×10 Problems – Chronological Backtracking).

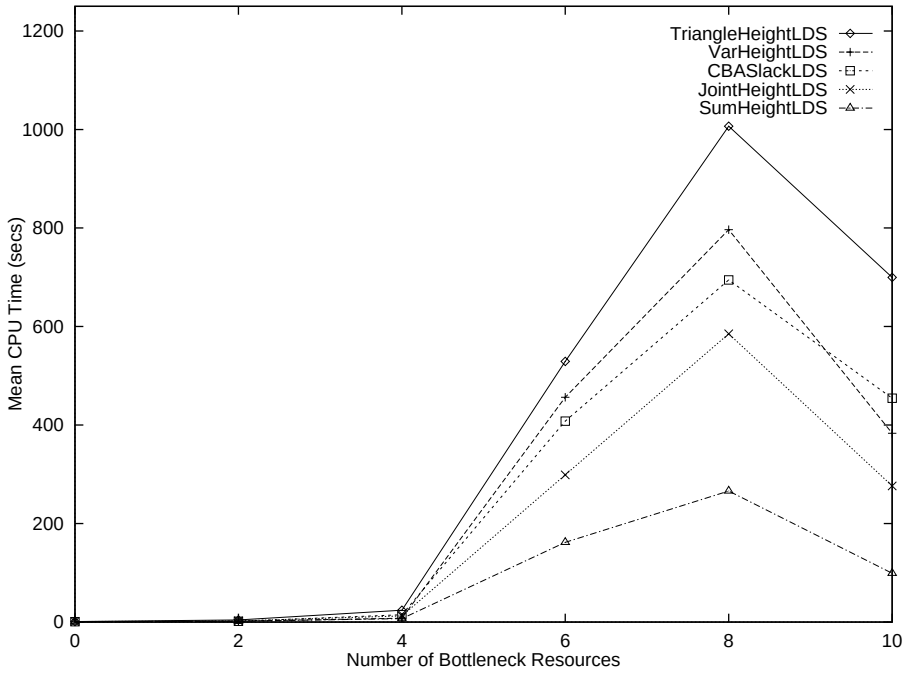


Figure 65. The Mean CPU Time in Seconds for Each Problem Set (10×10 Problems – LDS).

other heuristics (except SumHeight) with chronological backtracking and significantly lower ($p \leq 0.005$) than all other heuristics (except SumHeight and VarHeight) with LDS. Of the remaining algorithms, there are no significant differences between CBASlack and VarHeight, regardless of retraction condition; TriangleHeight incurs a significantly higher mean CPU time than CBASlack ($p \leq 0.001$), again regardless of retraction condition; and the only difference between

VarHeight and TriangleHeight is that the former incurs a significantly lower mean CPU time ($p \leq 0.005$) when using LDS.

Comparing the retraction conditions on the basis of time-outs indicates that the only significant difference is when TriangleHeight is used as the heuristic. In that condition, chronological backtracking times-out on significantly fewer problems ($p \leq 0.005$) than LDS. With mean CPU time, chronological backtracking incurs less mean CPU time ($p \leq 0.005$) than LDS with both JointHeight and TriangleHeight as well as with SumHeight and CBASlack as noted in Chapter 4.

The other search statistics indicate:

- In terms of the number of backtracks, the only significant differences are that SumHeight makes significantly fewer backtracks ($p \leq 0.0005$) than all other heuristics while JointHeight makes significantly fewer backtracks ($p \leq 0.0005$) than all other heuristics except SumHeight. These results hold regardless of retraction condition.
- Regardless of retraction condition, SumHeight makes significantly fewer overall commitments while TriangleHeight makes significantly more ($p \leq 0.0005$) than all other algorithms. With chronological backtracking, the only other difference is that JointHeight makes significantly fewer ($p \leq 0.001$) overall commitments than VarHeight. In the LDS condition, JointHeight makes fewer overall commitments than CBASlack and VarHeight ($p \leq 0.005$) while CBASlack makes significantly fewer commitments ($p \leq 0.005$) than VarHeight.
- Turning to the heuristic commitment results, we see that regardless of retraction condition, SumHeight makes significantly fewer than all other heuristics, while JointHeight makes significantly fewer than all other heuristics except SumHeight. There are no other significant differences among the heuristics.
- Finally, to evaluate the retraction techniques, we examine them when used with the VarHeight, TriangleHeight, and JointHeight heuristic commitment techniques.[4] There are no significant differences between chronological backtracking and LDS in terms of the number of backtracks. For the overall commitments, we observe that the chronological backtracking algorithms make significantly fewer than the LDS algorithms ($p \leq 0.0005$). For the heuristic commitment results, we see that the only significant differences are with TriangleHeight and JointHeight where the use of chronological backtracking results in significantly fewer heuristic commitments ($p \leq 0.0005$).

5.6.1.2 15×15 Problems

Figure 66 shows the number of problems timed-out in each problem set for each algorithm that uses chronological backtracking. The same data for the LDS algorithms is displayed in Figure 67.

SumHeight solves significantly more problems than any other heuristic in both retraction conditions. There are no other significant differences among the other heuristics, and no significant differences between chronological backtracking and LDS when the heuristic is held constant. Looking more closely at the individual problem sets, we see that LDS tends to outperform chronological backtracking on the problems with few bottlenecks while the opposite is true on problems with a large number of bottlenecks.

The mean CPU time for each algorithm and each problem set are shown in Figure 68 (chronological backtracking) and Figure 69 (LDS). These results also indicate that SumHeight performs sig-

4. For the statistical comparison of the retraction techniques with SumHeight and CBASlack see Section 4.8.2.1.

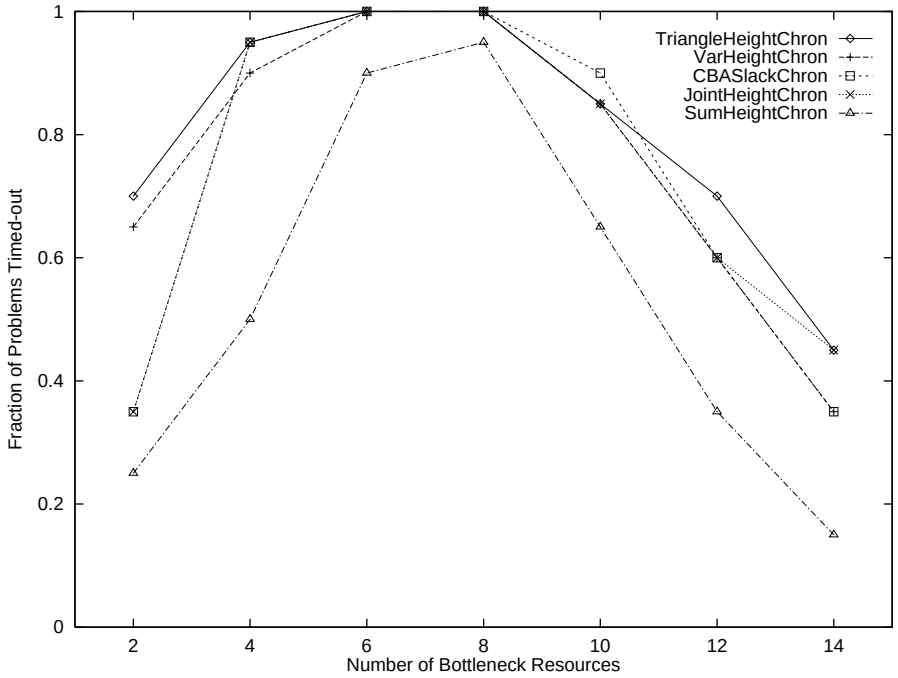


Figure 66. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (15×15 Problems – Chronological Backtracking).

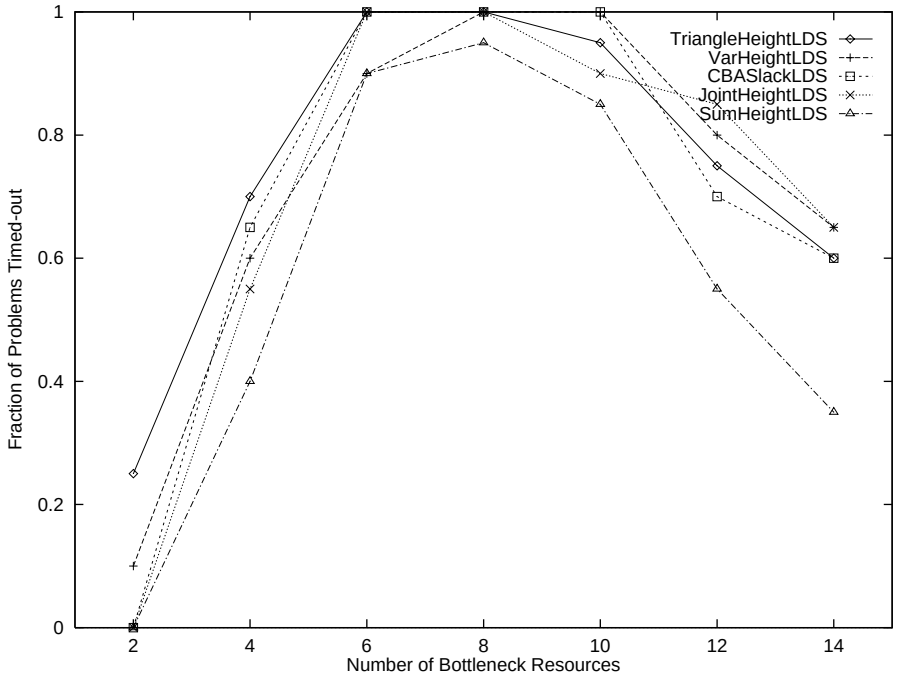


Figure 67. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (15×15 Problems – LDS).

nificantly better than each of the other algorithms regardless of retraction condition. There are no significant differences among the other heuristics using chronological backtracking; however, with LDS, TriangleHeight incurs a significantly higher ($p \leq 0.005$) mean CPU time than both CBASlack and JointHeight.

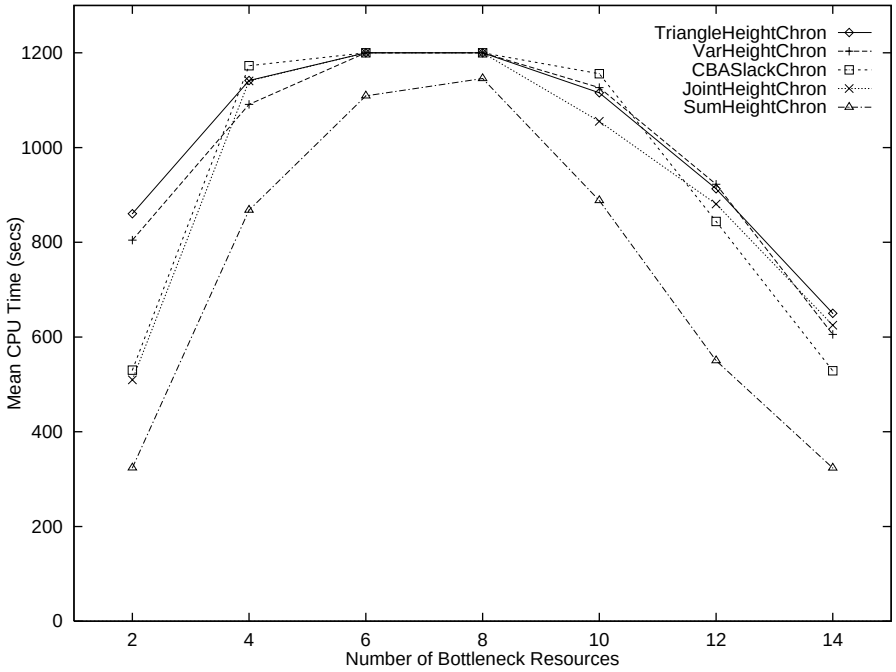


Figure 68. The Mean CPU Time in Seconds for Each Problem Set (15X15 Problems – Chronological Backtracking).

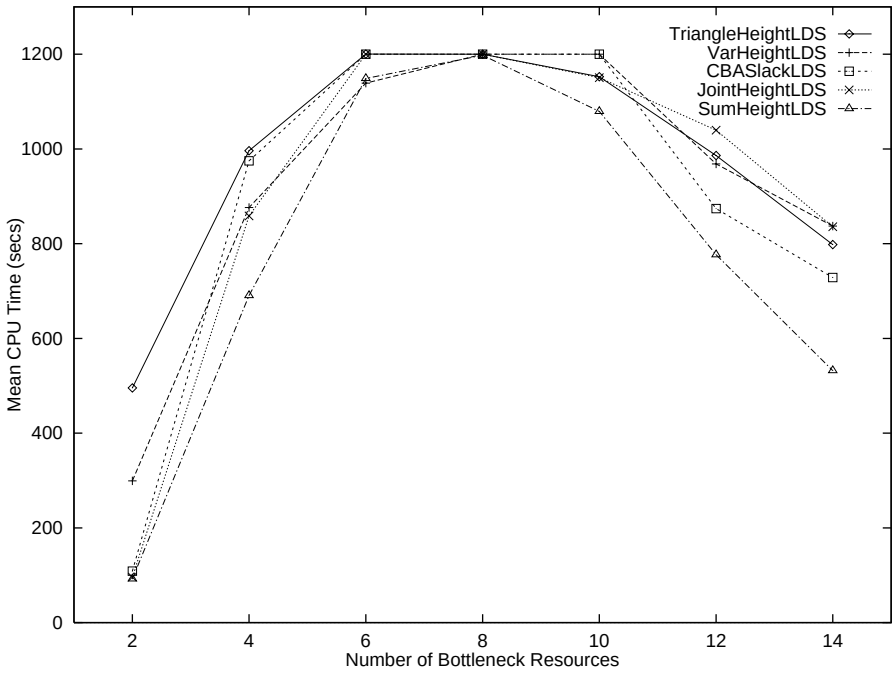


Figure 69. The Mean CPU Time in Seconds for Each Problem Set (15X15 Problems – LDS).

There are no overall differences in mean CPU time between chronological backtracking and LDS. As we observed in previous experiments, chronological backtracking incurs significantly more CPU time than LDS on the problem sets with few bottlenecks. In contrast, for problem sets with many bottlenecks, chronological backtracking incurs significantly less CPU time.

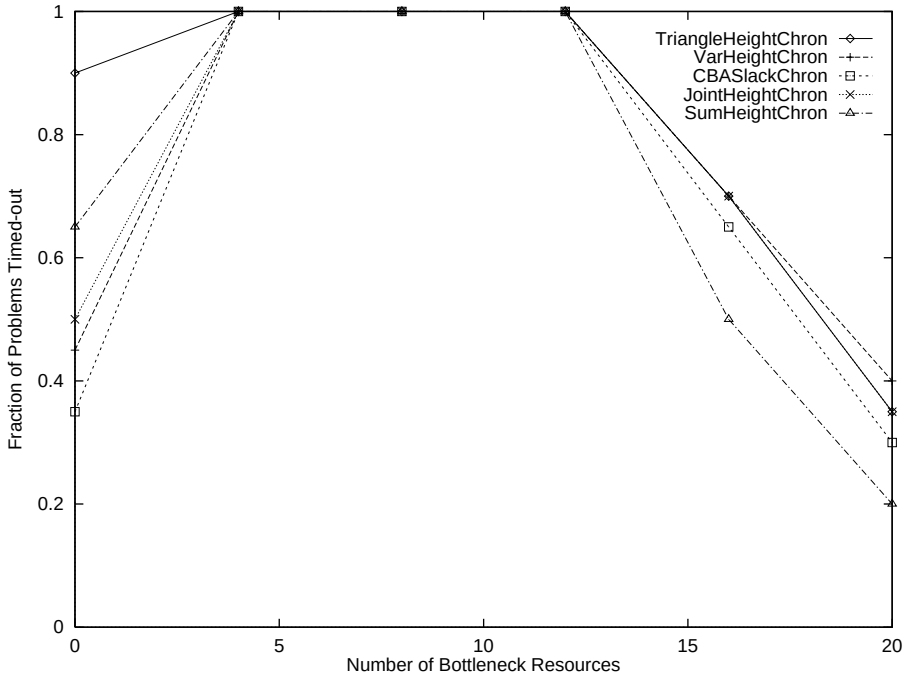


Figure 70. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (20×20 Problems – Chronological Backtracking).

The other search statistics tend to be consistent with the results that we have observed to this point: SumHeight performs better than all other heuristics, and while there is no overall difference among retraction techniques, LDS performs better on the problems with few bottlenecks and worse on the problems with many. Exceptions to these trends are:

- In terms of the number of backtracks, CBASlack makes significantly more backtracks than any of the other heuristics in both retraction conditions while SumHeight makes significantly more backtracks than JointHeight and TriangleHeight with LDS and significantly fewer than all other heuristics with chronological backtracking. All algorithms using chronological backtracking make significantly more backtracks than the corresponding algorithm using LDS.
- With chronological backtracking, SumHeight makes significantly fewer commitments than all other heuristics. CBASlack makes significantly fewer commitments than all other heuristics except SumHeight. With LDS the results are the same except for JointHeight which makes significantly fewer commitments than all other heuristics. The comparison of retraction techniques indicates that chronological backtracking algorithms make significantly fewer commitments than LDS algorithms.
- For heuristic commitments using chronological backtracking, CBASlack makes significantly more than all other heuristics while SumHeight makes significantly fewer ($p \leq 0.005$). For the LDS algorithms, this pattern repeats with the exception that JointHeight makes significantly fewer heuristic commitments than all heuristics but SumHeight. As with overall commitments, in terms of heuristic commitments, the LDS algorithms make significantly more than the chronological backtracking algorithms.

5.6.1.3 20×20 Problems

The fraction of problems in each problem set of the 20×20 problems for which each algorithm timed-out are shown in Figure 70 and Figure 71. The results for the algorithms using chronologi-

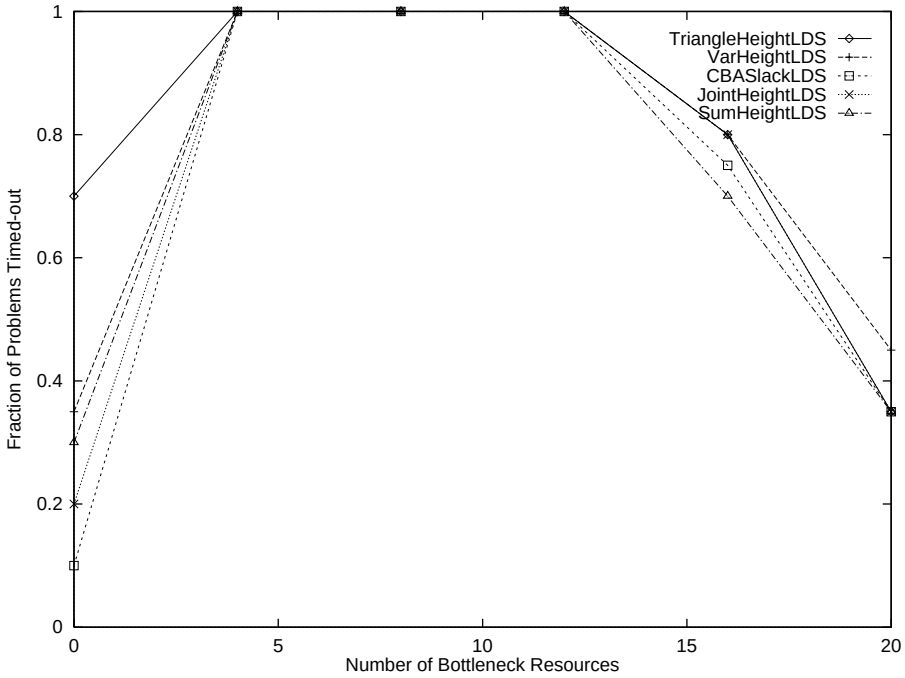


Figure 71. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out (20×20 Problems – LDS).

cal backtracking are displayed in the former figure and the results for the LDS algorithms are displayed in the latter. The only significant differences in these graphs surround the TriangleHeight heuristic commitment technique. When chronological backtracking is used, TriangleHeight times-out on significantly more problems ($p \leq 0.005$) than SumHeight, CBASlack, and JointHeight. In the LDS condition, TriangleHeight times-out on significantly more problems ($p \leq 0.005$) than CBASlack and JointHeight. There are no other significant differences in terms of the number of problems timed-out.

The mean CPU time results are displayed in Figure 72 for the chronological backtracking algorithms and in Figure 73 for the algorithms using LDS. Regardless of retraction technique, TriangleHeight incurs a higher mean CPU time ($p \leq 0.005$) than SumHeight and CBASlack. In addition, when LDS is used, CBASlack incurs a lower mean CPU time than JointHeight and VarHeight ($p \leq 0.005$). There are no other significant differences for mean CPU time.

There are no significant differences in either the fraction of problems timed-out or the mean CPU time between algorithms using chronological backtracking and algorithms using LDS.

The other search statistics indicate:

- For the number of backtracks, the only significant difference among heuristics using chronological backtracking is that both SumHeight and CBASlack make significantly more ($p \leq 0.0005$) backtracks than JointHeight. The results are more varied with LDS as we observe that JointHeight makes significantly fewer backtracks than all other heuristics. There is no significant difference between TriangleHeight and VarHeight; however, both incur significantly fewer backtracks than SumHeight and CBASlack ($p \leq 0.005$).
- In terms of the overall commitments, TriangleHeight makes significantly more commitments ($p \leq 0.0005$) than all other heuristics except VarHeight, while CBASlack makes significantly fewer commitments than all other heuristics. The only other significant difference in both retraction conditions is that JointHeight makes significantly fewer overall commitments than

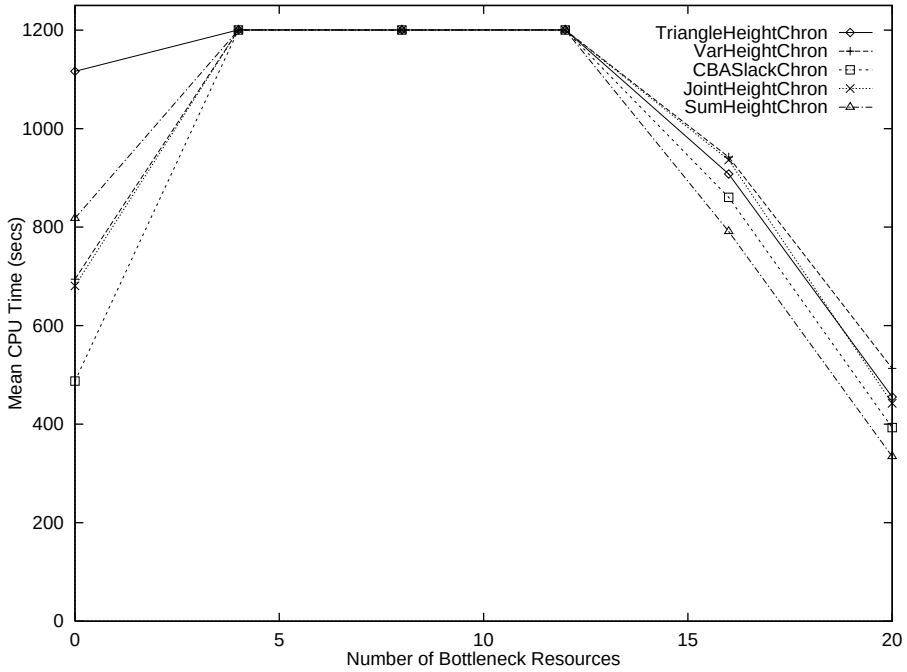


Figure 72. The Mean CPU Time in Seconds for Each Problem Set (20×20 Problems – Chronological Backtracking).

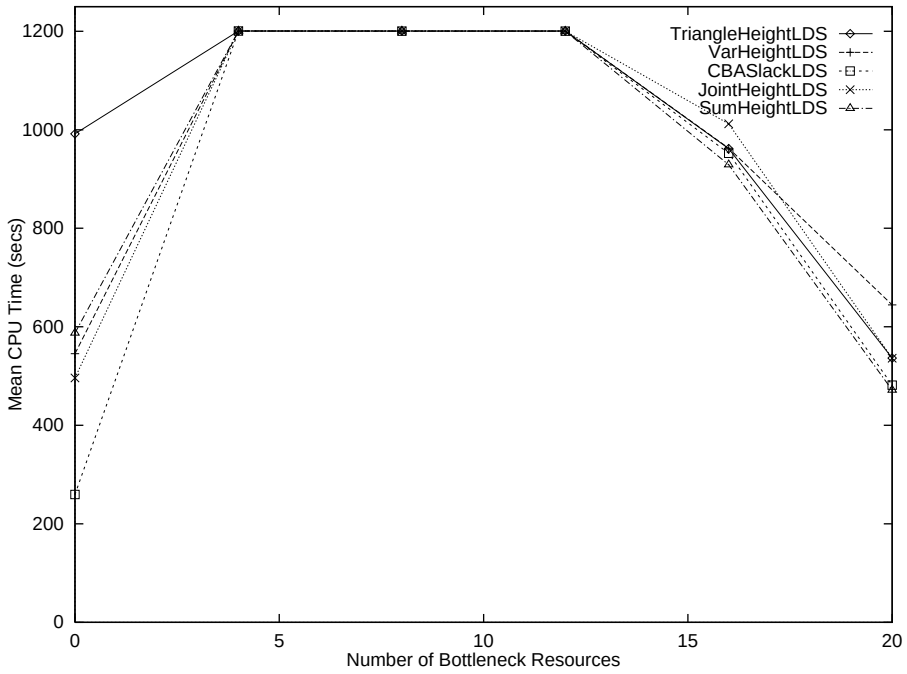


Figure 73. The Mean CPU Time in Seconds for Each Problem Set (20×20 Problems – LDS).

VarHeight ($p \leq 0.005$). In the LDS condition, in addition, the SumHeight makes significantly more overall commitments than JointHeight.

- Turning to the heuristic commitments, CBASlack makes significantly more than all other heuristics regardless of retraction condition. When LDS retraction is used, JointHeight makes

fewer heuristic commitments than all other heuristics, while with chronological backtracking JointHeight only makes fewer heuristic commitments than CBASlack and SumHeight.

- In comparing the retraction techniques in the context of VarHeight, TriangleHeight, and JointHeight,[5] we observe that each of the chronological backtracking algorithms make significantly more backtracks than the LDS algorithms, while making significantly fewer overall commitments and heuristic commitments.

5.6.2 Summary

Overall, Experiment 3 indicates:

- SumHeight is as good as or better than all other algorithms regardless of retraction condition. While JointHeight also performs well on the 10×10 problems, it is not significantly different on the larger problem sizes. There are few significant differences among the other heuristics.
- There is little overall difference among the retraction techniques. However, looking at the individual problem sets, LDS tends to outperform chronological backtracking on those sets with few bottlenecks while the reverse is true for the sets with many bottlenecks.

5.7 Discussion

The central purpose of the experiments in this chapter is to compare heuristics built around probability of breakage estimations to each other and to the top heuristics from Chapter 4 in the context of job shop scheduling. We, therefore, focus discussion on these heuristics.[6]

5.7.1 The Practical Utility of the Probability of Breakage

Our criterion of the practical utility requires that estimations of a measure of criticality should result in heuristic search techniques that perform as well as or better than existing techniques. In Experiments 1 and 2, both VarHeight and JointHeight perform as well as SumHeight. However, SumHeight outperforms each of the estimates of probability of breakage in Experiment 3. Overall, then, in the context of job shop scheduling, the VarHeight and JointHeight estimates for the probability of breakage criticality measurement partially satisfy the requirement of practical utility.

A further critical aspect of the practical utility of the probability of breakage measurement is the context in which it is applied. The experiments in this chapter speak only to the practical utility of the probability of breakage measure in job shop scheduling. Given that the probability of breakage measure meets our first three requirements and so can be applied more readily beyond job shop scheduling, such an application must be done to further evaluate its practical utility. We turn to such an application, in the next chapter.

---

5. See Section 4.8.2.3 for a comparison of the retraction techniques in the context of SumHeight and CBASlack.
6. A secondary purpose, the comparison of retraction techniques, showed that LDS performs either as well as or better than chronological backtracking. The only exception to this is when a problem set contains overconstrained problems, in which case the larger search space of LDS results in poorer results.

5.7.2 Probability of Breakage versus Aggregate Demand

One explanation of the results of Experiment 3 is that VarHeight and JointHeight are not as sensitive to resource-level non-uniformities as SumHeight. In terms of the underlying measures of criticality, this may imply that there is structural information that is captured by aggregate demand that is not identified by the probability of breakage. For example, aggregate demand uses the magnitude of a resource level in its measure of criticality. In the exact calculation of aggregate demand (Section 5.1.2), a state where a unary resource and time point have a demand of 20 has more impact on the aggregate demand curve than a state where the demand is two. In the probability of breakage calculation, such states are equally weighted: the constraint is broken in both.

The existence of structural information captured by aggregate demand, but not by the probability of breakage, is a speculative point. Other factors such as the accuracy and computational complexity of estimation algorithms, and how well a measurement of criticality correlates with true criticality likely play complex and interdependent roles in the overall heuristic search performance. The unraveling of such factors, together with the deeper analysis of the information examined by a measure of criticality and estimated by a texture measurement, remain key areas of future work.

5.7.3 Estimations of Probability of Breakage

Three estimations of probability of breakage were presented in this chapter: JointHeight, TriangleHeight, and VarHeight. Of the three, we believe that JointHeight is the most accurate estimator of the probability of breakage as its formulation does not require the assumptions made for TriangleHeight and VarHeight. If it is true that JointHeight is a better estimator, we would expect algorithms using JointHeight to outperform those using the other probability of breakage estimators.

Our experimental results indicate that TriangleHeight tends to perform worse than JointHeight, while VarHeight performs about the same. This supports our contention that, as an estimator of probability of breakage, TriangleHeight is inferior to JointHeight. The VarHeight results suggest that either VarHeight is as good an estimator as JointHeight or that the lower computation complexity of VarHeight makes up for the lower accuracy in estimation vis-a-vis JointHeight.

We have no independent evidence of the accuracy of the probability of breakage estimation algorithms. Evidence could be found by an exhaustive calculation of the probability of breakage of each constraint (in necessarily small problems) and an analysis of how the actual values are predicted by each estimation algorithm. Such an analysis remains for future work.

5.8 Conclusions

In this chapter, we examined the creation of a general notion of constraint criticality as well as measurements of such criticality. While the use of contention/aggregate demand has been shown to provide a strong practical basis for such a measure, particular aspects of the aggregate demand prevent easy generalization. We define requirements for a measure of the criticality of a constraints and suggest the use of the probability of breakage of a constraint as such a measure. We created three estimation techniques for probability of breakage and evaluated each on job shop scheduling problems. The justification for using the job shop problems first, before attempting to use the wider applicability of the new texture measurement estimation techniques, was to assess whether the extension of contention weakened the power of the texture measurements as a basis for heuristic commitment techniques.

Our empirical results indicate that while on many problems at least two of the new techniques (VarHeight and JointHeight) were competitive with the contention-based heuristic commitment technique (SumHeight), on the problem sets where SumHeight seemed best able to exploit the problem structure, the probability of breakage estimation techniques are unable to perform as well. While we have ideas to account for these results, they remain at the level of hypotheses that question basic assumptions with respect to texture measurements and deeper questions surrounding heuristic search (see for example the discussion in Section 4.9.1.2).

Chapter 6 Scheduling with Inventory

In this chapter, we examine the application of constraint-directed scheduling technology to the inventory scheduling problem. Our focus continues to be the use of texture-based heuristic commitment techniques and so a key part of the work in this chapter surrounds the extension of texture measurements to inventory constraints. This extension is achieved by the formulation of the individual impact of an activity on the probabilistic inventory level followed by the aggregation of individual impacts on each inventory and resource. The aggregate curves are used to estimate the probability of breakage of both resource and inventory constraints, and to identify the most critical constraint and time point. Once the most critical constraint is identified, we generate and post a heuristic commitment to reduce the criticality. In order to include inventory constraints within the scope of constraint-directed scheduling techniques, it is necessary to define their representation and propagation characteristics. The second component of this chapter, therefore, is a presentation of the inventory constraint representation together with the creation of propagation techniques.

The chapter is organized as follows: in the following section we define the inventory scheduling problems addressed in this chapter and provide an overview of our approach to this problem. We then present our inventory representation, including algorithms for the calculation of upper and lower bounds on inventory levels. Section 6.3 turns to the commitments that are made on inventories during search, and Section 6.4 details the extension of texture measurements to the inventory minimum and maximum constraints. Two inventory propagators are described in Section 6.5, and the scheduling strategies with which we experiment are presented in Section 6.6. Our empirical studies and results compose Section 6.7 through Section 6.11. Finally, we discuss our results and conclude the chapter in Section 6.12 and Section 6.13 respectively.

6.1 Introduction

The management of inventory, its storage, production, and consumption, represents the core function of a manufacturing organization, be it a diversified global manufacturing enterprise or a single factory work-center. Manufacturing is primarily concerned with the transformation of raw materials into finished goods. The economic viability of a manufacturer depends on the efficiency with which this transformation can be achieved.

There are a wide variety of constraints concerning inventory, only a subset of which will be addressed in this chapter. In general, the inventory scheduling problem is to produce some amount of finished goods, given a set of process plans and a schedule of raw material supply events. The activities in a process plan may produce and consume inventory at varying rates and in different amounts. There may be multiple process plans that produce the same inventory. The solver, there-

fore, must instantiate the correct mix of process plans and schedule production and consumption activities to meet the demands while satisfying the inventory and resource constraints. In addition to these constraints, we may have some of the following:[1]

- compatibility constraints between inventories specifying that they may not share a storage facility or plant location.
- spoilage and curing constraints specifying maximum and minimum lengths of time that an inventory can exist before being used in a subsequent manufacturing step.
- multiple storage facilities with specific spatial relations to other storage and machine resources (*e.g.*, $\texttt{tank}_4$ may only feed $\texttt{mixer}_1$ or $\texttt{mixer}_8$).
- capacity constraints that vary over the scheduling horizon.

6.1.1 Motivation and Problem Definition

In manufacturing, activities may produce and/or consume inventory that must be stored before consumption and after production, subject to minimum and maximum limits on the amount of each inventory that can be stored at any time. Scheduling, then, must take into account not only the temporal and resource constraints, but also these minimum and maximum inventory constraints.

An inventory problem can be defined as follows, given:

- A set of *process plans*. Each process plan defines a temporally connected set of activities for the production of a particular inventory or set of inventories. Each activity has a duration, resource requirements, and, perhaps, inventory requirements.
- A set of *resources*. Each resource has maximum capacity specifying the number of activities that can simultaneously execute on the resource. This maximum may vary across the scheduling horizon.
- A set of *inventories*. Each inventory has both a minimum and maximum storage capacity, specifying, respectively, the smallest and largest amount of the inventory that can exist at any time. The capacity constraints may vary, independently, across the scheduling horizon with the only requirement being that the minimum can not be greater than the maximum at any time point. A non-zero amount of each inventory may be present at the beginning of the scheduling horizon.
- A set of *supply and demand event*s. Each event indicates a time interval during which an external force will instantaneously add (supply) a specific amount of inventory to or remove (demand) it from the plant.

Find:

- An instantiation of process plans and an assignment of start times and resources to each instantiated activity such that all resource, inventory, and temporal constraints are satisfied. It is, in addition, desirable to minimize inventory levels (while satisfying the minimum constraints).

The inventory problem encompasses the Generalized Resource Constrained Project Scheduling Problem [Herroelen and Demeulemeester, 1995] while adding inventory production and consumption, inventory constraints, and the need to instantiate the to-be-scheduled activities.

---

1. This list of constraints should in no way be interpreted as an exhaustive description of the inventory-related constraints found in manufacturing organizations. Its purpose is to provide a general sense of the issues surrounding inventory scheduling.

6.1.1.1 Restrictions on the Problem in this Chapter

The above description of inventory scheduling contains many characteristics that have not been systematically addressed in the scheduling literature. Indeed, the necessity of instantiating process plans blurs the line between scheduling and planning (see Chapter 7). The problem model addressed in this dissertation will therefore be simpler than the above description. We have taken the job shop scheduling model as our starting place, and have defined a problem model such that inventory production, consumption, and storage are added to the job shop.

The problem consists of a set of jobs each of which consists of a set of activities. The first activity in the job consumes the raw material inventory corresponding to the job, and the final activity produces the finished goods inventory corresponding to the job. Therefore, we have in the model, one raw material inventory and one finished goods inventory for each job. If we were to place the supply of all raw materials at time 0 and the demand for all finished goods at the end of the horizon, we would have a job shop problem. We add to this job shop model the following characteristics:

- Each activity in a job may consume one or more of the raw material inventories in the problem. The number of consumptions in a job is an independent variable and can range from 1 (in the job shop model) to the product of the number of activities and the number of raw materials. The latter extreme is formed when all activities consume all raw materials.
- Each production and consumption is instantaneous: production occurs at the end time of the activity, consumption at the start time.
- There are multiple supply events for each raw material. These events are spread over the scheduling horizon.
- There is a minimum constraint on each inventory. This constraint is constant across the scheduling horizon.[2]
- There is a maximum constraint on each inventory. This constraint is constant across the scheduling horizon.

6.1.2 Overview of Approach

The approach to inventory scheduling in this dissertation is to represent the minimum and maximum inventory constraints so that heuristics based on texture measurements can be applied. The goal, from a heuristic perspective, is to be able to estimate the probability of breakage of all resource and inventory constraints in order to identify the most critical constraint overall. We then focus on that constraint and make a commitment to reduce its criticality. If the constraint is a resource constraint, we will use the sequencing heuristics presented in Chapter 4 (see Section 4.3.2.2). If the critical constraint is an inventory constraint we assert an inventory commitment, discussed in Section 6.3.

Propagators are also a critical component of a scheduling algorithm and we define two novel propagators for inventory constraints. These propagators, respectively, enable the inference of new temporal constraints on activities that produce and consume inventory, and allow inference that a particular consumer must consume inventory produced by a particular producer (see Section 6.5).

2. See Section 6.12.4 for a discussion of relaxing this assumption.

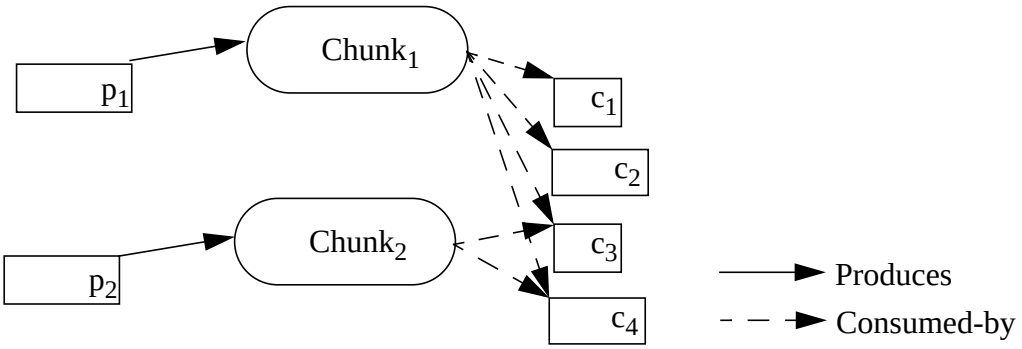


Figure 74. Example InventoryChunks, Their Producers, and Their Consumers.

6.2 Inventory Representation

Our inventory representation is a generalization of the resource representation used in the previous chapters. Each activity has one or more inventory requirements represented as a variable that takes the value of an inventory. For each requirement, there is an amount variable corresponding to the amount of the inventory that the activity produces or consumes. Consumption is assumed to take place instantaneously at the start of an activity while production occurs instantaneously at the end of an activity. The inventory constraints are represented analogously to resource capacity constraints. Each inventory defines a maximum constraint corresponding to the highest inventory level allowable at any time point, and a minimum constraint corresponding to the lowest inventory level allowable. Unlike the cumulative constraint representation, we do not represent the inventory usage as activities with duration (see Section 2.6.2).

InventoryChunks. A key piece of the inventory representation is the explicit modeling of InventoryChunks. These data structures represent the physical inventory that is produced by a single producer and consumed by a set of consumers. The size of an InventoryChunk is represented by a domain variable constrained to be equal to the amount produced by the producer and greater-than-or-equal-to the sum of the amounts consumed by the consumers.

When a producing activity is instantiated, a corresponding InventoryChunk is also created on the inventory which is produced. The consuming activities are not, initially, linked to any InventoryChunk. As shown in Figure 74, while a producer is limited to producing a single InventoryChunk (of any one type of inventory), a consumer can consume from multiple InventoryChunks, and an InventoryChunk can be consumed by multiple consumers. Therefore, at any point in the search, a consumer may be consuming some portion of its input from one InventoryChunk, ic_1, another portion from another InventoryChunk, ic_2, and not yet have the rest of its consumption linked with any InventoryChunk. We refer to consumers with some portion of their consumption not yet linked to an InventoryChunk as *loose consumers*.

In the balance of this section, we present the calculation of the upper and lower bounds on inventory level. The bounds are used as part of the procedures for inventory termination, inventory dead-end detection, and inventory bound propagation. After presentation of the bound calculation, we present the termination criteria for inventory.

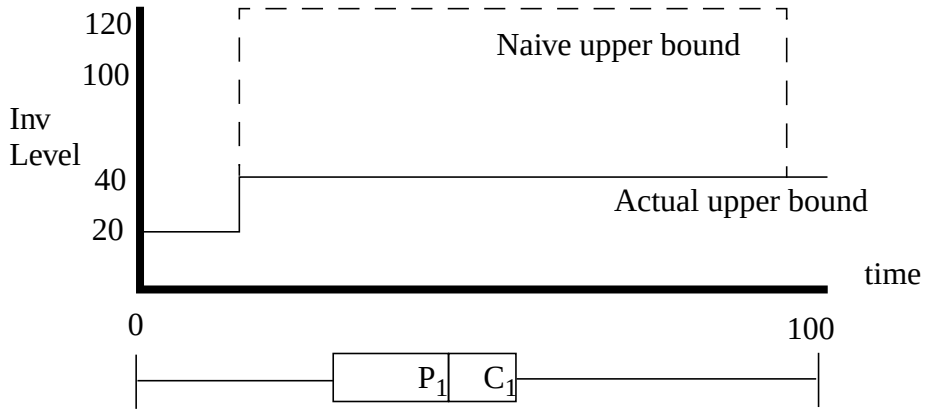


Figure 75. The Naive Upper Bound and the Actual Upper Bound When a Producer and Consumer are Linked with a Meets Constraint.

6.2.1 Calculating the Inventory Bounds

The criteria for an upper bound on the inventory level is that, at each time point, the upper bound must be equal to or greater than the maximum possible inventory level given any assignment of start times to the activities on the inventory. Similarly, for the lower bound it is necessary to guarantee that, at each time point, the lower bound is less than or equal to the minimum possible inventory level. In particular, note that it is possible for the lower bound to be negative in the situation where consumption may out-strip production.

6.2.1.1 Calculating the Upper Bound

A naive upper bound on the inventory level can be found by assuming all the production activities execute at their earliest possible start times and all the consumption activities execute at their latest possible start times. In the presence of “meets” constraints, such a calculation can be significantly strengthened. With a meets constraint between a producer and a consumer, some amount of the inventory is produced and consumed instantaneously. For example, as shown in Figure 75, with a starting inventory of 20 units, one producer contributing 100 units, one consumer consuming 80 units, and a meets constraint between the producer and consumer, the naive upper bound is much higher than the actual inventory level.

We represent the upper bound curve with a list of events, ordered by time. This representation is the same as that used for the texture measurements curves (Section 4.3.3). Each event is composed of an amount and a time point. For consumption events the amount is negative while for production events, the amount is positive. The events on the upper bound curve come from two sources: loose consumers and InventoryChunks.

Loose Consumers. If a consumer has some portion of the amount it consumes that is not yet linked to any InventoryChunk, then, to calculate the upper bound, the amount that is unlinked is assumed to be consumed at the latest start time of the consuming activity. The inventory event that is placed on the upper bound curve is, therefore, a consumption event with the amount equal to the unlinked amount of the consumer, and the time equal to the latest start time of the consumer. Note that it is only the amount of consumption which is unattached that is assumed to be consumed at the latest start time. If a consumer also has some of its consumption linked to an InventoryChunk, that portion is treated independently.

```
1:  for each InventoryChunk of Inventory I
2:     producerAmt = total amount of I produced by the producer
3:     for each consumer of the InventoryChunk
4:        if consumer meets the producer
5:           producerAmt -= amount of InventoryChunk consumed by the consumer
6:        else
7:           new-event.time = consumer.lst
8:           new-event.amount = -amount of InventoryChunk consumed by the con-
               sumer
9:           event-list.insert(new-event)
10:
11:    producer-event.time = producer.eft
12:    producer-event.amount = producerAmt
13:    event-list.insert(producer-event)
14:
15: for each loose consumer on Inventory I
16:    new-event.time = consumer.lst
17:    new-event.amount = -amount of consumer not linked to any InventoryChunk
18:    event-list.insert(new-event)
```

Figure 76. Pseudo-code for the Calculation of the Upper Bound on the Inventory Level for Inventory *I*.

InventoryChunks. An InventoryChunk maintains links to a single producer and a set of consumers. As part of the producer/consumer commitment (see Section 6.3.1), a temporal constraint is used to link a producer and consumer that share an InventoryChunk. If all temporal constraints among the producer and consumers sharing an InventoryChunk are precedence constraints, then the upper bound events consist of one event per activity. For the producer, the event is a production event with time point equal to the earliest finish time of the activity and amount equal to the total amount of inventory produced by the activity. Each consumer on the InventoryChunk has an event with an amount equal to the amount of the InventoryChunk that is consumed and a time point equal to the latest start time of the consumer.

If some of the temporal constraints between the producer and some consumers in an InventoryChunk are meets constraints, the calculation of the upper bound contribution from the InventoryChunk is slightly different. All consumers with precedence constraints still contribute as described above. The difference is that the producer event and the events from the consumers that meet the producer are merged into a single event. The event has a time point equal to the earliest finish time of the producer and an amount equal to the difference between the amount produced and the sum of the amounts consumed by those consumers constrained to meet the producer. This special treatment of meets constraints achieves a much tighter upper bound as illustrated in Figure 75. Note that if a consumer is linked to multiple InventoryChunks, it is treated independently on each InventoryChunk.

The pseudo-code for the upper bound calculation is displayed in Figure 76. After all events for an inventory are added to the (sorted) event list, an algorithm runs through the list and calculates the aggregate upper bound curve as is done with the texture measurements curves (Section 4.3.3).

Complexity. The worst case complexity for calculation of the upper bound occurs when all consumers are attached to all InventoryChunks and each have some unlinked consumption. In such a case the loop starting at line 1 has an $O(n^2)$ time-complexity and the loop starting at line 15 has an

$O(n)$ time-complexity (with n equal to the number of activities on an inventory). The overall worst-case complexity, however, comes from the sorting of the list of events. In our worst-case scenario there can be $O(n^2)$ events on this list: each producer has one event and each consumer has one event for each InventoryChunk. The sorting of the event list therefore incurs a worst-case time complexity of $O(n^2 \log n)$. In practice, the average complexity can be reduced by caching information and incrementally updating the bound, as is done in ODO. Also, as we demonstrate in Section 6.3.1 our heuristic commitment technique will tend to minimize the number of InventoryChunks to which a single consumer is linked.

6.2.1.2 Calculating the Lower Bound

As with the upper bound calculation, a naive approach to the lower bound would result in a much looser bound than is possible using the InventoryChunks. The naive calculation would represent all consumers as consuming at their earliest start time and all producers producing at their latest finish time. This ignores the temporal constraints due to inventory commitments between a producer and consumers. Our implementation of the lower bound calculation, therefore, follows the upper bound calculation structure. Again, the contributors for the lower bound inventory events are the loose consumers and the InventoryChunks.

Loose Consumers. The loose consumers also contribute events to the lower bound. The event has a time equal to the earliest start time of the consumer and an amount equal to the amount of the consumer that has not yet been linked to an InventoryChunk.

InventoryChunks. An InventoryChunk will never have a set of consumers whose amounts sum to greater than the amount produced by the producer. Therefore, the lower bound contribution from an InventoryChunk can never be less than 0. Given a producer and a set of consumers on the same InventoryChunk, the producer and each consumer will be temporally linked with at least a precedence constraint. Therefore, while the producer may have a window of possible start times, it is not possible for any of the consumers to occur before the producer. The lower bound contribution from an InventoryChunk up to (but not including) the latest finish time of the producer must be 0. It can not be lower, as no consumer can execute before the producer, and it can not be higher, as the producer can, by definition, produce at its latest finish time. Therefore, the initial lower bound event from an InventoryChunk has a time equal to the latest finish time of the producer. The amount of this event is the difference between the amount produced by the producer and the sum of the amounts from all consumers that can execute at the latest finish time of the producer. The rest of the consumers each contribute events at their earliest start times with amounts equal to the amount of the inventory that they consume.

Figure 77 presents the pseudo-code for the lower bound calculation. As with the upper bound, we do not show that the event list is then run through to complete the aggregate lower bound curve.

Complexity. By the same arguments applied to the upper bound calculation, the worst-case complexity for the lower bound is incurred by the sorting and maintenance of the event-list. As with the upper bound calculation, this time-complexity is $O(n^2 \log n)$.

6.2.2 Inventory Termination

One of the requirements of the inventory constraint representation is a set of termination criteria. The inventory termination criteria are conditions under which it can be guaranteed that the constraints on an inventory will be satisfied in all subsequent search states.

```
1:  for each InventoryChunk of Inventory I
2:     producerAmt = total amount of I produced by the producer
3:     for each consumer of the InventoryChunk
4:        if consumer.est <= producer.lft
5:           producerAmt -= amount of InventoryChunk consumed by the consumer
6:        else
7:           new-event.time = consumer.est
8:           new-event.amount = -amount of InventoryChunk consumed by the con-
               sumer
9:           event-list.insert(new-event)
10:
11:    producer-event.time = producer.lft
12:    producer-event.amount = producerAmt
13:    event-list.insert(producer-event)
14:
15: for each loose consumer on Inventory I
16:    new-event.time = consumer.est
17:    new-event.amount = -amount of consumer not linked to any InventoryChunk
18:    event-list.insert(new-event)
```

Figure 77. Pseudo-code for the Calculation of the Lower Bound on the Inventory Level for Inventory *I*.

The inventory termination in ODO uses the upper and lower bounds to evaluate two conditions:

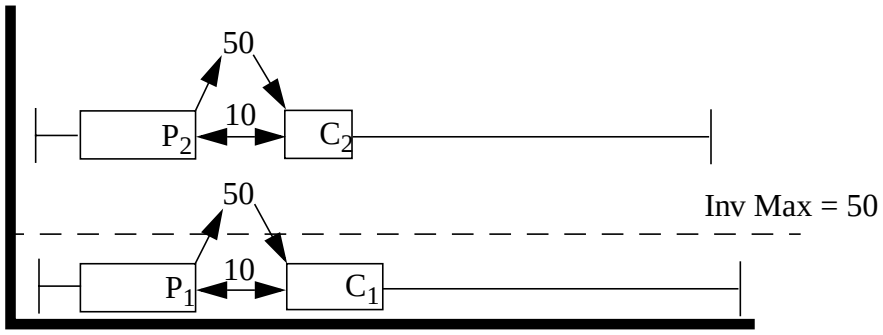
1. Is the lower bound greater than or equal to the minimum constraint at all time points?
2. Is the upper bound less than or equal to the maximum constraints at all time points?

If both conditions are satisfied on an inventory, then no further commitments are required to guarantee that the inventory constraints are satisfied.

Note that this use of the inventory bounds allows us to pursue a least commitment approach. It is not necessarily the case that we need to assign start times to all producer and consumer activities. We simply have to satisfy the termination criteria, which may leave significant windows of start times for the activities.

6.3 Inventory Commitments

As remarked in Chapter 2, previous work on inventory scheduling uses common scheduling commitments (*e.g.*, start time assignment) and depends on propagation to detect dead-ends and prune alternatives for inventory activities. With the ability to detect a critical inventory (described in Section 6.4), however, it is not clear that start time assignments address the root of the criticality. For example, if a minimum inventory constraint is critical, we could find a producer and assign it to its earliest start time. Conversely, we could assign a consumer to its latest start time. However, it is not the start times of the activities that are at issue. The minimum constraint is critical because there is a danger that some consumer will be scheduled such that no producer can occur before it to supply its required inventory. The problem is not the start times, but the lack of a commitment ensuring the ordering of a producer-consumer pair. The commitment, then, could consist of a precedence constraint specifying that the producer will occur before the consumer as well as a constraint specifying that some amount of the inventory produced by the producer is consumed by the consumer.



Existing commitments: $(P_1 \rightarrow C_1, 50)$, $(P_2 \rightarrow C_2, 50)$,
$et_{P1} + 10 = st_{C1}$, $et_{P2} + 10 = st_{C2}$

Figure 78. An Example of the Need to Assign Some Start Times to Ensure Satisfaction of Inventory Constraints.

Such a commitment is consistent with our least commitment approach: producers can be matched to multiple consumers and consumers can be matched to multiple producers and no start times need be assigned. However, simply matching consumers with producers does not guarantee that the termination criteria for an inventory (or a dead-end) will be met. It may be the case that we have to further constrain the linked producer-consumer pairs to execute within a certain time interval of each other. For example, with an inventory maximum constraint of 0, each producer must end precisely at the start of each of its consumers. In the worst case, in fact, it may be necessary to actually assign start times to some subset of activities in order to guarantee the satisfaction of inventory constraints. Such a case is displayed in Figure 78. Both producer-consumer pairs must execute 10 time units apart (perhaps because of resource commitments); however, they must be further constrained to ensure the inventory maximum constraint is satisfied.

To address such situations in a least commitment approach, we propose three types of commitments to reduce inventory criticality: producer/consumer commitments, producer/consumer interval commitments, and start time commitments.[3]

6.3.1 Producer/Consumer Commitments

We use the notation ($P \rightarrow C$, 25) to indicate a producer/consumer commitment between *P* and *C*. Specifically, this notation indicates that *C* must start at or after the end of *P* and that 25 units of the inventory produced by *P* are consumed by *C*. *P* is not constrained to *only* produce 25 units of inventory, nor is *C* constrained only to consume 25 units: each may be linked with other consumers and producers, respectively. When all of the inventory that is produced (respectively, consumed) by an activity has been matched to a corresponding consumer (respectively, producer) via producer/consumer constraints, we say that the activity's inventory has been *completely matched*. Note that in a solution, a producer does not necessarily have to be completely matched since some of the inventory it produces may remain in storage at the end of the scheduling horizon. A consumer, however, must be completely matched: there is no way to store negative inventory levels.[4]

---

3. See Section 6.6 for a description of the entire heuristic commitment technique that mixes resource and inventory commitments based on texture measurements.

4. Starting inventory is represented as being produced by an activity with a duration of 0 and a start time of 0. Therefore, even if there is starting inventory, all consumers must be completely matched in a solution.

The scheme adopted here is that in a search state, *S*, where an inventory, *I*, is critical and there exists a consumer, *c*, of the inventory that has not been completely matched, we assert a producer/consumer commitment of the following form:

$$(p \rightarrow c, \min(\text{amount-unmatched}(p, S), \text{amount-unmatched}(c, S)) \quad (25)$$

Where:

- *p* is a producer of *I*,
- amount-unmatched(*a*, *S*) is the amount of inventory produced (or consumed) by *a* that has not been matched with a corresponding consumer (or producer) in state *S*,
- and amount-unmatched(*p*, *S*) > 0.

If, through a complete retraction technique, we derive that the producer/consumer commitment results in a dead-end, we post the alternative commitment ¬ (*p* → *c*, min(amount-unmatched(*p*, *S*), amount-unmatched(*c*, *S*)). This alternative commitment is a no-good and has the effect of removing from consideration any heuristic commitment between *p* and *c* until a state, *S'* is reached such that:

$$\begin{aligned} &\min(\text{amount-unmatched}(p, S'), \text{amount-unmatched}(c, S')) < \\ &\min(\text{amount-unmatched}(p, S), \text{amount-unmatched}(c, S) \end{aligned} \quad (26)$$

The possibility of a heuristic commitment between *p* and *c* in state *S'* is a necessary condition for the completeness of the inventory branching scheme. For example, in Figure 79, we discover, based on backtracking to state *S*, that the commitment ($P_1 \rightarrow C_2$, 50) does not lead to an overall solution. We cannot, however, discard *any* producer/consumer commitment between P_1 and C_2. As shown in state *S'*, the commitment ($P_1 \rightarrow C_2$, 25) is necessary in a solution. This commitment does not contradict the no-good discovered in state *S* and satisfies Expression (26).

This branching scheme is similar to the "schedule versus postpone" branching scheme for start time assignments used in [Le Pape et al., 1994]. It is not immediately clear, however, that the branching scheme we propose results in a complete search, assuming we use a complete retraction algorithm (*e.g.*, chronological backtracking).

6.3.1.1 The Completeness of the Producer/Consumer Branching Scheme

To demonstrate the completeness of the proposed branching scheme (when used with a complete retraction technique) it is necessary to show that a dead-end will never be improperly derived. Clearly, if the set of asserted commitments leads to a state that breaks one or more of the problem constraints, we have found a true dead-end. More interesting is a state, *S*, such that there exists a consumer, *c*, that is not completely matched, but all the possible producer/consumer commitments contradict a no-good (of the form described above). The proposed branching scheme detects a dead-end at *S*. It is not obvious, however, that some combination of inventory transfers that do not match our branching scheme will not lead to a solution. For example, Figure 80 shows a situation

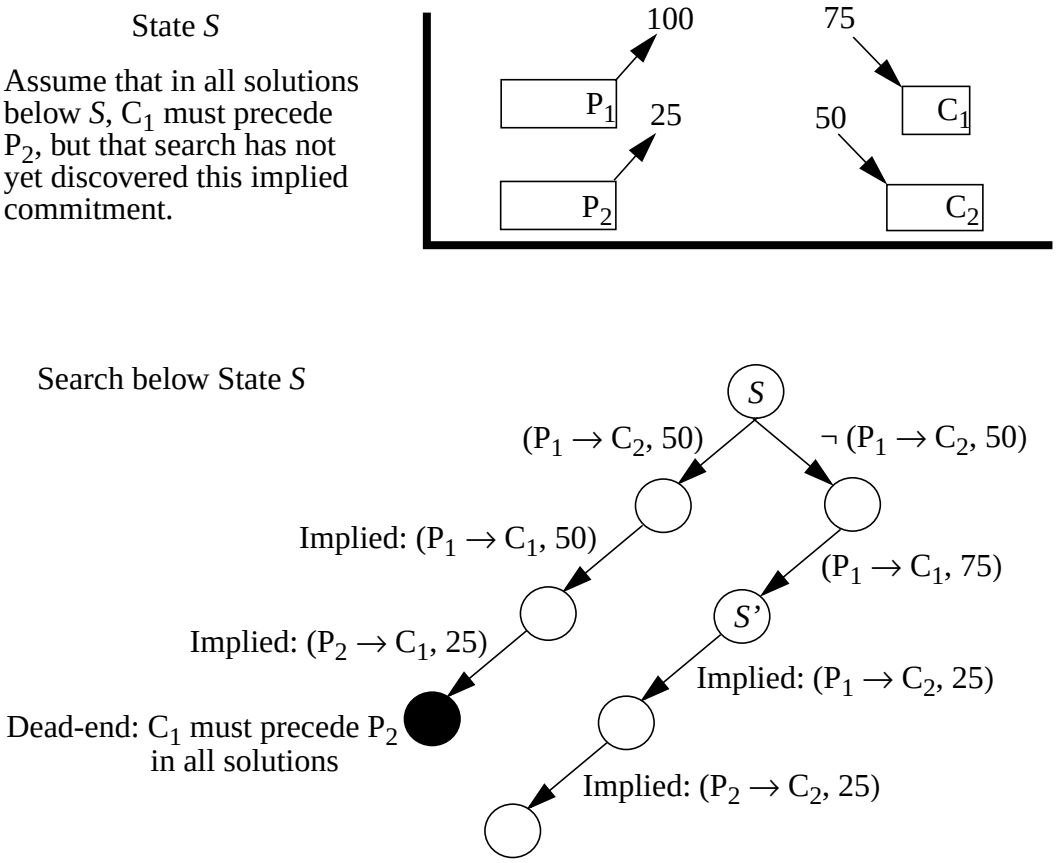


Figure 79. A Situation Where Consideration of a Commitment in Between P_1 and C_2 in State *S'* is Necessary for Completeness.

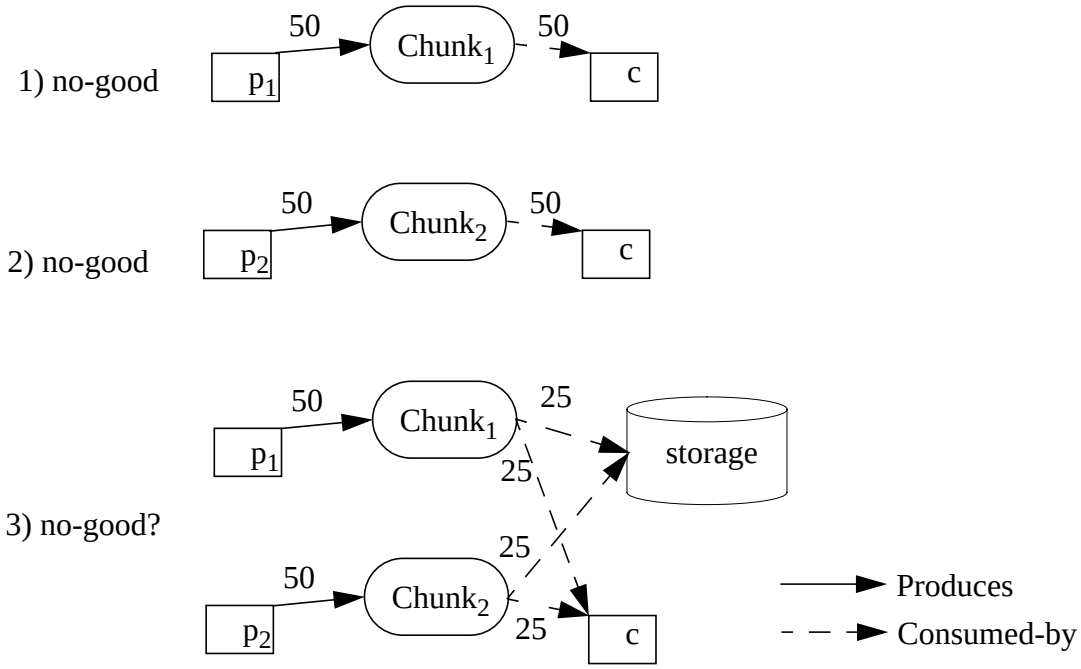


Figure 80. Given Two No-goods (1 and 2), Will the Third also Result in a Dead-end?

where c has 50 units of unmatched inventory and two producers, p_1 and p_2, each produce 50 units. The commitments that transfer 50 units of inventory to c from either of p_1 or p_2 have already been discovered to result in dead-ends. Perhaps, however, two commitments that transfer lesser amounts (and will not be found by our branching scheme, *e.g.*, $(p_1 \rightarrow c, 25)$, $(p_2 \rightarrow c, 25)$) will lead to a solution. In this section, we show that this can not be the case: we derive a valid dead-end.

Definitions

- Let *NOGOODS*(S) be the set of asserted producer/consumer no-goods at state S.
- Let $P(c, S)$ be the set of producers such that:

$$\forall p_i \in P(c, S)\ \neg(p_i \rightarrow c, \min(\text{amount-unmatched}(p_i, S), \text{amount-unmatched}(c, S)) \in \mathit{NOGOODS}(S) \quad (27)$$

Let us examine the cardinality of $P(c, S)$.

$|P(c, S)| = 0$. If $P(c, S)$ is empty, clearly we are at a dead-end: we have a consumer, c, that is not completely matched, yet we have no producers with which it can be matched regardless of the derived no-goods.

$|P(c, S)| = 1$. Let p be the only element of $P(c, S)$. Clearly, amount-unmatched(c, S) ≥ min(amount-unmatched(p, S), amount-unmatched(c, S)). Imagine that a producer/consumer commitment transferring some smaller amount of inventory than specified in our branching scheme is consistent. That commitment would necessarily transfer an amount, $A^* <$ min(amount-unmatched(p, S), amount-unmatched(c, S)). We would be left in a state where c still is not completely matched. We could again commit to transferring a smaller amount; however, in order to meet the requirements of c, we would eventually have to transfer a total of min(amount-unmatched(p, S), amount-unmatched(c, S)). Since multiple transfers of inventory between the same producer and consumer are identical (with respect to the problem constraints) to a single transfer, the multiple transfer must lead to the same dead-end that was found when the no-good was derived. Either we end in a state where c has some unmatched consumption or in a state after the assertion of a producer/consumer commitment that results in a dead-end. Both options are dead-ends.

$|P(c, S)| > 1$. Assume we have a state that is not a dead-end, such that the producers in $P(c, S)$ supply inventory to c at amounts that do not contradict the asserted no-goods. By the existence of the producer/consumer constraints each producer $p_i \in P(c, S)$ must produce its inventory at or before the start time of c. Therefore, at the start time of c, some of the accumulated inventory is consumed by c and some is left in storage. All the inventory required by c is in storage at or before the time c starts. The same type of inventory is produced by each producer; therefore, when c consumes the inventory, it is irrelevant which producer actually produced it. Therefore, of the inventory from each producer that is in storage at the start of c, we are free to define the amounts consumed by c. Without changing either start times or inventory levels (and so retaining our consistent state) we can specify that for at least one producer, $p^* \in P(c, S)$, the amount transferred from p^* to c is min(amount-unmatched(p^*, S), amount-unmatched(c, S)). However, we have

already discovered that this commitment leads to a dead-end. We have a contradiction. It is not the case that some set of transfers of less inventory than our branching scheme can lead to a solution if the commitments in our branching scheme are all no-goods.

Summary. We have shown that a complete branching rule for the producer/consumer commitments need not consider all possible inventory transfers between the producer and consumer to remain complete. In fact, at a search state, *S*, there is only a single commitment for each producer, *p*, and consumer, *c*, that need be considered: ($p \rightarrow c$, min(amount-unmatched(*p*, *S*), amount-unmatched(*c*, *S*)).

6.3.1.2 The Temporal Component of a Producer/Consumer Commitment

As noted above, the standard temporal constraint that forms part of the producer/consumer commitment is a precedence constraint that enforces the relationship that the consumer must start at or after the end of the producer. In some cases, however, we can derive that the consumer must start exactly at the end time of the producer. Consider the case where a producer, *p*, produces 100 units of inventory, a consumer, *c*, consumes 75 units, and the maximum inventory constraint is 50 units. Regardless of inventory level due to other producers and consumers, it must be the case, if *p* and *c* participate in the same producer/consumer commitment, that *c* consumes exactly when *p* produces. Otherwise, the inventory maximum constraint will be broken. Under such circumstances, the temporal component of a producer/consumer commitment is a meets constraint which enforces that the start time of *c* is equal to the end time of *p*.

6.3.2 Producer/Consumer Interval Commitments

It is not the case that a state in which each consumer is completely matched is necessarily a solution or a dead-end. It may be that commitments to further constrain the start time domains of activities are necessary to find a solution. Rather than immediately assigning start times to activities, we again follow a least-commitment approach by constraining the time interval that is allowed between a producer and a consumer that already participate in a producer/consumer relationship.

When all consumers are completely matched, however, it can not be the case that an inventory minimum constraint is violated. Since all consumers have been paired with producers that supply sufficient inventory and are constrained to execute before the consumer, in any assignment of start times that obeys the existing temporal constraints, the inventory minimum constraint must be satisfied. It is possible, however, for an inventory maximum constraint to be broken in such a situation. In particular, imagine a critical inventory maximum constraint on an inventory that neither is at a dead-end nor meets the inventory termination criteria, but has no consumers that have not been completely matched. The critical inventory maximum constraint indicates that there is a relatively high probability that the inventory maximum constraint will be broken. To reduce this criticality, we can constrain a producer-consumer pair to execute closer together: we can limit the allowed time interval between the end of the producer and the start of the consumer. The intuition behind this commitment is that when a producer and consumer execute closer together in time, the average inventory level over time is decreased. This decrease is likely to reduce the criticality of the inventory maximum constraint.

The actual commitment made is to constrain the interval between producer and consumer to be equal to or less than the middle value of the current domain of possible interval lengths. If such a commitment is retracted, we can post the opposite commitment: constraining the temporal inter-

val to be greater than the middle value in the domain. For example, if the maximum interval between the end of a producer and the start of a consumer is 100 time units, the heuristic commitment is to constrain the interval to be less than or equal to 50 time units. On backtracking, this is reversed to constrain the interval to be greater than 50 time units. Since the interval between producer and consumer must be either less than or equal to its middle domain value, or greater than it, clearly this branching scheme results in a complete search (when used with a complete retraction technique).

6.3.3 Start Time Commitments

Finally, it may be the case that even if all consumers are completely matched and all producer-consumer pairs are constrained to occur within a constant time of each other, the search state is still neither a dead-end nor a solution. In such a state, it is necessary to assign start times to at least some of the activities.

6.4 Texture Measurements for Inventory

As with the existing work on texture measurements, our goal in each search state is to identify the constraint that is most in danger of being broken and focus our heuristic commitment technique on reducing this danger. Incorporation of inventory constraints into this scheduling scheme requires:

1. The ability to estimate the criticality of inventory minimum and maximum constraints.
2. The ability to compare the criticality of inventory constraints with resource capacity constraints.
3. The ability to make heuristic commitments that will tend to reduce the criticality of the most critical constraint, regardless of the type of constraint.

In the previous section we discussed the commitments that we can make in order to reduce criticality of inventory minimum and maximum constraints. Previous chapters have examined commitments to reduce the criticality of resource capacity constraints. Here, therefore, we address the first two components of the incorporation of inventory constraints through the adaptation of the VarHeight texture measurement estimation technique for inventory constraints.

6.4.1 Adapting VarHeight to Inventory

Recall (Section 5.2.4) that the VarHeight texture measurement represents both the expected value and the variance of an activity's individual demand for a resource. The individual demand is then aggregated (based on assumptions discussed in Chapter 5) and the overall probability of breakage of the resource constraint is estimated by examining the area under the normal distribution determined by the aggregate expected value and aggregate variance. The proportion of the area of the distribution that lies to the right of the maximum constraint is used as an estimate of the probability that the constraint will be broken at that time point.

In extending the VarHeight texture estimation technique from resource to inventory constraints, we modify the individual demand curve that an activity has for an inventory. The procedure for the aggregation of individual inventory demands is identical to the aggregation of individual resource demands (see Section 5.2.4).

6.4.1.1 Individual Demand

An activity that produces or consumes inventory makes a positive or negative contribution to the level of that inventory. Assuming that each of the remaining start times is equally likely to be assigned, we can generate an activity's individual demand. *ID*(*A*, *I*, *t*) is (probabilistically) the amount of inventory *I*, produced (consumed) by activity *A*, at time *t*. (Recall that as defined in Section 2.1.3.2 for activity *A*, est_A is the earliest start time, lst_A is the latest start time, eft_A is the finish time, lft_A is the latest finish time, and STD_A is the domain of *A*'s start time variable.)

If *A* is a production activity, *ID* is calculated as follows, for all $eft_A \le t \le lft_A$:

$$ID(A, I, t) = AMT_A(I) \times \frac{t - eft_A + 1}{|STD_A|} \tag{28}$$

If *A* is a consumption activity, *ID* is calculated the same way, relative to the start time window. That is, for all $est_A \le t \le lst_A$:

$$ID(A, I, t) = AMT_A(I) \times \frac{t - est_A + 1}{|STD_A|} \tag{29}$$

Where $AMT_A(I)$ is the total amount of inventory *I* that *A* produces or consumes. If *A* is a consumer, $AMT_A(I) < 0$.

6.4.1.2 Variance of the Individual Demand

In considering inventory *I*, a time point *t*, and an activity *A*, we can associate a random variable *X* with the contribution (positive or negative) that *A* has to *I* at time *t*. The domain of *X* is {0, $AMT_A(I)$}. The expected value for *X*, *EX*, assuming a uniform distribution for the start time of *A*, is *ID*(*A*, *I*, *t*) as calculated in Equation (28) or Equation (29). We calculate the variance of *X*, *VX*, as follows:

$$VX = EX \times (AMT_A(I) - EX) \tag{30}$$

Derived as follows:

1. $VX = EX^2 - (EX)^2$ and $EX^2 = \sum x^2 p(x)$ by definition.
2. *x* can take on only the values 0 and $AMT_A(I)$ therefore:

$$EX^2 = (0 \times p(0)) + (AMT_A(I)^2 \times p(AMT_A(I))) = AMT_A(I)^2 \times p(AMT_A(I)) \tag{31}$$

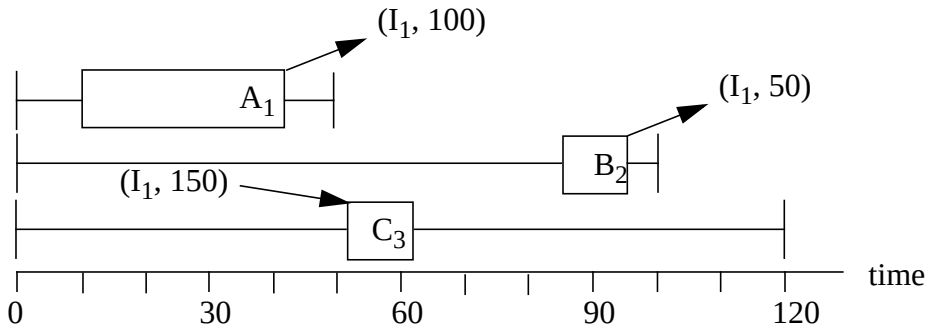


Figure 81. Activities A_1, B_2, and C_3 Producing or Consuming Inventory I_1.

3. So:

$$\begin{aligned} VX &= EX^2 - (EX)^2 \\ &= AMT_A(I)^2 \times p(AMT_A(I)) - AMT_A(I)^2 \times p(AMT_A(I))^2 \\ &= AMT_A(I) \times p(AMT_A(I)) \times (AMT_A(I) - AMT_A(I) \times p(AMT_A(I))) \\ &= EX \times (AMT_A(I) - EX) \end{aligned} \tag{32}$$

Given this definition, we can calculate EX_i and VX_i for all activities, A_i, at time point t.

6.4.1.3 Event-based Representation of Individual Demand

Recall from Expression (16) (Section 4.3.1) and Expression (24) (Section 5.2.4) that the individual curves representing VarHeight on a resource used six (*t, ID*) event points. Given the simpler shape of the inventory curves, we reduce the number of event points to five pairs $\{t_0, \ldots, t_4\}$ to represent the *ID* curves on inventory.

If activity *A* is a producer, the five event pairs used are those defined in Expression (33). For consumers, where $AMT_A(I) < 0$, the event pairs in Expression (34) are used.

$$i \in [0, 4], t_i = \left(eft + \frac{i}{4} \times (lft - eft), \frac{AMT_A(I)}{|STD|} + \frac{i}{4} \times AMT_A(I) \times \frac{|STD| - 1}{|STD|} \right) \tag{33}$$

$$i \in [0, 4], t_i = \left(est + \frac{i}{4} \times (lst - est), \frac{AMT_A(I)}{|STD|} + \frac{i}{4} \times AMT_A(I) \times \frac{|STD| - 1}{|STD|} \right) \tag{34}$$

The individual demand curves for the activities in Figure 81 are displayed in Figure 82. I_1 is the aggregate demand curve: the expected value of the demand. It is used, with the variance curve, to find the probability of breakage of the minimum and maximum constraints at each time point.

6.4.2 Aggregating Demand

The aggregation of the individual demands on inventory is done exactly as the aggregation on resources described in Section 5.2.4. In particular, note that the aggregation depends on two assumptions:

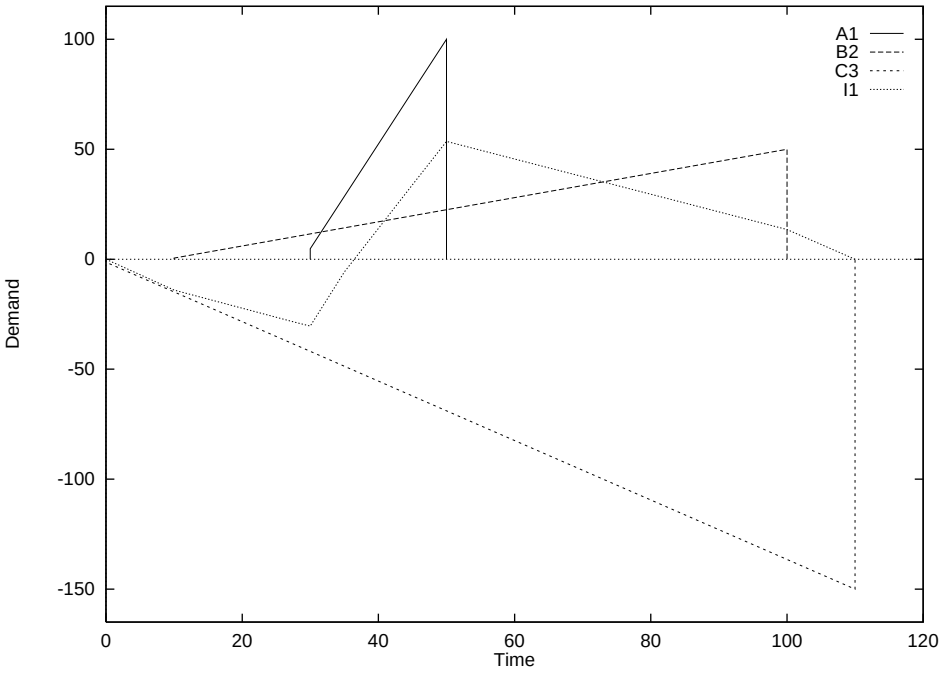


Figure 82. Inventory Curves for the Activities Shown in Figure 81.

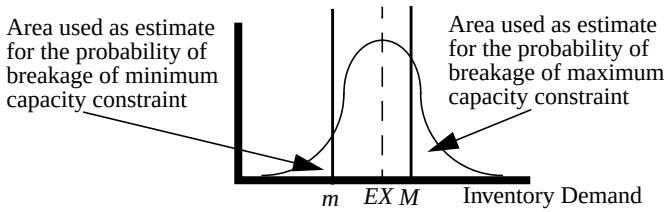


Figure 83. Calculating the Probability of Breakage at Event *t* with VarHeight.

1. The random variables representing the individual demand associated with each activity are mutually independent.
2. The aggregate random variable is normally distributed around the expected value.

Both of these assumptions are suspect; however, as with the VarHeight texture measurement on resources, we will make them in order to estimate the probability of breakage of inventory constraints as illustrated in Figure 83. The area under the curve greater than the maximum capacity constraint is used as an estimate of the probability of breakage of the maximum constraint, and the area under the curve less than the minimum capacity constraint is used as an estimate of its probability of breakage.

Complexity. We can use the same technique as with resource texture measurements of storing the incoming and outgoing slopes at each point in the individual curves and sorting the individual time points into a single list. In a single pass the probability of breakage curves can be generated. This process has complexity of $O(mn \log n) + O(mn)$. The space complexity is $O(mn)$, as we maintain an individual demand curve for each activity.

6.5 Propagators for Inventory

Two propagators for inventory constraints are used in this dissertation. In the first, inventory bound propagation, the upper and lower bounds on the inventory levels (Section 6.2.1) are used to detect dead-ends and to derive new unary temporal constraints on the producing and consuming activities. The second propagator, producer/consumer propagation, uses the temporal characteristics of the producers and consumers to detect dead-ends and to infer new producer/consumer constraints.

6.5.1 Inventory Bound Propagation

Inventory bound propagation is based on the comparison of the upper and lower bound on inventory levels with the minimum and maximum inventory constraints. Based on such a comparison, it is possible, in some cases, to detect a dead-end due to the inventory levels or to infer new unary temporal constraints on production and consumption activities.

6.5.1.1 Dead-end Detection

Given the calculation of the lower and upper bounds on inventory levels, a straightforward dead-end detection technique is to compare the bounds with the inventory constraints. If, at any point on the scheduling horizon the upper bound is less than the minimum inventory constraint, we can clearly derive a dead-end: no combination of commitments will result in a higher inventory level and that level is less than the minimum inventory constraint. Similarly, if the lower bound on the inventory level is ever greater than the maximum constraint, we can infer a dead-end.

Complexity. Once the inventory bounds have been calculated as described above, inventory bound dead-end detection can be done in time O(m), where m is the number of inventories in the problem. The inventory bound calculation algorithm can be augmented so that the maximum point on the upper bound curve and the minimum point on the lower bound curve are cached each time the bound is recalculated. Assuming that the inventory constraints are constant across the horizon, it is only necessary to examine the minimum point on the lower bound and the maximum point on the upper bound curve to detect such a dead-end.

6.5.1.2 Inventory Bound Propagation of the Minimum Inventory Constraint

When propagating the minimum inventory constraint, we make use of the upper bound on the inventory level. The basic calculation of the upper bound (as described in Section 6.2.1) is to have all producers executing at their earliest start time and all consumers executing at their latest start time. (Recall that the presence of a meets constraint between a producer and a consumer is treated as a special case.)

Imagine a time point, t, and a producer, p, such that $eft_p \leq t < lft_p$. Assume that the upper bound at t is $UB(t)$. Inventory bound propagation on the minimum constraint compares that value $UB(t) - AMT_p(I)$ with the minimum constraint at time t. If the value is less than the minimum constraint, we can infer that p must produce its inventory at or before time t and therefore we have derived a new temporal constraint on p. (If p happens to be in a meets relationship with a set of consumers then the value subtracted from the upper bound is the amount produced by p less the amount consumed by all the consumers that it meets.)

Similarly, imagine a consumer, c, that does not take part in any meets constraints with producers of inventory I, such that, $est_c \leq t < lst_c$. If the value $UB(t) + AMT_c(I)$ (recall that $AMT_c(I) < 0$) is less than the minimum constraint, then c must consume its inventory after t.

Complexity. At worst, at each event point we must examine each activity on the inventory. Given that there can be $O(n^2)$ event points (Section 6.2.1), this leads to an overall time-complexity of $O(n^3)$. In practice, however, this is a large over-estimate since it is unlikely that each consumer consumes from each producer. In addition, there are a number of average time optimizations in place that significantly reduce the run-time over a naive implementation (*e.g.*, the activities are sorted in descending order of $|AMT_A(I)|$ so that when comparisons of activity A fail to break the minimum constraint at time point t, we know that none of the other activities will break it either and so can move to the next time point).

6.5.1.3 Inventory Bound Propagation of the Maximum Inventory Constraint

Inventory bound propagation for the maximum inventory constraint is analogous to that for the minimum inventory constraint. Using the lower bound on the inventory level at a time point, $LB(t)$, and a producing activity, p, such that $eft_p \leq t < lft_p$, the bound propagation compares the value $LB(t) + AMT_p(I)$ to the maximum constraint. If the value is greater than the maximum constraint, we derive that p must produce after time t. When a producer is linked to consumers via the InventoryChunk representation, a slight variation to the above is required. The amount added to the lower bound is not simply $AMT_p(I)$, but rather $AMT_p(I)$ less all the consumers of that InventoryChunk that can consume at t.

For a consumer, c, attached to an InventoryChunk produced by activity p, we examine the time points, t, such that, $\max(lft_p, est_c) \leq t < lst_c$. If $LB(t)$ less the amount consumed from the InventoryChunk by c is greater than the maximum constraint, c must consume at or before t. Finally, for a loose consumer, we examine the time points, t, such that $est_c \leq t < lst_c$, and compare the lower bound less the amount of unmatched consumption to the maximum inventory.

Complexity. The time-complexity of the maximum constraint propagation is the same as for the minimum constraint propagation: $O(n^3)$ when each activity needs to be examined at each time-point. In practice, again, the average time complexity is much lower.

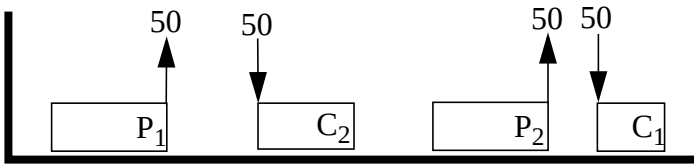
6.5.2 Producer/Consumer Propagation

Producer/consumer propagation simply examines the available consumers for each producer and the available producers for each consumer.

6.5.2.1 Dead-end Detection

If there are no producers that can (temporally) supply a consumer or if the total amount of inventory from the possible producers is less than that consumed by the consumer, we can derive a dead-end.

Note that producer/consumer dead-end detection is not subsumed by inventory bound dead-end detection. The situation displayed in Figure 84 is a case where inventory bound dead-end detection does not identify a dead-end that producer/consumer dead-end detection does. Due to the matching production and consumption amounts, it appears that the inventory minimum will never be broken. However, because the inventory produced by P_1 has been constrained to be consumed



Existing commitment: ($P_1 \rightarrow C_1$, 50)

Figure 84. An Example of a Dead-end that Inventory Bound Dead-end Detection Does Not Find but Producer/Consumer Dead-end Detection Does.

by C_1, C_2 will have no inventory to consume. To resolve this situation, the commitment ($P_1 \rightarrow C_1$, 50) must be retracted and replaced with ($P_1 \rightarrow C_2$, 50) and ($P_2 \rightarrow C_1$, 50).

6.5.2.2 Propagation

Producer/consumer propagation is a simple extension of the dead-end detection. If, rather, than finding no possible producers for a consumer, one is found, then a producer/consumer commitment is inferred. Similarly, if there is only one possible consumer and a producer can not place all its inventory in storage, a producer/consumer commitment is inferred.

6.5.2.3 Complexity

The time-complexity of the producer/consumer dead-end detection and propagation, together, is $O(mn \log n)$. On each of the m inventories, the n activities contributing to each inventory are sorted on a list in ascending order. Producers are sorted based on their minimum start time while consumers are sorted based on their maximum finish time. A single iteration through this list then keeps track of both the number of producers with some unmatched production and the total amount of the unmatched inventory. If the unmatched inventory drops below zero or if a consumer has no unmatched producers before it, we have a dead-end. If a consumer has only one unmatched producer preceding it, we infer a new producer/consumer commitment.

6.6 Inventory Scheduling Strategies

Based on the texture measurements and propagators presented above, we can now specify the scheduling strategies used to address inventory scheduling problems.

6.6.1 Propagators

To evaluate the inventory propagation techniques, we use two sets of propagators, one including the inventory propagators, and one including only the inventory dead-end detection. The scheduling problems (described in Section 6.7) contain unary capacity resources; therefore, the propagators used in the previous chapters are also used.

Specifically, the two sets of propagators consist of the following, in order:

1. temporal propagation, edge-finding exclusion, edge-finding not-first/not-last, CBA, inventory dead-end detection, inventory bound propagation, and producer/consumer propagation
2. temporal propagation, edge-finding exclusion, edge-finding not-first/not-last, CBA, and inventory dead-end detection

We refer to the former set of propagators as the *Inventory* propagators and the latter set as the *Non-Inventory* propagators.

6.6.2 A Texture-based Heuristic Commitment Technique

The aggregate texture curves of inventories allow us to estimate the probability of breakage of inventory minimum and maximum constraints over time. The model of a texture-based heuristic commitment technique used to this point has not been changed by the addition of the new texture measurements. The heuristic does the following:

1. Calculates texture measurements on all resource and inventory constraints.
2. Identifies the most critical constraint and time point, defined to be the constraint and time point with the highest probability of breakage. Ties are broken arbitrarily.
3. Generates a commitment to reduce the criticality of the most critical constraint.

The heuristic commitment that is generated depends on which type of constraint is judged as most critical. Therefore, the different types of commitments are interleaved throughout the scheduling process based on the texture information.

The texture measurement used in this chapter is the VarHeight texture as it was the best of the easily extendible texture measurement-based heuristics in the previous chapter.

6.6.2.1 Resource Commitment

The commitment made when a resource constraint is the most critical constraint is the same as that used in Chapter 4 and Chapter 5: the two activities which contribute the most individual demand to the critical resource and time point, and that are not already connected by a path of temporal constraints are sequenced.

The heuristics to determine which sequence is selected are the same as those used in previous chapters: MinimizeMax, Centroid, and Random (see Section 4.3.2.2).

6.6.2.2 Minimum Inventory Commitment

When a minimum inventory constraint is found to be most critical, we are in danger of going below minimum inventory level. The heuristic, therefore, identifies all consumers that can consume at or before the critical time point, and selects the one with the largest unmatched inventory. The activity is, heuristically, the most critical consumer.

Having the most critical consumer, *c*, we then examine all producers that can supply that consumer. These producers must have an earliest finish time less than or equal to the latest start time of the critical consumer, and also must have some unmatched inventory. The producer, *p*, chosen to participate in a heuristic producer/consumer commitment with *c*, is the one that minimizes the value $lst_c - eft_p$. The justification for this heuristic is that it will tend to minimize inventory levels if the producer and consumer are able to execute close together in time. The commitment posted is a producer/consumer commitment between *p* and *c* as described in our branching scheme (Section 6.3.1).

A number of more complex techniques for selecting the producer/consumer pair were investigated such as minimizing the temporal pruning resulting from the producer/consumer commitment, minimizing the difference in amounts being produced and consumer, and various combinations. The simple heuristic used in our experiments proved to be as good as (and often better than) these more sophisticated heuristics.

6.6.2.3 Maximum Inventory Commitment

When a maximum inventory constraint is critical, we have the choice of three types of commitments to be made: producer/consumer, producer/consumer interval, and start time assignment (see Section 6.3). Following the least commitment strategy, the heuristic we use will make commitments in the following order:

1. A producer/consumer commitment unless all consumers on the critical inventory are completely matched.
2. A producer/consumer interval commitment unless all producer-consumer pairs on the critical inventory already have a singleton domain for their interval.
3. A start time assignment.

Each of these commitment choices have their own sub-heuristics.

Producer/Consumer Heuristic

When a maximum inventory constraint is critical, the heuristic identifies the producer, *p*, with the largest unmatched inventory that can produce at or before the critical time point. The consumers whose latest start times are greater than the earliest finish time of *p* are evaluated and, as with the inventory minimum heuristic, the consumer, *c*, that minimizes $lst_c - eft_p$ is chosen.

Producer/Consumer Interval Heuristic

If all consumers are completely matched, there are no possible producer/consumer commitments to be made. However, since the inventory maximum constraint is critical, we still need to post a commitment that will tend to reduce the inventory level at the critical time point, *t**. The commitment that is posted is a producer/consumer interval commitment which constrains the time-interval between the end of the producer and the start of the consumer to be equal to or less than the current middle value in the domain representing the interval. We post this commitment because we are attempting to lower the inventory level, and by shortening the amount of time between the production and consumption of the inventory, we will tend to lower the inventory level.

The producer-consumer pair, *p* and *c*, is heuristically selected such that the inventory transferred from *p* to *c* is the maximum of all producer-consumer pairs that meet the requirement that $eft_p \leq t^* \leq lst_c$. This selection will choose a pair of activities that are likely to transfer the most inventory (via storage) at the critical time point.

Start Time Assignment Heuristic

Finally, if all the producer-consumer intervals have a singleton value, no further producer/consumer interval commitments are possible. However, it still may be the case that the inventory maximum constraint is not guaranteed to be satisfied. In this situation, we assign start times to activities.

Our start time assignment follows the “schedule earliest versus postpone” branching scheme for start time assignments used in [Le Pape et al., 1994]. All activities that are not already assigned a start time are ordered in ascending order of earliest start time with ties broken by ascending order of latest start time. A commitment results from selecting the first element on the list and assigning its earliest start time. If a dead-end is found and a start time assignment is undone, a no-good is posted specifying the activity and the start time that failed. The activity then can not take part in a start time commitment until propagation results in a new earliest start time for the activity.

We refer to this start time commitment as the *EorP* heuristic for “earliest-or-postpone.”

6.6.3 Non-texture-based Inventory Heuristic Commitment Techniques

Without a technique to evaluate and compare the criticality of inventory and resource constraints, there does not appear to be a principled way to interleave the types of heuristic commitments made during scheduling. How does a heuristic judge that it should post a resource sequencing commitment rather than a producer/consumer commitment? If we are to make commitments on both inventories and resources, it appears that we are required to specify a static ordering such as making all heuristic commitments on inventory first, then switching heuristics to make resource commitments. This is the technique used in the set of non-texture based inventory heuristic commitment techniques.

The heuristic commitment technique is as follows:

1. If there are any unmatched consumers, post a producer/consumer commitment.
2. Otherwise, if there are unsequenced activities on a resource, make a commitment on activities on a resource.
3. Finally, if all possible commitments on resource activities have been made, assign start times.

Different techniques can be used in each of these three steps.

6.6.3.1 Producer/Consumer Commitment Heuristic

The non-texture-based producer/consumer heuristic arbitrarily selects consumers in descending order of their amount of unmatched consumption. An upstream producer is identified (following the minimization of the $lst_c - eft_p$ value as in the texture-based heuristic) and a producer/consumer commitment is posted.

We refer to this heuristic as *GreedyInv.*

6.6.3.2 Resource Commitment Heuristics

When making a commitment on a resource, after all consumers have been completely matched, any of the techniques used for job shop scheduling can be used. In particular, we use SumHeight, CBASlack, and EorP. For SumHeight, texture measurements are maintained on the resources but not on the inventories. After the producer/consumer heuristic has matched all consumers, SumHeight is then used to estimate the criticality of all resources and to sequence the top two activities on each resource, exactly as done in the job shop problems in Chapter 5.

6.6.3.3 Start Time Assignment Heuristics

Both SumHeight and CBASlack sequence activities on the resources. It is possible, however, to reach a search state such that all consumers are completely matched and all activities on the resources are sequenced, yet the termination criteria on one or more inventories are not met. In such a search state, start time assignments are necessary.[5]

The start time assignment heuristic used after all producer/consumer commitments, producer/consumer interval commitments, and resource sequencing commitments have been made is the EorP heuristic described above (Section 6.6.2.3).

---

5. When the EorP heuristic is used as the resource commitment heuristic (Section 6.6.3.2), no separate start time assignment heuristic is needed.

6.6.4 Scheduling Without Inventory Heuristics

As noted in Chapter 2, the existing work on inventory scheduling does not directly make heuristic commitments on the inventory activities. Rather, it uses more standard scheduling heuristics, and allows propagation to constrain the inventory levels and detect dead-ends. To provide a basis of comparison for our inventory heuristics against heuristics typical of those in the literature, we use the resource assignment heuristics and start time assignment heuristics discussed in the previous section. We run one of our resource heuristics (SumHeight, CBASlack, and/or EorP) and allow the inventory propagation to maintain the inventory constraints.

6.6.5 A Note on Early Termination

All of the heuristic commitment techniques discussed above may require the assignment of start times when applied to inventory scheduling problems. Recall, however, the inventory termination criteria presented in Section 6.2.2. If a state is reached such that all inventory bounds are within the minimum and maximum constraints, and all resources are completely sequenced, the algorithm terminates with success: a solution has been found.

6.6.6 Instantiations of the ODO Framework

We have presented a number of novel heuristic commitment techniques and propagators in this chapter. In particular, as described in Section 6.6.1, we have two sets of propagators: Inventory and Non-Inventory. We have also introduced three general classes of heuristic commitment techniques that can be applied to inventory scheduling.

The first class consists of the texture-based inventory heuristics whose defining characteristic is the calculation of texture measurements on both inventory and resource constraints in order to compare constraint criticality and find a commitment that will tend to decrease that criticality. In the case where all producers and consumers are matched and the activities on each resource are sequenced, it is necessary to assign start times. The sole instance of this heuristic commitment class used in this chapter is the VarHeight heuristic described in Section 6.6.2 which uses the EorP heuristic, if necessary, to assign start times.

The second class of heuristic commitment techniques is one in which the producer/consumer commitments are made before transitioning to a second heuristic commitment component to sequence the activities on each resource, followed (possibly) by a third heuristic to assign start times. The only non-texture heuristic to make producer/consumer commitments is the GreedyInv heuristic described in Section 6.6.3. It is followed, in our experiments, by one of SumHeight, CBASlack, or EorP. Since it may be necessary to assign start times, SumHeight and CBASlack are each followed by EorP.

The final class of heuristic commitment technique is composed of heuristics which simply make commitments on the resources and leave the inventory propagation to maintain the inventory constraints. In this chapter, the instances of this class are identical to the instances of the above class except that they do not use the GreedyInv heuristic to make inventory commitments.

The scheduling strategies instantiated in the ODO framework and used in the experiments in this chapter are summarized in Table 6. Note that this is not a fully crossed experimental design. Because our central interest is the evaluation of heuristics, it was decided that experimentation with all possible combinations of the heuristics and propagators presented in this chapter was not

Class	Algorithm	Heuristic Commitment Technique	Propagators	Retraction Technique
Texture-based Heuristics	VarHeight	VarHeight, EorP	Non-inventory[a]	Chronological Backtracking
	VarHeightProp	VarHeight, EorP	Inventory[b]	Chronological Backtracking
Non-texture-based Heuristics	GreedySum-Height	GreedyInv, Sum-Height, EorP	Non-inventory	Chronological Backtracking
	GreedySum-HeightProp	GreedyInv, Sum-Height, EorP	Inventory	Chronological Backtracking
	Greedy-CBASlackProp	GreedyInv, CBASlack, EorP	Inventory	Chronological Backtracking
	GreedyEorPProp	GreedyInv, EorP	Inventory	Chronological Backtracking
Non-inventory Heuristics	SumHeightProp	SumHeight, EorP	Inventory	Chronological Backtracking
	CBASlackProp	CBASlack, EorP	Inventory	Chronological Backtracking
	CBASlack	CBASlack, EorP	Non-inventory	Chronological Backtracking
	EorPProp	EorP	Inventory	Chronological Backtracking

a. Temporal propagation, edge-finding exclusion, edge-finding not-first/not-last, CBA, and inventory dead-end detection.

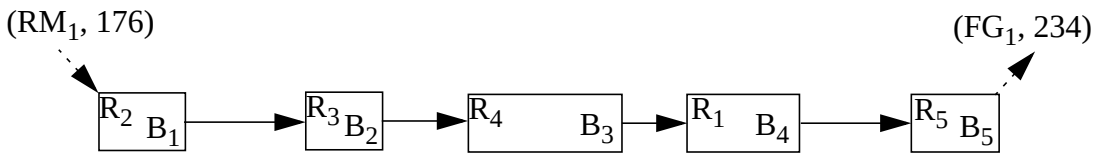
b. Temporal propagation, edge-finding exclusion, edge-finding not-first/not-last, CBA, inventory dead-end detection, inventory bound propagation, and producer/consumer propagation.

Table 6. **The Algorithms Used in the Inventory Experiments.**

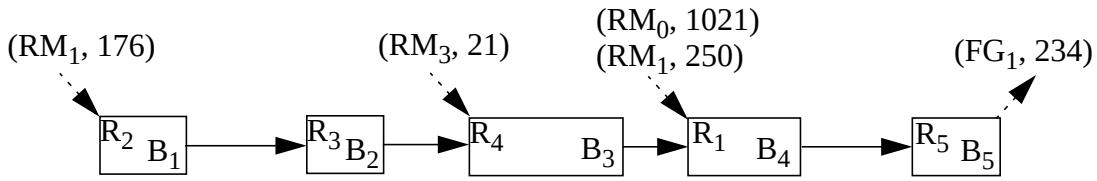
warranted. We have selected a subset of the strategies so that we can briefly evaluate the usefulness of the inventory propagation techniques and, more importantly, investigate the inventory heuristics.

6.7 Problem Generation

Given the relative lack of inventory problems in the literature, our approach to the generation of problem instances is to start as simply as possible by adding inventory requirements to the job shop scheduling model. While these one-stage problems introduce new constraints among process plans based on requirements to consume inventory, there are no requirements for chains of production and consumption, for example, where one process plan produces an inventory that is required by a subsequent process plan. To add such production/consumption chains, we combine pairs of one-stage problems into two-stage problems.



Process Plan with Initial Consumption and Production



Process Plan with Subsequent Consumption Added (c = 4)

Figure 85. A Process Plan from a One-Stage 5×5 Inventory Problem.

6.7.1 One-Stage Inventory Problems

As noted above, the goal with the one-stage inventory problems is to add the necessity for inventory reasoning to the job shop scheduling problem model. Given a square, $n \times n$ job shop problem, we calculate the lower bound on the makespan as described by [Taillard, 1993]. The scheduling horizon, then, is based on a *makespan factor* multiplied by the lower bound calculation. As in other chapters of this dissertation, the makespan factor is varied in our experiments.

We introduce inventory by associating a single raw material inventory and a single finished goods inventory with each process plan. This gives the problem a total of $2n$ inventories: n raw materials and n finished goods. The minimal inventory addition to the job shop problem is to specify that the first activity in each process plan consumes the corresponding raw material and the final activity produces the corresponding finished good. We can then specify supply events for each raw material and demand events for each finished good.

This minimal addition does not, however, significantly change the job shop problem since there is no search required to make inventory commitments: there is only a single consumer for each raw material and only a single producer for each finished good. To further modify the underlying problem, we specify that a number, c, of the existing activities in each process plan consume one of the raw material inventories. Figure 85 displays a single process plan from a 5×5 problem with c equal to 4, meaning that, on average, all but one activity in the process plan consumes a raw material inventory. The activities that consume and the raw materials that are consumed are randomly selected with uniform probability and so, as shown in Figure 85, it is possible for one activity to consume multiple inventories (*e.g.*, B_4 consumes RM_0 and RM_1) and also for more than one activity in a process plan to consume the same inventory(*e.g.*, B_1 and B_4 both consume RM_1). The only restriction on a specific activity is that each consumption must be of a different inventory. As a result of this generation of consumption, the maximum value for c is n^2, when each activity in a process plan consumes each raw material inventory in the problem. The value of c is one of the key independent variables in our experiments.

6.7.1.1 Supply and Demand Events

After all the inventory consumption has been generated, it is necessary to generate both the amount and time of the supply and demand events. The raw materials must be introduced into the problem formulation via supply events, while the finished goods must have corresponding demand events.

The amount of a raw material RM_i supplied is found by summing the amount of all consumers of RM_i. The amount of RM_i produced by supply events is then calculated by multiplying this sum by a *supply factor*. For all the problems examined in this chapter, we set the supply factor to be 1.2 meaning that 20% more inventory than necessary is supplied to the problem. Future work will examine the manipulation of the supply factor.

The time of the supply events for RM_i is determined by calculating, $maxST(RM_i)$, the latest time that a consumer of RM_i can begin execution. This time is found by calculating the minimum tail [Carlier and Pinson, 1994] of all consumers of RM_i and subtracting that value (weighted by the makespan factor) from the length of the scheduling horizon as shown in Equation (35).

$$maxST(RM_i) = horizon - mkspFactor \times minTail(RM_i) \tag{35}$$

The supply of raw material is created via five supply events (each contributing an approximately equal amount of inventory) whose times are evenly distributed on the temporal interval $[0, maxST(RM_i)]$.

There is a single demand event for each finished good. It occurs at the end of the horizon and demands the total amount of a finished good produced in the problem.

Recall that the motivation for the one-stage problems is to control the inventory complexity. By having multiple supply events and multiple consumers for each raw material, we introduce the need to schedule consuming activities so as to meet supply events. This introduces new dependencies (via competition for raw materials) among activities in the problem. In the same spirit, our goal is to limit the inventory complexity by not requiring any search for the finished goods inventories: since there is a single producer and a single demand event for each finished good, they do not add extra complexity to the one-stage problem.

6.7.1.2 Inventory Constraints

The final aspect of a one-stage problem generation is the inventory constraint levels. For all inventories, the minimum constraint is an inventory level of 0.

For a finished good, FG_i, the maximum constraint is equal to the total amount of the FG_i produced in the problem. Given a single producer and a single demand event for each finished good, this form of a maximum constraint will not constrain the problem. At any time in the horizon, a producing activity can execute and create the finished good, and it can then remain in storage until the demand event at the end of the horizon.

For a raw material, RM_i, the maximum constraint is equal to 0.75 times the total amount of RM_i produced by supply events. This maximum does constrain the problem: at most three supply events can occur before a consumption must take place. (Recall that the five supply events are temporally distributed and each supplies approximately 20% of the inventory for a raw material).

As above, our motivation is to limit the inventory complexity in a one-stage problem to the raw materials.

6.7.2 Two-Stage Inventory Problems

Two-stage inventory problems are generated by combining two one-stage problems. There are three components to the combination process: inventory, resource, and temporal combination.

6.7.2.1 Inventory Combination

Given the two problems, P_1 and P_2, the production originally done by the supply events in P_2 is replaced with production done by existing activities in P_1. For example, for RM_{12} in P_2, rather than having five supply events producing RM_{12}, there are now five activities from P_1 that produce RM_{12}, in the quantities originally produced by the supply events. The activities that produce each raw material are chosen randomly with uniform probability from the activities in P_1. The only restriction is that intermediate inventory is produced by five different activities; however, a single activity may produce more than one inventory.

The finished goods production from P_1 and P_2 remain the same as does the raw material supply and consumption from P_1. In particular, the time of the supply events for the raw materials from P_1 remains unchanged.

More specifically, let us examine a process plan, P_1PP_i, originally in problem P_1 and another P_2PP_j originally from P_2. The activities in P_1PP_i consume raw materials from P_1 as before and the final activity in P_1PP_i still produces the finished good P_1FG_i. The demand event for P_1FG_i from P_1 is also present in the combined problem (though its timing is changed, see Section 6.7.2.3). The only difference in the activities from the original problem is that some of them now produce intermediate inventories that are then consumed by activities originally in P_2. The supply events for the P_1 raw materials are still present in the combined problem.

For the activities in P_2PP_j, all consumption and production remains the same as in P_2. The only difference is that the supply events for the raw materials no longer exist as their production has been taken over by activities from P_1.

6.7.2.2 Resource Combination

Rather than doubling the resources, as is done with the inventories, the combination process only uses the resources from P_1. The activities from P_2 are randomly assigned to execute on a P_1 resource with the usual job shop restriction that each activity in a process plan executes on a different resource.

6.7.2.3 Temporal Combination

The only change in the temporal characteristics of the problems is the calculation of the end of the horizon and the timing of the demand events for the finished goods inventories.

For two-stage problems, we find the lower bound on resource usage by summing the durations of the activities on each resource. This lower bound is then multiplied by the makespan factor to generate the length of the scheduling horizon. Each demand event is scheduled to occur at the end of the horizon regardless of whether the finished good inventory originally came from P_1 or P_2.

6.7.2.4 Summary

The combination of two $n \times n$ problems results in a single problem with n resources, $4n$ inventories (n raw materials, n intermediate inventories, and $2n$ finished goods), and $2n$ process plans each with n activities. There are $5n$ supply events (five for each raw material in P_1) and $2n$ demand events (one for each finished good in P_1 and P_2).

The resource combination in particular requires that the original problems must have the same number of resources. In practice, we only combine an $n \times n$ problem with another $n \times n$ problem.

6.8 Empirical Evaluation

The primary goal of the experiments in this chapter is to evaluate the use of the texture-based inventory heuristic commitment techniques. Before addressing this goal, however, it is necessary to evaluate the usefulness of the inventory propagators proposed in this chapter. Therefore, in Experiment 1 we evaluate their use in three different experimental conditions depending on the heuristic commitment technique that is used.

Experiment 2 turns directly to the evaluation of inventory heuristics by using the same set of propagators for all problems. We use two problem sizes (5×5 and 10×10) in separate experimental conditions and hold the makespan factor for all problems constant at 1.2 while varying the average number of inventories consumed by activities in each process plan. The makespan factor was chosen based on the results of Chapter 4 that most of the random job shop problems are soluble at such a factor. The goal here (as discussed in Section 4.5.4) is to generate sets of problems with different levels of difficulty. Given that most of the underlying job shop problems at makespan factor 1.2 are soluble, we expect to observe the easy-hard-easy pattern as we increase the number of consumers per process plan. With a low number of consumers, a solution will be relatively easily found while at a high number of consumers, a proof that the problem is overconstrained will be easily found. Between these extremes we will see the more difficult problems.

In Experiment 3, we make two further manipulations to the problem sets. First, we move to two-stage problems in order to test algorithm performance when there are more complex producer-consumer relationships in the problems. Second, we vary the makespan factor while holding the number of consumers per process plan constant. The number of consumers is set at a level that, based on Experiment 2, represents a point in the parameter space with relatively difficult problems. The makespan factor is varied between 1.0 and 1.5 in order to examine the performance of the algorithms as the tightness of the resource constraints vary.

6.9 Experiment 1: Inventory Propagators

In Experiment 1, we examine a subset of the heuristic commitment techniques with and without the inventory propagators. Our goal is to evaluate the usefulness of the propagators. Further experiments then compare the heuristic commitment techniques.

We use the following six instantiations of the ODO framework in Experiment 1 (see Table 6 for a full description): VarHeight, VarHeightProp, GreedySumHeight, GreedySumHeightProp, CBASlack, and CBASlackProp.

The problems in Experiment 1 are one-stage 10×10 problems with the independent variable being the number of consumptions per process plan. The independent variable takes on the following

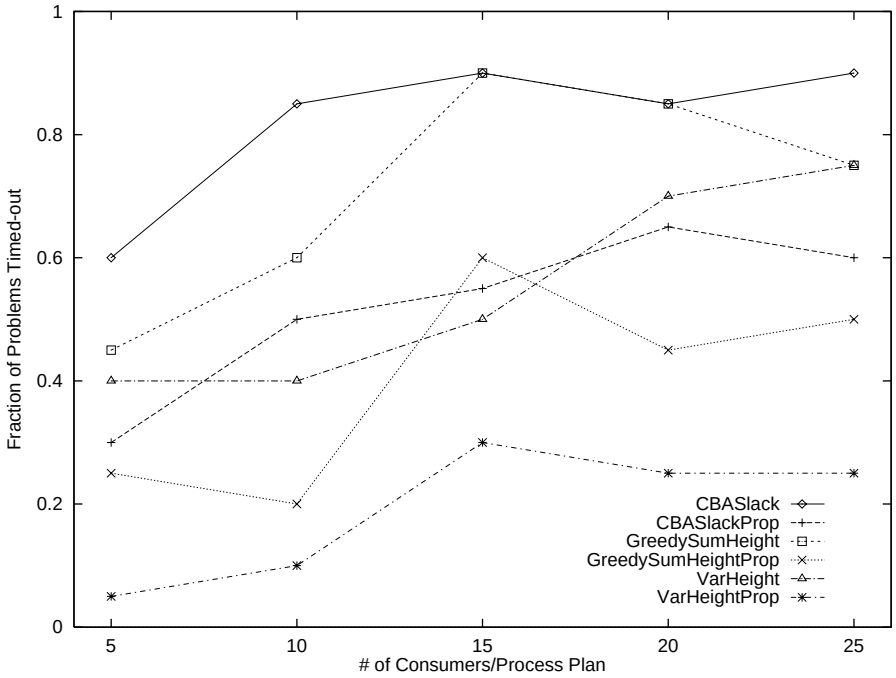


Figure 86. The Fraction of Problems Timed-out for Each Problem Set and Algorithm.

values: 5, 10, 15, 20, 25. For each value of the independent variable and for each problem size, twenty problem instances are generated. The makespan factor for all problems is 1.2. This value was chosen so that we could be confident that the underlying job shop problems (without inventory constraints) are not overconstrained. Overconstrained problems, if any, arise from the inventory constraints or the combination of the inventory and temporal constraints. This ensures that reasoning about inventory is necessary in solving the problems.

6.9.1 Results

Tables with all results from Experiment 1 can be found in Appendix C Section C.1.

The proportion of problems that each algorithm timed-out on for each problem set is shown in Figure 86. The mean CPU time for each algorithm across the problem sets is shown in Figure 87. While there is some overlap, the algorithms using inventory propagation tend to cluster in the lower parts of each graph indicating better performance (*i.e.*, timing out on fewer problems and using less mean CPU time).

Statistical analysis[6] indicates that the algorithms that use inventory propagation time-out on significantly fewer problems than the corresponding algorithms that do not use propagation. The CPU results echo the timed-out results as, again, all algorithms using inventory propagators incur significantly less mean CPU time than their counterparts without inventory propagation.

On all other search statistics evaluated (number of backtracks, number of heuristic commitments, and number of commitments) the algorithms using inventory propagators perform better. The dif-

6. As in the previous chapters, we measure statistical significance with the boot-strap paired-t test [Cohen, 1995] with $p \leq 0.0001$ (unless otherwise noted).

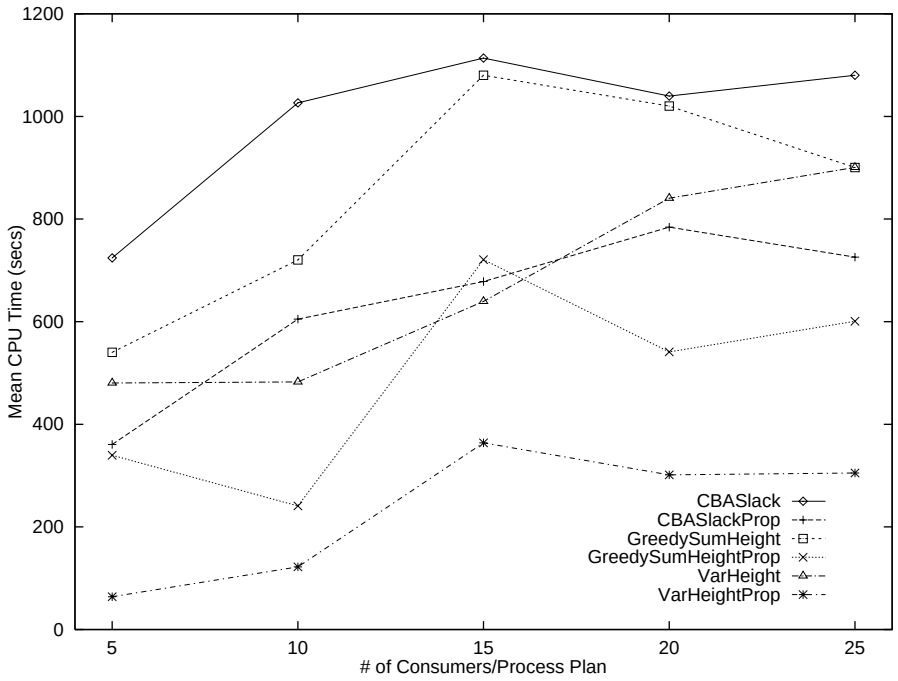


Figure 87. The Mean CPU Time in Seconds for Each Problem Set and Algorithm.

ferences are all highly significant except for the total number of commitments where the difference is not significant for CBASlack and is significant at $p \leq 0.001$ for GreedySumHeight.

These results indicate that the use of inventory propagators is beneficial to the overall problem solving effort both in terms of expanding the number of problems for which a solution can be found and in terms of reducing the effort to finding a solution. While there is still work to be done in the evaluation of these propagators and the conditions under which they contribute to the overall problem solving, we leave this to future work. These results are consistent with the results regarding propagation techniques in the literature and, as our primary focus in this dissertation is the use of heuristics, we do not investigate the use of inventory propagators further in this dissertation.

6.10 Experiment 2: One-Stage Problems

6.10.1 Algorithms

We now turn to the examination of the heuristics for inventory scheduling. All algorithms used in Experiment 2 use the Inventory propagators as the results from Experiment 1 show them to be worthwhile.

We are primarily interested in evaluating the use of texture measurement-based heuristics that calculate the criticality of both resource and inventory constraints. The VarHeightProp algorithm (see Table 6) is the only algorithm that uses such a heuristic in Experiment 2. For comparison to VarHeightProp, we use algorithms with both non-texture inventory heuristics (GreedySumHeightProp, GreedyCBASlackProp, and GreedyEorPProp), and non-inventory heuristics (SumHeightProp, CBASlackProp, and EorPProp). The latter class of heuristics are typical of

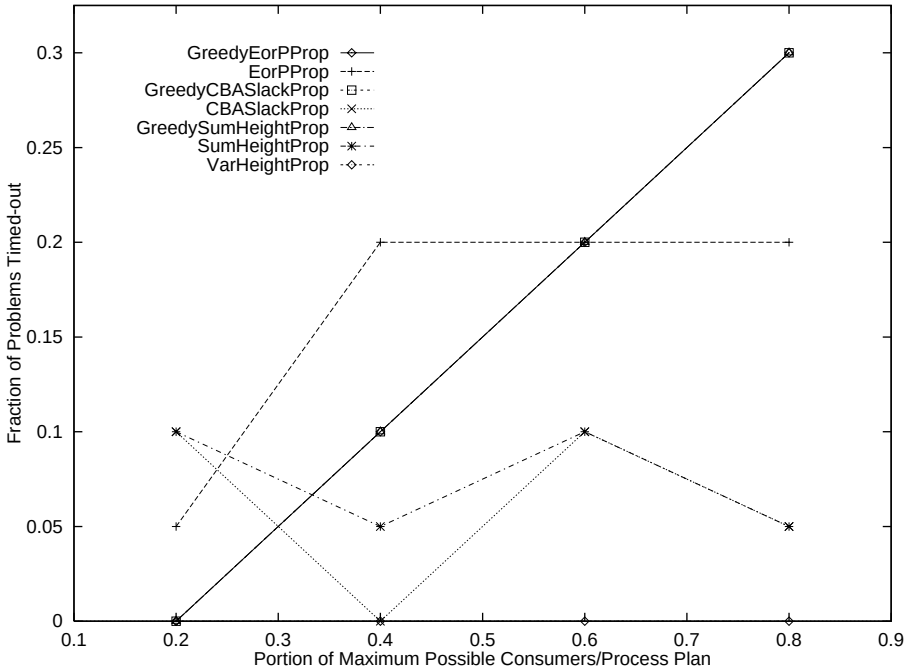


Figure 88. The Fraction of Problems Timed-out for Each Problem Set and Algorithm.

those found in the literature: scheduling commitments are made based on the resource requirements and depend on the inventory propagation to maintain the inventory constraints.

6.10.2 Problems

The problems used in this experiment are one-stage inventory scheduling problems generated as described in Section 6.7.1. Two sizes of problems are used: 5×5 and 10×10. The makespan factor for all problems is held constant at 1.2 and the primary independent variable is c, the number of consumers per process plan. Given the differing problem sizes, rather than directly manipulating c, we manipulate c/n^2, the proportion of the maximum possible consumers for each process plan (see Section 6.7.1). For our experiments, c/n^2 varies from 0.2 to 0.8 in steps of 0.2.[7] For each value of c/n^2 and for each problem size, twenty problems are generated.

6.10.3 Results

Full results from Experiment 2 can be found in Section C.2 of Appendix C. More specifically, the results from the 5×5 problems can be found in Section C.2.1 while the results from the 10×10 problems are in Section C.2.2.

6.10.3.1 5×5 Problems

For the 5×5 problems the percentage of problems timed-out for each algorithm is shown in Figure 88 while the mean CPU time in seconds is displayed in Figure 89. Note that the scales of the vertical axes in these graphs are set to better show the results.

7. In the 5×5 problems these values correspond to c values of 5 to 20 in steps of 5. In the 10×10 problems the c values are 20 to 80 in steps of 20

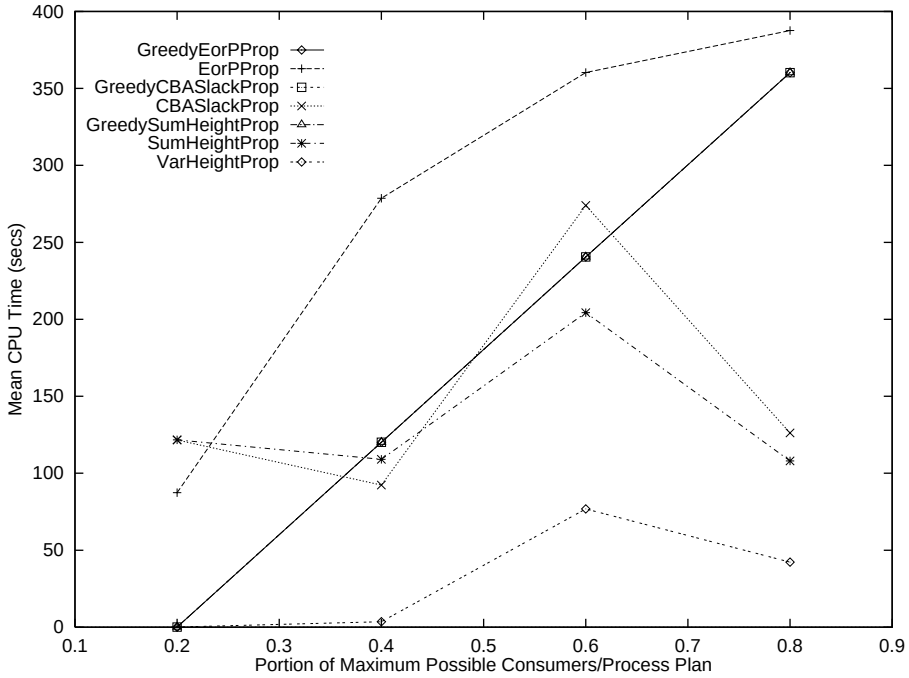


Figure 89. The Mean CPU Time in Seconds for Each Problem Set and Algorithm.

The similarity in performance among a number of the algorithms obscures the results and so further description is in order. In Figure 88, VarHeightProp does not time out on any of the problems. All algorithms using non-texture-based inventory heuristics (GreedySumHeightProp, GreedyCBASlackProp, and GreedyEorPProp) achieve identical performance and so the plots of their results coincide on the plot that ends at (0.8, 0.3). The plots of SumHeightProp and CBASlackProp also coincide on all problem sets except 0.4. The final plot, starting at (0.2, 0.05) represents the data for the EorPProp algorithm. In Figure 89, the algorithms using the non-texture-based inventory heuristics again achieve identical performance and so coincide (within a few hundredths of a second) on the plot ending at (0.8, 360.2). This identical performance is discussed below (Section 6.12).

In our statistical analysis, we examined the differences between VarHeightProp and all other algorithms, and the pair-wise differences between each algorithm using non-texture-based heuristics and its counterpart using non-inventory heuristics (*e.g.*, GreedySumHeightProp versus SumHeightProp). VarHeightProp times-out on significantly fewer problems than all of the algorithms using non-texture-based inventory heuristics ($p \leq 0.0005$) and on significantly fewer problems than EorPProp. There are no significant differences among VarHeightProp, SumHeightProp, and CBASlackProp. In terms of CPU time, however, VarHeightProp incurs significantly less CPU time than all other algorithms ($p \leq 0.005$) except SumHeightProp where there is no significant difference. There are no significant differences in mean CPU time among the other pairs of algorithms examined.

The other search statistics reveal that VarHeightProp makes significantly fewer backtracks and heuristic commitments ($p \leq 0.005$) than all other algorithms. In terms of the total number of commitments, VarHeight makes significantly fewer than all other algorithms ($p \leq 0.005$) except CBASlackProp and SumHeightProp where there are no significant differences. In comparing the algorithms using non-texture-based heuristics with their counterparts using non-inventory heuristics the only significant differences (all at $p \leq 0.005$) are that:

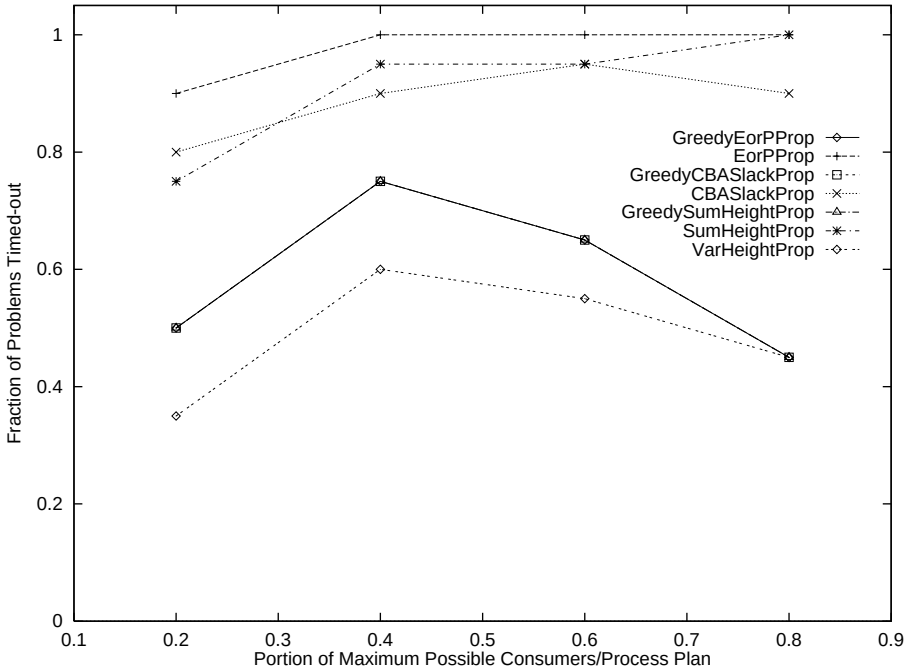


Figure 90. The Fraction of Problems Timed-out for Each Problem Set and Algorithm.

- GreedyEorPProp makes significantly fewer heuristic commitments than EorPProp.
- SumHeightProp makes significantly fewer total commitments than GreedySumHeightProp.
- CBASlackProp makes significantly fewer total commitments than GreedyCBASlackProp.

6.10.3.2 10×10 Problems

For the 10×10 problems, the timed-out results are presented in Figure 90 while the mean CPU time results are in Figure 91. Again, there is similar performance among a number of the algorithms, obscuring some of the results.

In Figure 90, all the algorithms using the non-texture-based inventory heuristic achieve the same results and coincide on the plot beginning at (0.2, 0.5). In Figure 91, we again observe that the algorithms using the non-texture inventory heuristics achieve identical performance within a few hundredths of a second and coincide on the plot beginning at (0.2, 601.0). The results for the other algorithms are clear in both graphs.

Statistically, VarHeightProp times-out on significantly fewer problems than SumHeightProp, CBASlackProp, and EorPProp while showing no significant difference when compared to the algorithms using non-texture-based inventory heuristics. In comparing the corresponding pairs of algorithms, we see that each non-texture-based inventory heuristic times-out on significantly fewer problems than its non-inventory heuristic counterpart ($p \leq 0.0005$).

Turning to the CPU time results, the statistical analysis reveals that VarHeightProp incurs significantly less CPU time than each of the algorithms using non-inventory heuristics ($p \leq 0.0005$). Examining the corresponding pairs of non-texture heuristics versus non-inventory heuristics, we observe that each algorithm using a non-texture inventory heuristic incurred significantly less CPU time than the corresponding one using non-inventory heuristics.

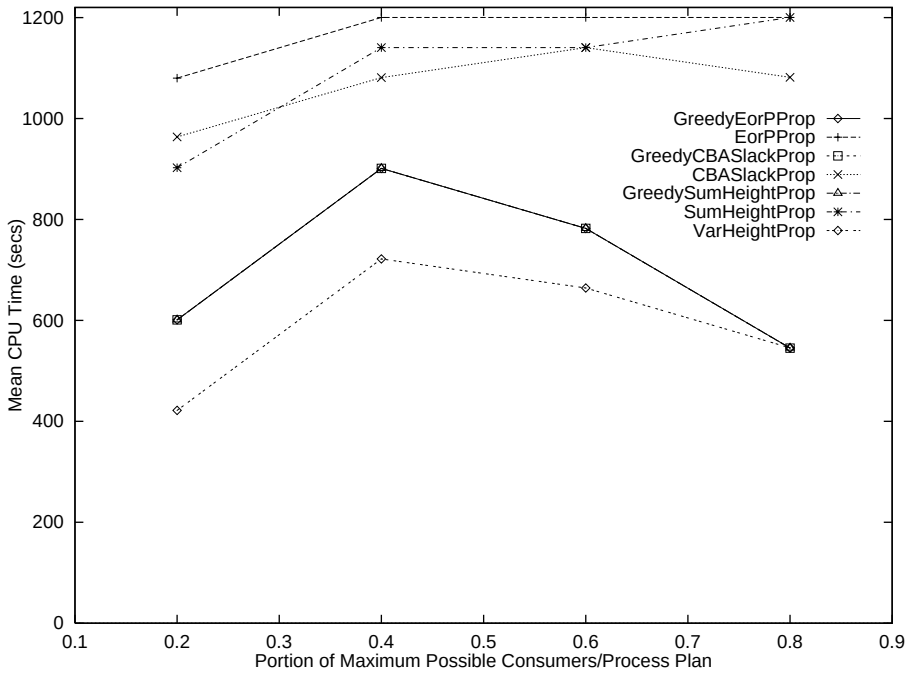


Figure 91. The Mean CPU Time in Seconds for Each Problem Set and Algorithm.

The other search statistics indicate that VarHeightProp makes significantly fewer backtracks and fewer heuristic commitments than each of the non-inventory heuristics ($p \leq 0.0005$). These results are also seen for the number of overall commitments except for the comparison between VarHeightProp and CBASlackProp which shows no significant difference. There are no significant differences on these other statistics between VarHeightProp and any of the non-texture-based inventory heuristics. The heuristic commitment and overall commitment statistics show that each of the algorithms using non-texture-based inventory heuristics outperforms its non-inventory heuristic counterpart ($p \leq 0.001$ for the overall commitments). In terms of the number of backtracks, the only other significant difference is that GreedyEorPProp incurs significantly fewer than EorPProp.

6.10.4 Summary

The results of Experiment 2 indicate that:

- The non-inventory heuristics (SumHeightProp, CBASlackProp, and EorPProp) do not, in general, perform as well as the algorithms that take into account the inventory constraints when making heuristic commitments.
- The algorithms using the non-texture-based inventory heuristics (GreedySumHeightProp, GreedyCBASlackProp, and GreedyEorPProp) achieve identical performance including mean CPU time results within a few hundredths of a second of each other. We discuss the reasons for this below (Section 6.12).
- The performance of the VarHeightProp is as good as or better than all other algorithms.

6.11 Experiment 3: Two-Stage Problems

Experiment 3 uses two-stage problems as described in Section 6.7.2. These problems introduce more inventory complexity than the one-stage problems used above. Again our primary goal is the evaluation of the inventory heuristics. We use the same algorithms used in Experiment 2.

The experiment problems are again divided into two experimental conditions corresponding to the size of the original one-stage problems. We refer to each set by the size of the original problems. Each problem in the 5×5 problem set is generated from the combination of two one-stage 5×5 problems while each 10×10 problem results from the combination of two one-stage 10×10 problems.

In all problems, the proportion of possible consumers (c/n^2) is held constant at 0.4.[8] The primary independent variable for these problems is the makespan factor which is varied from 1.0 to 1.5 in steps of 0.1. Recall that the makespan factor is multiplied by the resource lower bound to give the end of the horizon in each problem. For each makespan factor and each problem size, twenty two-stage problems are generated.

6.11.1 Results

Full results from Experiment 2 can be found in Section C.3 of Appendix C: the results from the 5×5 problems can be found in Section C.3.1 while the results from the 10×10 problems are in Section C.3.2.

6.11.1.1 5×5 Problems

Results of the 5×5 experiments are shown in Figure 92 and Figure 93. The former graph displays the proportion of problems for which each algorithm timed-out while the latter displays the mean CPU time for each algorithm. As in Experiment 2, the algorithms using non-texture-based inventory heuristics produce very similar results. In both Figure 92 and Figure 93, the plots corresponding to them coincide except for the results for makespan factors 1.3 and 1.5.

Statistical analysis indicates that VarHeightProp times-out on significantly fewer problems and incurs significantly less mean CPU time than all other algorithms. The only significant difference comparing the non-texture-based inventory heuristics to their non-inventory heuristic counterparts is that GreedyCBASlackProp incurs significantly less mean CPU time than CBASlackProp ($p \leq 0.005$).

The other search statistics, in general, agree with the timed-out and CPU results:

- In terms of the number of backtracks, VarHeightProp makes significantly fewer than all other heuristics, and each of the non-texture-based inventory heuristics makes significantly fewer backtracks than its corresponding non-inventory heuristic ($p \leq 0.005$ for GreedySumHeightProp versus SumHeightProp).
- VarHeightProp makes significantly fewer overall commitments than all non-texture-based inventory heuristics and than EorPProp. Interestingly, GreedyCBASlackProp and GreedySumHeightProp make significantly more overall commitments than CBASlackProp and Sum-

8. Each 5×5 problem, therefore, has 10 consumptions per process plan while each 10×10 problem has 40.

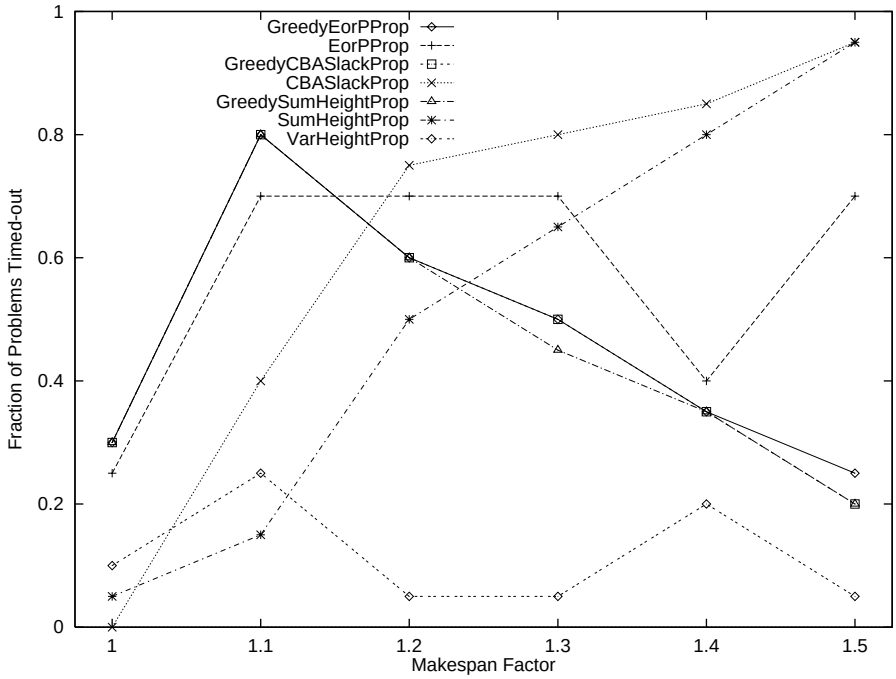


Figure 92. The Fraction of Problems Timed-out for Each Problem Set and Algorithm.

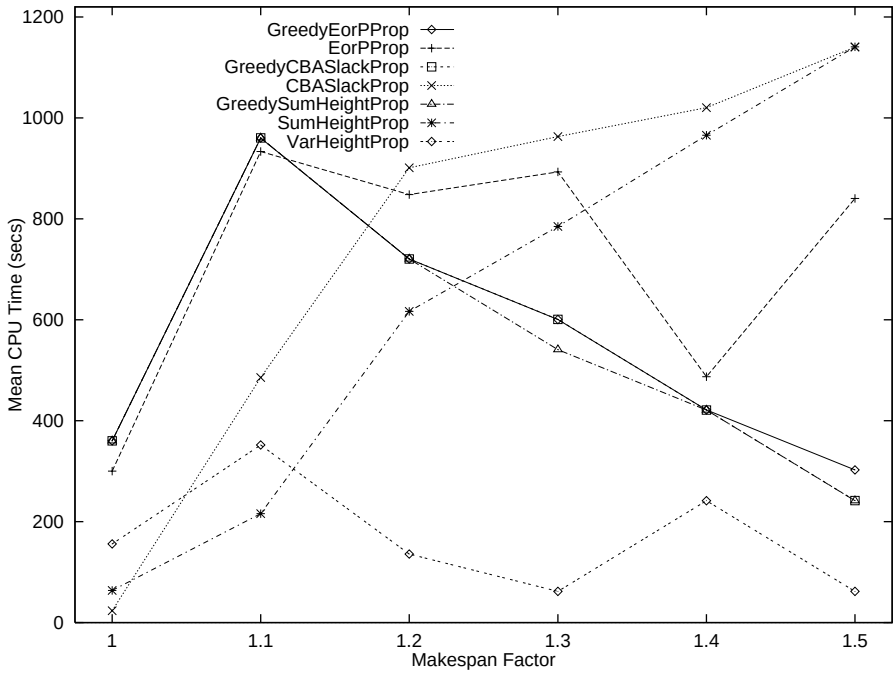


Figure 93. The Mean CPU Time in Seconds for Each Problem Set and Algorithm.

HeightProp respectively. There are no other significant differences for the overall commitment results.

- Turning to the heuristic commitments, VarHeightProp again makes significantly fewer than all other algorithms while each of the non-texture-based inventory heuristics makes significantly fewer heuristic commitments than its corresponding non-inventory heuristic ($p \leq 0.0005$).

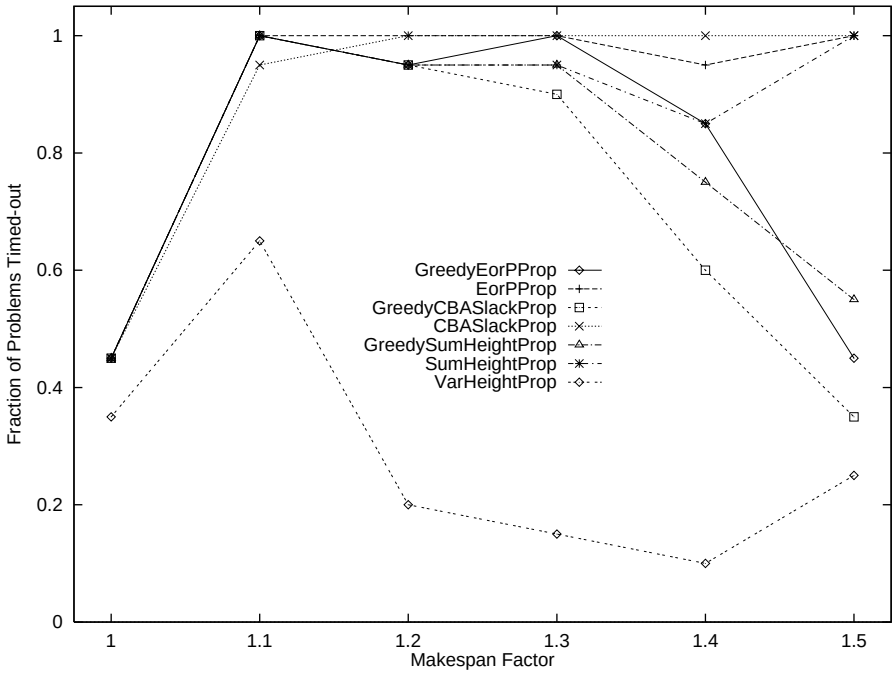


Figure 94. The Fraction of Problems Timed-out for Each Problem Set and Algorithm.

6.11.1.2 10×10 Problems

The proportion of problems in each problem set that the algorithms time-out on is shown in Figure 94. All algorithms except VarHeightProp exhibit identical performance on the problems at makespan 1.0, but diverge at higher makespan factors. In general, the algorithms using non-inventory heuristics do poorly on all problem sets from makespan factor 1.1 to makespan factor 1.5. The algorithms using non-texture-based inventory heuristics also perform poorly on problem sets with low makespan factors (1.1 and 1.2), but improve as the makespan factor increases (we discuss reasons for this below). VarHeightProp times-out on fewer problems than each of the other algorithms on all problem sets.

The mean CPU time for each algorithm and problem set is shown in Figure 95. The results, here, mirror those in Figure 94. VarHeightProp incurs lower mean CPU time than all other algorithms across all problem sets. The other algorithms perform close to the same on the problem sets with low makespan factor, but diverge at higher makespan factors. As above, the algorithms with non-texture-based heuristics appear to improve (relative to the non-inventory heuristic algorithms) with the higher makespan factors.

Statistical analysis reveals that VarHeightProp times-out on significantly fewer problems and incurs significantly less mean CPU time than all other algorithms. In addition, VarHeightProp makes significantly fewer backtracks and significantly fewer heuristic commitments than all other heuristics. In terms of overall commitments, however, VarHeightProp significantly outperforms only GreedyCBASlackProp ($p \leq 0.0005$), CBASlackProp ($p \leq 0.005$), GreedySumHeightProp, and SumHeightProp.

In comparing the non-texture-based heuristics with their non-inventory heuristic counterparts, we see that the former significantly outperform the latter ($p \leq 0.005$) in terms of the number of problems timed-out, the mean CPU time, the number of backtracks, and the number of heuristic commitments. For overall commitments, the only significant difference between these pairs of

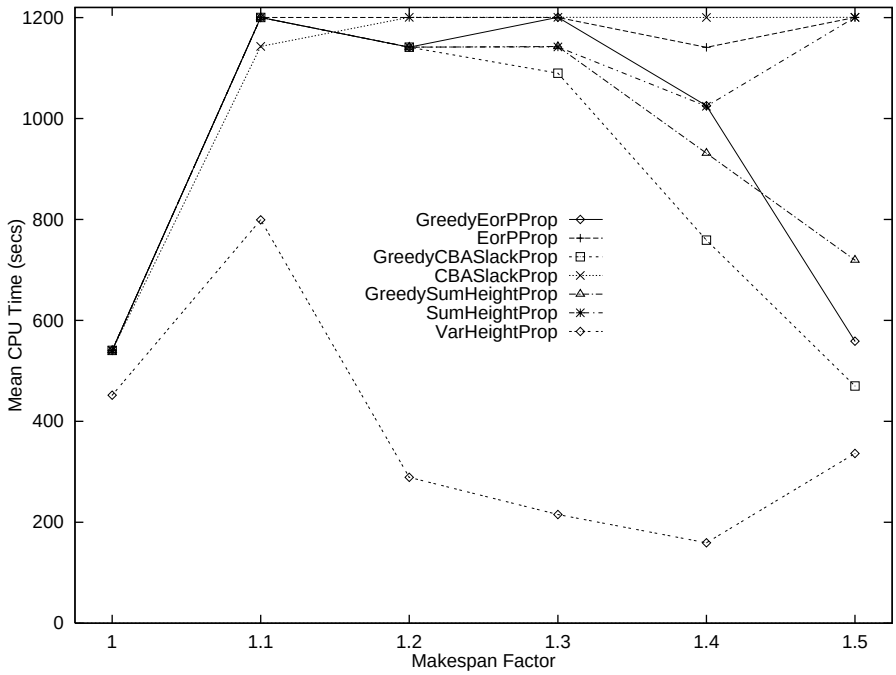


Figure 95. The Mean CPU Time in Seconds for Each Problem Set and Algorithm.

algorithms is that GreedySumHeightProp makes significantly fewer overall commitments than SumHeightProp.

6.11.2 Summary

In summary, Experiment 3 indicates that:

- VarHeightProp outperforms all the other algorithms.
- The algorithms using non-texture-based inventory heuristics tend to outperform their non-inventory counterparts. This performance difference is especially clear on problems with higher makespan factors.

6.12 Discussion

The experiments in this chapter were primarily concerned with the evaluation of heuristics for inventory scheduling and in validating the use of inventory propagation techniques. We look at each of these goals before briefly examining the relaxation of the requirement that inventory constraints remain constant across the scheduling horizon.

6.12.1 Inventory Heuristics

Experiments 2 and 3 show that the use of texture-based heuristics results in as good as or better performance than non-texture-based inventory heuristics and non-inventory heuristics. In particular, when the inventory relationships among activities are made more complex (as in the two-stage problems of Experiment 3), the algorithm using a texture-based heuristic significantly outperforms all other algorithms on all problem sets tested. We interpret these experimental results as

positive evidence toward the use of texture-based heuristics in constraint-directed scheduling algorithms.

6.12.1.1 Dynamic Focus on Critical Constraints

Recall that the motivation for a measure of criticality that can be applied to a wide range of constraints is that, at each search state, we want to be able to identify the most critical constraint, regardless of constraint type, and focus our heuristics to reducing the criticality. It is not only the ability to estimate criticality that leads to high-quality heuristic search, but also the ability to dynamically focus on the most critical constraint at each search state.

The contributions of the dynamic focus ability are particularly evident in Experiment 3. As we argue below (Section 6.12.1.3), as the makespan factor increases the relative criticality of the resource and inventory constraints changes. While the resource constraints are, on average, more critical for low makespans, the inventory constraints are, on average, more critical for high makespans. Throughout this change in relative criticalities VarHeightProp outperforms all other algorithms. We interpret these results as clear support for our approach to scheduling via texture measurement-based heuristic search. The ability to not only identify the constraints that are more critical at the beginning of the problem but also in every search state allows the texture-based heuristics to achieve high-quality search on problems with varying characteristics.

6.12.1.2 The Probability of Breakage as a Measure of Criticality

In Chapter 5, the results on job shop scheduling problems indicated that the probability of breakage measure of criticality only partially meets the requirement of practical utility. We speculated (Section 5.7.2) that the aggregate demand may represent information important to the search that is not present in probability of breakage measurement of criticality.

In this chapter, we see that when inventory constraints are modeled, the wider applicability of probability of breakage (*i.e.*, the fact that it meets the first three of our requirements for a measure of criticality (see Section 5.1.3)) results in superior overall performance. Therefore, in the presence of both inventory and resource constraints, the probability of breakage meets the requirement of practical utility.

6.12.1.3 Simple Inventory Heuristics

Experiments 2 and 3 demonstrated that taking inventory into account when making heuristic decisions even with a greedy heuristic leads to better overall search performance than when the heuristics focus solely on resource constraints and allow propagators to maintain the inventory constraints. This is an important point in view of the fact that the few scheduling strategies in the literature that address inventory problems tend to account for inventory via inventory propagators alone. They do not make the information in the inventories and inventory constraints part of the heuristic decision making process. Our results show that even a simple inventory heuristic like the GreedyInv heuristic can lead to improved performance.

The manipulation of makespan factor in Experiment 3 provides an interesting basis on which to compare the non-texture-based inventory heuristics and the non-inventory heuristics. For low values of the makespan factor, there is little slack on each resource, and therefore the criticality of the resource constraints is high. When the makespan is larger, however, there is more slack time on each resource. The resource constraints are easier to satisfy and therefore the inventory constraints become, relatively, more critical.

On this basis, then, we would predict that a heuristic that focuses on resource constraints will tend to perform well for problem sets with low makespan factors while at larger factors, heuristics that put more effort toward the inventory constraints will be superior. These are precisely the results we see: the non-inventory heuristics perform well at makespan factors 1.0 and 1.1 (especially on the 5✕5 – Figure 93) and very poorly on the problem sets with makespan factors 1.4 and 1.5. The non-texture-based inventory heuristics, in contrast, improve with higher makespan factors.

We take these results to indicate that the ability to even *statically* focus on inventory constraints results in better heuristic search when inventory constraints are more critical. Of course, as noted in the previous section, one of the key advantages of texture-based heuristics such as VarHeight-Prop is that they dynamically focus on whatever constraint is most critical at a search state

6.12.1.4 The Performance of the Non-Texture-Based Inventory Heuristic

Finally, there is another characteristic of our experimental results concerning heuristics that deserves explanation. In many of the problem sets, the performance of the algorithms using the non-texture-based inventory heuristics was identical within a few hundredths of a second in mean CPU time. Given that each algorithm (GreedySumHeightProp, GreedyCBASlackProp, and GreedyEorPProp) uses very different resource heuristics, the identical results are curious. Deeper analysis of the problem solving revealed that each of the algorithms was almost completely dependent on the GreedyInv heuristic. Recall that these algorithms use the GreedyInv heuristic to form producer/consumer commitments. The GreedyInv heuristic demonstrated one of two behaviors in many of the experimental problems.

1. The GreedyInv heuristic was able to find a set of producer/consumer commitments that could be extended to an overall solution with no further backtracking regardless of the resource heuristic used.
2. The GreedyInv heuristic was not able to find a set of consistent producer/consumer commitments at all.

In the first case, identical performance was achieved by each algorithm in solving the problem while in the second, identical performance was achieved in failing to solve the problem.

6.12.2 Inventory Propagators

Experiment 1 compared a number of algorithms with and without inventory propagation techniques. Regardless of the heuristic (or type of heuristic) used, the algorithms using inventory propagators outperformed those that did not use them. This result is not surprising given the literature which shows significant advantages in the use of propagation techniques in constraint-directed scheduling.

6.12.3 Appropriateness of the Experimental Problems

Our interpretation of the experimental results in this chapter relies on the appropriateness of the experimental problem sets. These problems were designed to test the quality of reasoning about both resource and inventory constraints in the same problem. If these problem sets do not test such reasoning abilities, the results we have observed must be attributed to other causes.

There are two main patterns we observe in our results that lend support to our claim that our problem sets are appropriate for their purpose. The first, shown explicitly in Experiment 1, is that inventory propagation techniques significantly contribute to the overall search performance. That

we observed such results indicates that reasoning about inventory constraints is part of what is necessary to perform well on the problem sets.

The second pattern in our results is the one noted above in regards to the behavior of the non-texture-based inventory heuristics and the non-inventory heuristics as the makespan factor of the problems is varied. In problem sets where the resource constraints were tight (low makespan) the non-inventory heuristics performed as well as or better than the non-texture-based inventory heuristics while with looser resource constraints (high makespan) the opposite performance comparison is observed. This pattern indicates that reasoning about both resource and inventory constraints is necessary (in different problem sets) to achieve high search performance.

Finally, it must be noted that no problem sets including inventories and inventory constraints have been found in the literature. While this does not speak directly to the appropriateness of the problems generated in this chapter, it does prevent comparison among problems generated by other researchers. Comparison, both of the problems themselves and of any algorithmic performance differences among problem sets, is a valuable tool in the evaluation of the problem sets and algorithms. Further work, with problem sets of widely varying parameters, is required to allow such comparison.

6.12.4 Allowing Varying Inventory Constraints

One of the assumptions for the inventory propagators and texture measurements presented in this chapter is that the inventory constraints are constant across the scheduling horizon. This is not necessarily the case in a real work scheduling environment where it may be desirable to build inventory levels (*i.e.*, increase the minimum constraint) in anticipation of a surge in demand.

Varying inventory constraints do not pose a major problem to the techniques described here. Recall that both the texture measurements and inventory algorithms are based around an ordered list of events corresponding either to the estimated inventory levels, or their upper or lower bounds. Varying inventory constraints can be incorporated by adding events to these lists corresponding to the times at which the inventory constraint changes. Since the algorithms evaluate the constraint value at each event point, the added events are all that is required. The complexity of these algorithms will now not only depend on the number of producers and consumers of each inventory, but also on the number of times that the inventory constraints change over the horizon.

6.13 Conclusions

In this chapter, we applied one of the extended texture measurements introduced in the previous chapter (VarHeight) to the problem of scheduling with inventory production, consumption, and storage constraints. In addition to the creation of a constraint-based representation of inventory requirements, we also formulated two inventory propagators and a number of heuristics for inventory scheduling, both texture-based and non-texture-based.

Our experimental results validate the use of the two inventory propagators presented in this chapter. It was empirically shown that significantly better overall search performance can be achieved by the use of texture-based heuristics. Texture measurements dynamically evaluate the criticality of both resource and inventory constraints at each search state, and allow the heuristic to focus on the constraint that is most critical. Heuristic commitments can then be formulated to expressly address and reduce the criticality.

Chapter 7 Scheduling with Alternative Activities

The final characteristic of realistic scheduling problems investigated in this dissertation is the existence of activity alternatives. Typically, there are choices to be made in scheduling among alternative resources on which to run an activity or among alternative process plans (also called "routings") of an order through a factory. While this characteristic is ubiquitous in industry, there has been little examination of such alternatives in the constraint-directed scheduling literature. In this chapter, we present a unified representation of activity alternatives that addresses both alternative resources and alternative process plans. We show how the representation is incorporated into the constraint-directed scheduling framework developed in this dissertation, and demonstrate how to extend existing heuristics and propagators to reason directly about activity alternatives. Experimental results validate our approach and explore a number of variations in scheduling techniques.

7.1 Introduction

The scheduling problems we have investigated in this dissertation have had a static activity definition: each activity has to be scheduled on its specified resource in order to arrive at a solution. We now expand the scope of constraint-directed scheduling techniques to deal with the case where not only does the scheduler have a choice of when to execute an activity, but also the choice of not to execute it at all.

Note that in this chapter we do not represent and reason about inventory production, consumption, or storage as we did in Chapter 6. From an empirical perspective this limitation allows us to narrow the focus of this chapter to concentrate on the central problem of representation and reasoning about alternatives. In future work, then, our techniques can be broadened by the application to inventories. In addition, from a practical perspective, the dependence of the inventory propagation techniques on the availability of bounds on inventory level presents some problems in application to alternative inventory reasoning. We return to this point in Chapter 8.

7.1.1 Motivation and Problem Definition

It is not uncommon in a realistic scheduling problem to have a number of choices that are not typically represented in constraint-directed scheduling approaches. In particular, there are two extensions of typical research scheduling models that we examine in this chapter: alternative resources and alternative process plans.

Alternative Resources. Given a facility in need of scheduling (*e.g.,* a chemical plant) and difficulty in creating schedules due to high competition for a resource, the company may attempt to

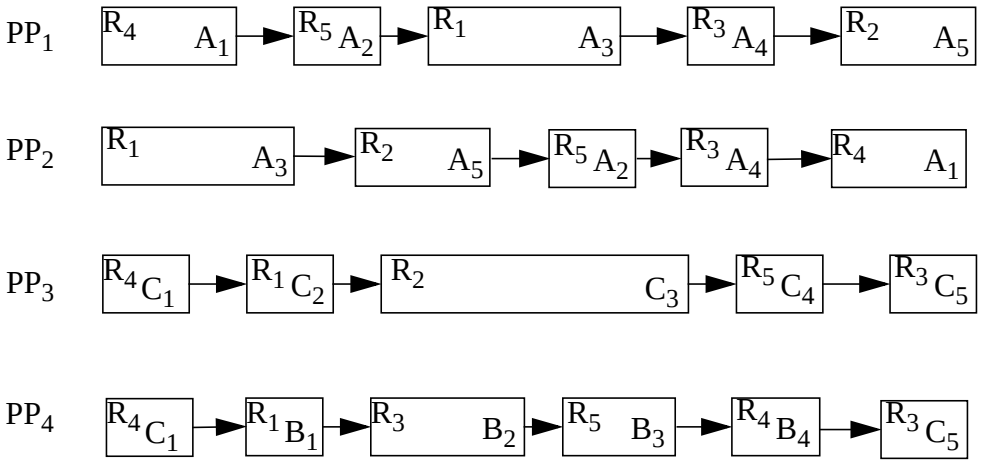


Figure 96. Four Alternative Process Plans.

reduce contention by purchasing an additional resource that can run the same activities as the current bottleneck resource. Using our chemical plant example, additional reaction vessels may be purchased to both expand the capacity of the plant and to loosen the scheduling problem. The existence of an alternative resource, however, introduces the need to decide which activities will be performed on which resource. If the alternatives are truly identical, then it may be possible to represent the alternatives with a single multi-capacity resource[1] [Nuijten, 1994]. However, in many cases the resources are not truly identical: one resource may incorporate new technology and so is able to process activities more quickly, produce higher quality inventory, etc.

Alternative Process Plans. Depending on the flexibility of the production facility, it may be the case that there are multiple ways to achieve the same goal. The different ways may result from flexibility in the sequence of activities, separate processes that result in the same goal, or choices within a process plan. Figure 96 displays four alternative process plans (PP_1, …, PP_4). The label in the upper-left corner of each activity represents the activity's resource requirement while the lower-right label is the name of the activity. Thus, activities with the same name (*e.g.,* A_3 in PP_1 and A_3 in PP_2) are the same. The first two process plans, PP_1 and PP_2, are simply different orderings of the same activities. The third process plan, PP_3, is a completely different recipe while the fourth, PP_4, is a variation on the third: the first and last activities are identical, but the middle ones are different. Only one of the alternative process plans needs to be executed. Therefore, the scheduling problem not only consists of deciding *when* to execute the activities but which of the alternative process plans will be executed.

In this chapter, we show how problems with alternative resources and alternative process plans can be represented and reasoned about with alternative activities.

1. Even with identical resources, however, this is not always possible. If the activities leave some residue on the resource there may be the requirement that the resource is cleaned out between incompatible activities. When a clean-out is required, it is difficult to model the alternatives as a single resource as it is necessary to specify the single real resource where the clean-out will occur. The investigation (and minimization) of clean-outs and their generalization, *changeovers*, forms an important area of future work in constraint-directed scheduling.

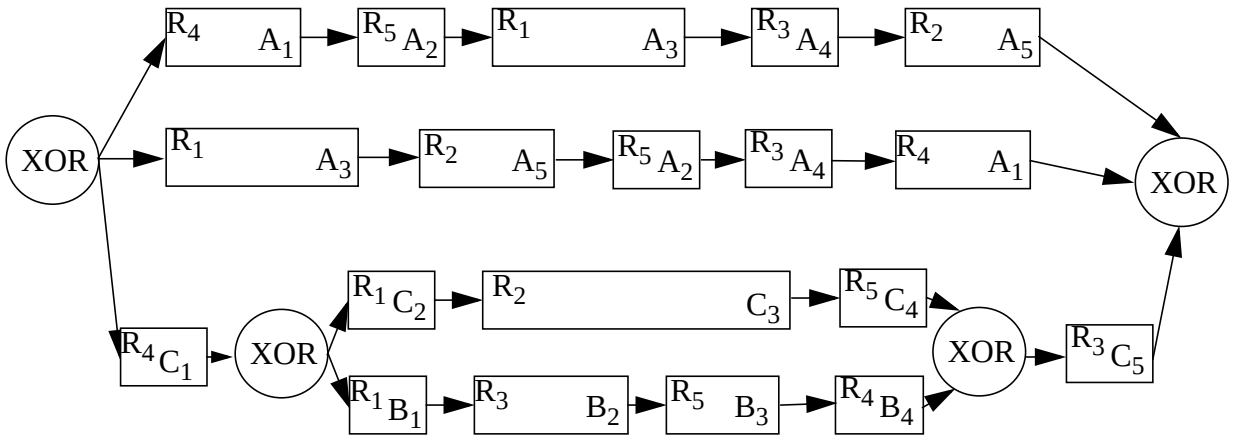


Figure 97. Modification of the Temporal Network to Directly Model the Alternatives Implicit in Figure 96.

7.1.2 Overview of Approach

Our approach to scheduling alternative activities is to represent and directly reason about the fact that an activity present in the problem definition may not be present in a final schedule. To do this, we introduce the notion of an activity's *probability of existence* (PEX), informally defined to be the probability that an activity will be present in the final solution (assuming such a solution exists). PEX is a real number value on the interval [0, 1] with the standard probability interpretation. Each activity has a PEX variable that can be manipulated during the scheduling process. In addition, the temporal network is modified so that PEX values and temporal values are propagated among activities. One of the key components of this extension is a modification of the temporal graph to explicitly represent alternative activities.

While detailed explanations are presented below, a general idea of the representational approach can be seen in Figure 97. The alternative process plans (and sub-process plans) from Figure 96 have been modified by the incorporation of Xor nodes in the temporal graph.

7.2 Probability of Existence

The *probability of existence* (PEX) for an activity is the estimated probability at a point in the search that an activity will be present in a final solution to the problem. The PEX value of an activity is in the range [0, 1] with 1 indicating that an activity will certainly be part of the solution and 0 indicating that it certainly will not.

7.2.1 Desired Functionality

Given a PEX variable on each activity, we need a method of maintaining consistent PEX values among activities related by temporal constraints. Furthermore, given PEX values of less than one during the search, we also need to maintain the temporal aspects of the network correctly, given that some of the activities may later be removed. The small example in Figure 98 serves to illustrate these two challenges. The figure represents a single process plan with a choice of activities:

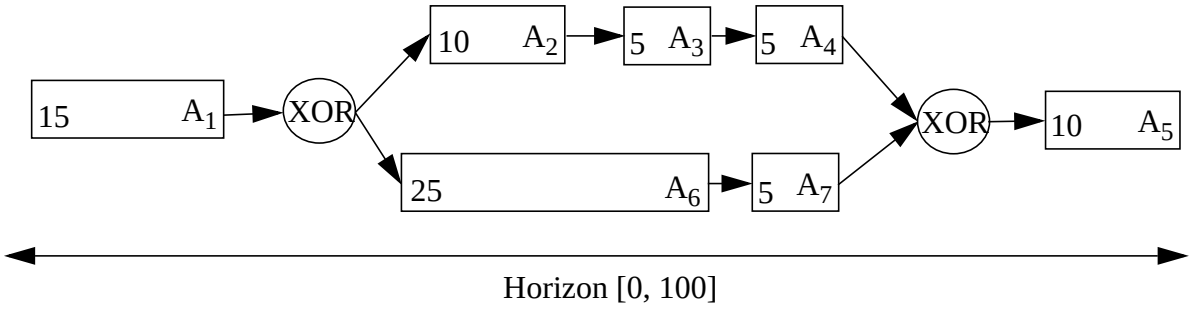


Figure 98. A Process Plan with a Choice of Activities. The duration of each activity is shown in the lower left corner of the activity.

either A_1 can be followed by A_2, A_3, A_4, and then A_5 or A_1 can be followed by A_6, A_7, and then A_5. The duration of each activity is indicated by the number in its lower left corner.

Assuming that A_1 and A_5 must be executed, we assign their PEX values to 1. Further, assuming that there is no information indicating which alternative path is preferred, we split the probability of existence equally between the two choices. Therefore, A_2, A_3, A_4, A_6, and A_7 have initial PEX values of 0.5. The PEX propagation must ensure that if we were to make an arbitrary PEX assignment, such as assigning the PEX value of A_3 to 1, the PEX values of the other related activities would be appropriately reset. In our example, the PEX variables of A_2 and A_4 should also be set to 1, and the PEX variables of A_6 and A_7 should be set to 0. Note that we are working within the options provided by the original definition of the process plan. While it may be logically (or even physically) possible to execute A_3 without A_2, the process plan defines one path from A_1 to A_5 that includes A_3. Therefore, if A_3 is to be executed (which is indicated by assigning its PEX variable to 1), then the other activities in that single path (A_2 and A_4) must also be executed. Similarly, the activities in an alternative path (A_6 and A_7) must not be executed.

The second challenge is the maintenance of temporal information in the graph. Returning to the original starting point with Figure 98 (*i.e.*, before assigning the PEX of A_3 to 1), the standard temporal propagation algorithm (which does not take into account the possibility that an activity in the network may later be removed) would derive the start time window of A_1 to be [0, 45]. The latest start time of A_1 is found by the calculation of the longest path from A_1 to A_5 which happens to include A_6 and A_7. If A_1 were to start at time point 46, and A_6 and A_7 were to be executed, A_5 would end at time point 101, which is after the end of the horizon. The difference in this graph, however, is the presence of the alternative path from A_1 to A_5. It is possible for A_1 to start at time 46 if the path going through A_2, A_3, and A_4 is chosen. In fact, with this choice, A_1 can consistently start as late as time point 55. Clearly, then, we need to modify the temporal propagation (or the temporal network) to account for the possibility that some activities may not be present in the final solution.

Actually, the challenge is slightly greater as we would also like to be able to do the following. Assume that the two alternatives in Figure 98 are still possible and further assume that through some other scheduling decision the earliest start time of A_1 is increased to 46. We would like our temporal propagation to be able to detect that such a change makes one of the alternative paths inconsistent: there is no way that A_6 and A_7 can execute consistently if A_1 starts at time point 46 or later. In other words, we would like our temporal propagation to be able to find implied PEX commitments. In this case the derived PEX commitment sets the PEX variables of A_6 and A_7 to 0.

7.2.2 Limitations on the PEX Implementation

Each activity in the temporal graph has a PEX variable; however, the variable is not a true domain variable in the usual CSP sense. It is not the case, under our interpretation of PEX, that in a final solution the variable can take on any value in the [0, 1] domain: in a solution, all PEX values must be either 0 or 1. In addition, during search we always want the PEX variable to have a single value representing the current estimated probability of existence, rather than a domain of values as is typical of true domain variables.

The semantics of our interpretation of PEX place a number of requirements on the PEX variables during search. The reasoning behind these requirements will be clear as the propagation and search techniques are described below:

- At each search state, a PEX variable has a value on the domain [0, 1].
- In a constructive search, if, in search state, *A*, a PEX variable is set to either 0 or 1, that value must be maintained in all search states below *A*. If the search backtracks to a state before *A*, then the assignment of the PEX variable is undone as with normal domain variables.
- If a PEX variable has not been assigned to either 1 or 0, it varies during search on the domain [0,1] according to the PEX propagation algorithm presented below.

In addition, there are a number of limitations that we have placed on the PEX representation in order to simplify the implementation for our model of alternative activities. Relaxation of these limitations form a significant topic for future work. See Chapter 8 for further discussion.

- As noted above, all PEX variables must have a value of 0 or 1 in a final solution.[2]
- The only commitments that can directly be made on a PEX variable are to assign it to either 0 or 1. PEX propagation will then reset the PEX values of other activities to a value on the [0, 1] interval. For example, this limitation prevents a heuristic commitment technique from posting a constraint that a particular PEX variable has a value of 0.75.
- Each choice that remains to an alternative is equally likely: we assume that we do not have any external knowledge that biases the choices at a XorNode. For example, for an activity with three possible alternative resources, the PEX value of each (assuming the activity itself must execute) is 0.33. If one of the alternatives is removed, the two remaining alternatives each have a PEX value of 0.5.

7.3 Adding PEX to the Temporal Network

In this section, we present the PEX representation and discuss extensions to the temporal activity network to incorporate PEX variables and propagation. The following two sections then present, respectively, the details of PEX propagation (Section 7.4) and the details of temporal propagation within the modified temporal network (Section 7.5).

---

2. This is not necessarily the case with a different interpretation of PEX. For example, in a situation where the duration of an activity is linked to the amount of inventory it produces, PEX can be interpreted as the portion of the full act that will be executed. For example, if a basic activity produces 30 units of inventory by running for 10 hours, a PEX of 0.5 would represent an activity that runs for 5 hours and produces 15 units of inventory. While alternative interpretations and uses for the PEX variable exist, in this dissertation we will limit ourselves to the probability of existence interpretation and so have adopted that as the name of the concept.

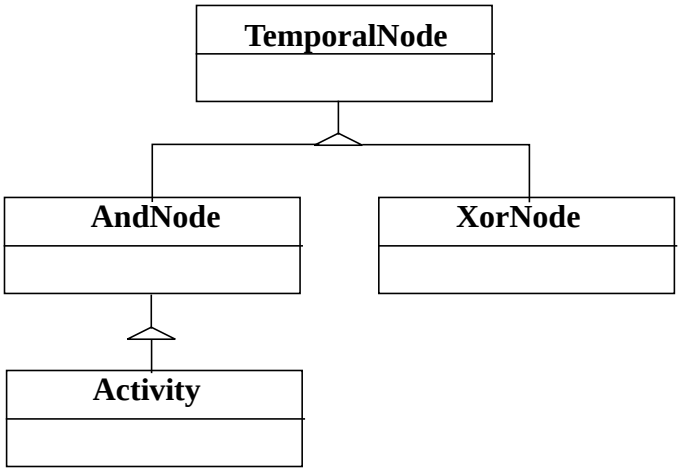


Figure 99. Extensions to the Activity Hierarchy to Implement PEX Functionality.

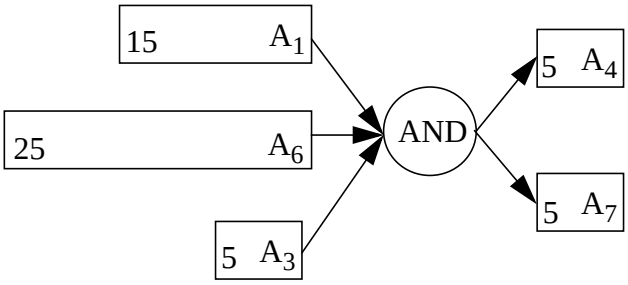


Figure 100. A Sample Temporal Sub-graph with an AndNode.

7.3.1 Extending the Temporal Graph

Figure 99 shows the TemporalNode class hierarchy that was created to replace the existing Activity class. Note that the Activity class is now a sub-class of AndNode. The TemporalGraph has multiple instances of AndNodes, XorNodes, and Activities connected via temporal constraints. In terms of temporal and PEX propagation, an Activity is identical to an AndNode.

7.3.1.1 AndNode

The AndNode has start time and end time temporal variables just as an Activity does. There are two semantic components in the AndNode corresponding to the functionality of the AndNode in temporal and PEX propagation. From a PEX perspective, all the TemporalNodes linked to an AndNode exist in a solution if the AndNode exists. Similarly, if one of the non-XorNodes directly linked to an AndNode exists in a solution, then the AndNode must exist. In terms of PEX values, therefore, all non-XorNodes directly connected to an AndNode must have the same PEX value as the AndNode itself. Since all the nodes an AndNode is connected to are either going to be present in a solution, or all the nodes, including the AndNode itself, are going to be removed, it is valid to perform standard temporal propagation. Figure 100 displays an AndNode connected via precedence constraints to a number of activities. In a consistent network all the displayed nodes must have the same PEX values. In terms of temporal propagation, the AndNode must end before the minimum latest starting time of A_4 and A_7, and must start after the maximum earliest end time of A_1, A_6, and A_3.

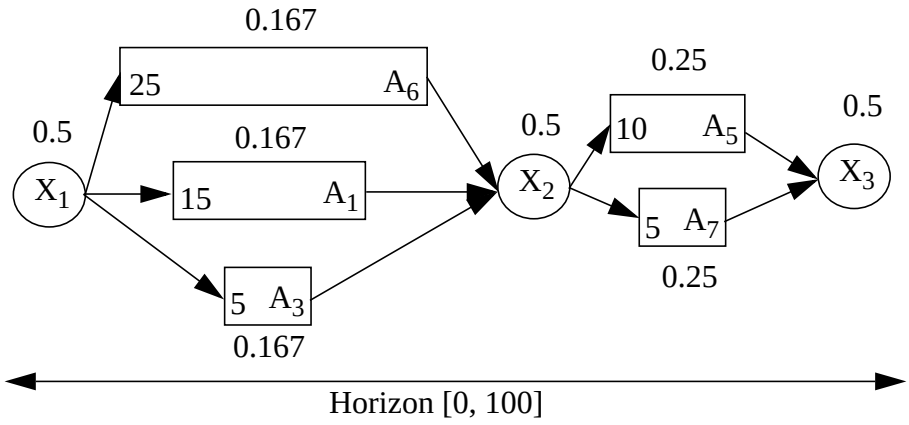


Figure 101. A Temporal Graph with XorNodes.

7.3.1.2 XorNode

As the name indicates, the logical semantics of a XorNode are such that if the XorNode itself is present in the final solution then one and only one of the nodes directly connected upstream to the XorNode can also be present. Similarly, one and only one of the nodes directly connected downstream can be present. Also, if a node directly connected upstream (or downstream) to a XorNode exists in a solution, so must the XorNode. From a PEX propagation perspective, these semantics mean that, in a consistent network, the PEX value of the XorNode is equal to the sum of the PEX values from the nodes directly connected upstream and the XorNode PEX value must also be equal to the sum of the PEX values from the nodes directly connected downstream.[3]

As noted above, for the purposes of this dissertation, we further limit the XorNode behaviour by specifying that all nodes directly connected upstream (or downstream) must have equal PEX values. If a XorNode has a PEX value of *x*, *k* upstream and *l* downstream links, the PEX value of each directly connected upstream node must be *x*/*k* and the PEX value of each directly connected downstream node must be *x*/*l*. The only exception to the rule of even division is when the upstream or downstream node has a PEX variable assigned to 0 or 1. If a neighboring (wolog) upstream node already has a PEX value of 0, it is not included in PEX propagation: the PEX value at the XorNode is simply divided among the upstream nodes whose previous PEX was greater than 0. Similarly, if the neighboring (wolog) upstream node has a PEX value of 1, all the other directly linked upstream nodes must have a value of 0 and so no division is done: the XorNode acts as if it only has a single upstream node.

Figure 101 represents a small temporal graph with XorNodes, X_1, X_2, and X_3. This graph, for example, might be used to represent an alternative resource problem. Assuming that the PEX variable of X_1 and X_3 have been set to 0.5 due to propagation from the rest of the network (not shown), the rest of the temporal nodes have the PEX values shown (above or below each node).

From a temporal perspective, the fact that only one of the upstream and one of the downstream links are to be present in a solution (if the XorNode itself is present) means that temporal propagation is different from that of the AndNode. A XorNode must end before the *maximum* latest start time of all the nodes directly connected downstream while it must start after the *minimum* earliest end time of all activities connected upstream. Assuming that the nodes between X_1 and X_3 must

3. The OPIS scheduler used a similar, though more restrictive, implementation of a XorNode to represent alternative resources [LePape and Smith, 1987].

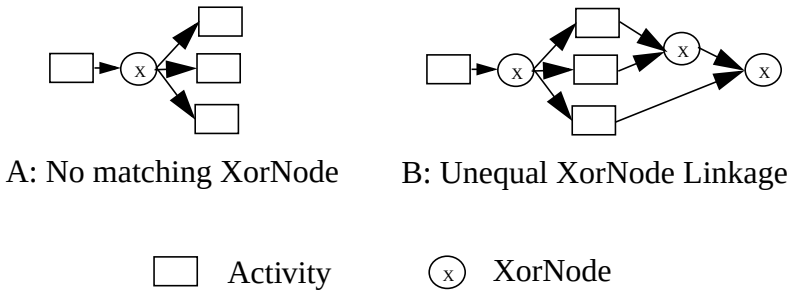


Figure 102. Examples of Illegal Temporal Networks.

be scheduled within a scheduling horizon of [0, 100], Table 7 displays the start and end time windows of each node.

7.3.1.3 Illegal Temporal Networks

We have extended our temporal network representation to allow definition of AndNodes and XorNodes. Not all temporal networks that are expressible are valid networks, however. The basic requirement for legality is that for any XorNode, *X*, in the graph with *k* upstream and *l* downstream links ($k, l > 1$), there must be:

- a corresponding XorNode, X^-, upstream of *X* with *k* downstream links, and
- a corresponding XorNode, X^+, downstream of *X* with *l* upstream links.

Figure 102 displays a number of illegal temporal graphs, while Figure 103 displays a number of legal ones.

Temporal network B in Figure 102 deserves additional explanation. Clearly network B breaks our basic requirement for corresponding linkage of XorNodes. The first XorNode has 3 downstream links while there are no other XorNodes in the network with 3 upstream links. Intuitively, the reason that this network is illegal is an ambiguity for PEX values. What should the PEX values be for the middle three activities? Starting from the upstream activity, we would assign a PEX of 1 to it and to the first XorNode. Following the PEX propagation rules for XorNodes we would then assign each of the middle three activities a PEX value of 0.33. Consider, however, starting from

Node	Start Time Domain	End Time Domain
X_1	[0 90]	[0 90]
A_6	[0 70]	[25 95]
A_1	[0 80]	[15 95]
A_3	[0 90]	[5 95]
X_2	[5 95]	[5 95]
A_5	[5 90]	[15 100]
A_7	[5 95]	[10 100]
X_3	[10 100]	[10 100]

Table 7. **The Time Windows for the Activities in Figure 101.**

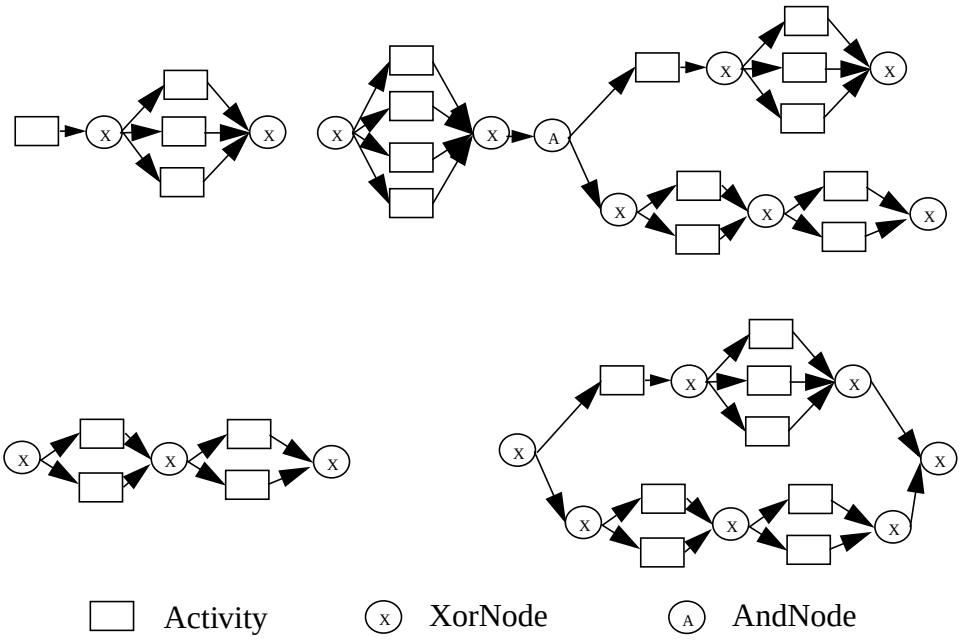


Figure 103. Examples of Legal Temporal Networks.

the last node and doing the PEX propagation upstream. The final XorNode will have a PEX of 1. The two nodes directly connected to the final XorNode (*i.e.,* the penultimate XorNode and the bottom activity) would then be assigned a PEX of 0.5 each and then propagating from the penultimate XorNode, the top two middle activities would have a PEX of 0.25 each. The ambiguity over the PEX values is the reason that network B is illegal and therefore the reason for the "matching links" requirement for XorNodes in a legal temporal network.

7.4 Propagating PEX

The basic PEX propagation algorithm is achieved through the PEX propagation behaviour at each temporal node. As described above, at an AndNode (or Activity), *A*, all the non-XorNodes directly linked to *A* must have the same PEX value as *A*. Therefore, when PEX propagation enters an AndNode, the local PEX value can be modified and the new local value is propagated to all neighboring nodes. For XorNodes, the sum of the PEX values for all nodes directly connected upstream must be equal to the sum for all nodes directly connected downstream which in turn must be equal to the PEX value of the XorNode.

To present the PEX propagation algorithms, we distinguish two cases. The first, initial propagation, begins with a temporal network where all PEX values are unassigned, and consistently assigns all the PEX variables in the network. The second case is where consistent PEX values exist for all activities and then some change is made (*i.e.,* a commitment is asserted). A change may modify the PEX values, and so we need to re-propagate and recreate the consistent network.

7.4.1 Initial Propagation

An unspoken assumption through our discussion above is that any temporal node that either has no upstream links or no downstream links must have a PEX value of 1. Clearly, given our temporal networks, if a node has either no upstream or no downstream links it cannot be subject to a XorNode and therefore must be present in the final solution.

```
1:  for each temporal node with no upstream links
2:     assign PEX = 1
3:
4:  create downstream topological sort of temporal nodes
5:  for each temporal node with unassigned PEX in sort order
6:     if the node is an AndNode OR Activity
7:        upstreamNode = any directly connected, active upstream node
8:        set PEX = downstreamPEXValue(upstreamNode)
9:     if the node is an XorNode
10:       set PEX = 0
11:       for upstreamNode = all directly connected, active upstream nodes
12:          set PEX += downstreamPEXValue(upstreamNode)
13:
14:
15: procedure downstreamPEXValue(Node node)
16:
17:    if the node is an AndNode OR Activity
18:       return PEX
19:    if the node is an XorNode
20:       return (PEX / number active downstream links)
```

Figure 104. Pseudocode for the Initial PEX Propagation Algorithm.

As indicated in the previous section, a requirement of a legal temporal network is that, regardless of starting with all nodes with no upstream links or all nodes with no downstream links, the PEX values assigned to each node in the network will be uniquely determined. Therefore, we can simply choose one of these options. We have arbitrarily chosen to perform the initial propagation by assigning all nodes with no upstream links to a PEX of 1 and then propagating downstream.

Pseudo-code for the initial propagation algorithm is given in Figure 104. Lines 7, 11, and 20 make use of active nodes and active links. An *active link* is a binary temporal constraint such that the PEX value of each of its nodes is not assigned to 0. An *active node* is a node connected via an active link.

Recall that a topological sort (line 4) in an acyclic, directed network is an ordering of the nodes such that if there is a path from node Y to node X in the graph then Y < X in the sort order [Roberts, 1984]. In our case, the topological sort establishes that if Y is upstream of X in the temporal graph, then Y < X in the sort order. In the loop beginning at line 5, therefore, we are guaranteed that all upstream nodes already have their PEX values assigned.

The complexity of the initial propagation given n nodes and m temporal constraints is $O(\max(m, n))$. The first loop (line 1) may visit each activity $O(n)$, the topological sort (implemented with a depth-first search) is $O(\max(m, n))$ as all nodes and constraints must be visited, and the final loop (line 5) is also $O(\max(m, n))$.

There is the possibility that the original temporal network will contain unary PEX constraints that assign or limit the PEX value of an internal node. Rather than incorporating the unary PEX constraints into the initial propagation, we perform the algorithm in Figure 104 and then transition to the incremental propagation for each of the unary PEX constraints. It is likely that the complexity and/or average run time of the initial propagation algorithm can be improved by taking into account unary PEX constraints during the initial propagation. We have not investigated such an initial propagation algorithm.

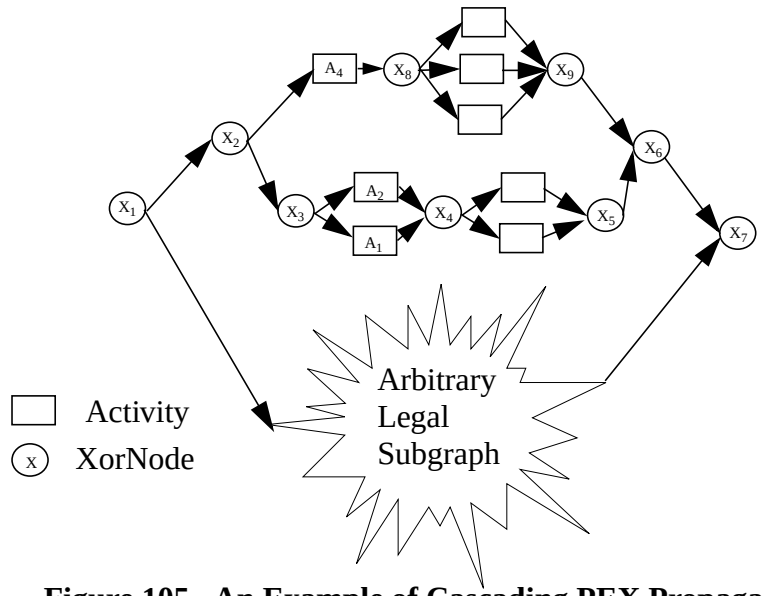


Figure 105. An Example of Cascading PEX Propagation.

7.4.2 Incremental Propagation

Incremental PEX propagation starts with a consistent network with respect to the PEX values and some change to a PEX value in the network. The change is represented by the addition of a new unary PEX constraint. Recall that we limit the new PEX constraints to be assignments of PEX variables to either 0 or 1.

7.4.2.1 Example of Incremental PEX Propagation

Before presenting the algorithm, it is useful to examine an example, especially to understand the requirement for "cascading" PEX propagation. Figure 105 presents a temporal network. Assuming X_1 and X_7 both have an initial PEX value of 1 and that, after initial propagation, we add the unary PEX constraint stating that the PEX value of activity A_1 is 1, Table 8 shows the PEX values of a subset of nodes from Figure 105 both before and after the constraint is added to the graph and incremental PEX propagation is performed.

To make these reassignments, a number of cascading PEX propagations must be performed. Beginning at A_1, we assign its PEX value to 1, and then search upstream and downstream for a pair of XorNodes. The graph in Figure 105 shows that the XorNodes can be nested: the alternative represented by X_3 and X_4 is nested within the alternative represented by X_2 and X_6 which in turn is nested within the alternative represented by X_1 and X_7. The initial step of the PEX propagation is to identify the inner-most XorNodes relevant to the node whose PEX is being assigned. Having identified these nodes (X_3 and X_4, in this case) we propagate from A_1 upstream to X_3 and downstream to X_4. This propagation assigns a PEX value of 1 to both X_3 and X_4. We propagate this PEX downstream along all paths from X_3 to X_4 following the same rules as with initial propagation. In this case the propagation sets the PEX of A_2 to 0.

This is the initial application of incremental PEX propagation. If, in the initial state, the PEX values of X_3 and X_4 had already been 1, we would be finished PEX propagation at this point. However, the propagation has reset the PEX values of X_3 and X_4, and therefore we need to continue the propagation. We call this continuation a *cascade* of PEX propagation. The cascade is, however, relatively straightforward as the only difference from the first round of incremental propaga-

tion is the nodes that form the starting point. Rather than starting with A_1, we now start with the inner-most XorNodes from the previous iteration. In our example, we begin with X_3 and identify the inner-most XorNode upstream which is X_2. Also starting from X_4, we identify the inner-most downstream XorNode which is X_6.[4] As with the initial iteration, we now reset the PEX values of the identified XorNodes (both to 1 in this case) and PEX propagate downstream from X_2 to X_6. The second iteration of PEX propagation will correctly reassign the PEX value of X_5 (to 1), the PEX values of the activities between X_4 and X_5 (both to 0.5), and the PEX values of all nodes along any path from X_2 to X_9 (all to 0). One more cascade of PEX propagation is necessary to arrive at our final PEX values. The final cascade sets the PEX values of all nodes in the "Arbitrary Legal Subgraph" to 0.

7.4.2.2 The Details

Figure 106 and Figure 107 present the pseudo-code for the incremental PEX propagation algorithm. As introduced above, the main components of this procedure are the identification of the inner-most enclosing XorNodes followed by the reassignment of PEX within the subgraph between the identified XorNodes.

Identifying the Enclosing XorNodes. The technique for identifying the upstream enclosing XorNode is analogous to that used in identifying the downstream XorNode. The only difference, of course, is a reversal of direction of the constraints. Therefore, we will present the downstream identification routine. The basic purpose of the routine is to find the closest downstream XorNode that encloses the starting node (*e.g.,* `downFrom` in Figure 106). There are a number characteristics of the downstream XorNode and a legal temporal graph that are used in this identification.

Node	Initial PEX Values	PEX Values After Commitment
X_1	1.0	1.0
X_2	0.5	1.0
A_4	0.25	0
X_8	0.25	0
X_9	0.25	0
X_3	0.25	1.0
A_1	0.125	1.0
A_2	0.125	0
X_4	0.25	1.0
X_5	0.25	1.0
X_6	0.5	1.0
X_7	1.0	1.0

Table 8. **The PEX Values for a Subset of the Nodes in Figure 105.**

4. Note that the inner-most XorNode downstream of X_4 is not X_5 because the X_3-X_4 alternative is not nested within the X_4-X_5 alternative. We return to this slight intricacy below.

```
1:  procedure incrementalPEX(Node upFrom, Node downFrom)
2:     (upXor, upTopoSort) = findUpstreamXor(upFrom)
3:     (downXor, downTopoSort) = findDownstreamXor(downFrom)
4:
5:     propagatePEXUpstream(upFrom, upTopoSort)
6:     propagatePEXDpdownstream(downFrom, downTopoSort)
7:
8:     propagatePEXBetweenXors(upXor, downXor)
9:
10:    return upXor, downXor
```

Figure 106. High-level Pseudocode for the Main Procedure of the Incremental PEX Propagation Algorithm.

```
1:  procedure findDownstreamXor(Node node)
2:     nestCount = 0
3:     if (node is a XorNode with multiple active downstream links)
4:        nestCount = 1
5:
6:     downXor = recursiveFindDownstreamXor(node, topoSort)
7:     return downXor, topoSort
8:
9:
10:
11: procedure recursiveFindDownstreamXor(Node node, NodeList topoSort)
12:    if node has already been visited
13:       return NULL
14:    else if identifyDownstreamXor(node)
15:       topoSort.push(node)
16:       return node
17:    else
18:       for downNode = each downstream neighbor
19:          downXor = recursiveFindDownstream(downNode, topoSort)
20:
21:       topoSort.push(node)
22:       return downXor
23:
24:
25:
26: procedure identifyDownstreamXor(XorNode node)
27:    found = FALSE
28:    if (node has multiple active upstream links)
29:       if (nestCount == 0)
30:          found = TRUE
31:       else if (node has multiple active downstream links)
32:          nestCount--
33:    else if (node has multiple active downstream links)
34:       nestCount++
35:
36:    return found
```

Figure 107. Pseudocode for Identifying the Downstream XorNode during PEX Propagation.

1. The downstream XorNode must have multiple active upstream links. This requirement is to avoid identifying a XorNode that originally enclosed the alternative, but due to search commitments no longer represents an alternative. Imagine that given the graph in Figure 105, some commitment has assigned all the nodes between X_3 and X_5, inclusive, to PEX values of 0. Further imagine a subsequent commitment that sets the PEX value of activity A_4. The inner-most XorNodes in this state are X_1 and X_7. Because the alternative represented by X_2 and X_6 has already been determined, X_2 and X_6 are no longer enclosing XorNodes for A_4.
2. Any path downstream from a node must pass through its enclosing downstream XorNode (if it has one). This is due to the construction of a legal temporal network.
3. Any downstream path from a node may contain XorNodes that include further alternatives. In the second iteration of our example PEX propagation routine (when identifying the enclosing XorNode downstream of X_4 in Figure 105), the alternative represented by X_4 and X_5 is contained within the path from X_4 to X_6 and so to identify X_6 as the correct downstream XorNode, we must be able to deal with such a situation.

Using these three characteristics, the pseudo-code presented in Figure 107 identifies the downstream XorNode from the passed-in node. This code returns the downstream XorNode together with a topological sort of the nodes between the passed in node and the XorNode. The `findDownsteamXor` function does the initial call to `recursiveFindDownstreamXor` which implements a depth-first search until the downstream XorNode is identified. The `identifyDownstreamXor` procedure manages the counting of XorNodes in order to ensure the one identified is truly the inner-most enclosing XorNode.

Propagating PEX to the XorNodes. Once the enclosing pair of XorNodes has been identified, PEX propagation is done using the topological sorts created during the search for the XorNodes.[5] Given the downstream topological sort, we can simply follow the algorithm in Figure 104 to propagate between the node that has been assigned a PEX value and the downstream XorNode.

Regardless of the temporal network, when we reach the downstream XorNode we are guaranteed, by the definition of an enclosing XorNode, that the PEX value propagated to the XorNode will be equal to the value assigned to the original node by the new PEX constraint. If the value is 0, the PEX value at the XorNode remains unchanged. If the value is 1, the PEX value of the XorNode is set to 1 regardless of its previous value.[6]

Propagating PEX between the XorNodes. The final step, assuming there is no PEX cascade, is to re-propagate the values for the network between the two enclosing XorNodes. This, again, is simply done using the pseudo-code in Figure 104.

Identifying a Cascade. After an iteration of PEX propagation, it is necessary to decide if a cascade of PEX propagation should occur. The simple criterion to establish the necessity of a cascade is the change in the PEX value of the enclosing XorNodes. If the PEX value of the enclosing XorNodes is greater than it was before the propagation, it is necessary to perform a cascade of PEX propagation.

---

5. As the upstream propagation is symmetrical we will again, without loss of generality, discuss only the downstream propagation.
6. If its previous value was less than 1, a cascade of PEX propagation will later be triggered.

7.4.2.3 Complexity

At worst, PEX propagation will require $O(n)$ cascades where n is the number of temporal nodes in the graph, since, in the extreme case, there can be $O(n)$ nested alternatives. Each cascade incurs a worst-case complexity of $O(\max(n, m))$ by the same arguments used for initial propagation complexity. Therefore, the overall worst-case complexity is $O(\max(n^2, nm))$.

7.5 Temporal Propagation with PEX

There are three differences in temporal propagation when XorNodes are present in the network.

1. Propagation through a XorNode is different than propagation through an Activity since a XorNode represents a different temporal relationship.
2. As with normal temporal propagation, it is possible to derive a dead-end in the search by emptying the domain of a variable during temporal propagation. When there are nodes with PEX < 1 in the graph, such a state may not be a dead-end, but rather may indicate an implied PEX commitment.
3. Temporal propagation needs to be performed after PEX propagation.

7.5.1 Temporal Propagation through a XorNode

The temporal propagation through a XorNode is straightforward given the temporal semantics of a XorNode. A XorNode enforces the temporal relationship that it must start at the same time as or after the end time of at least one of its upstream neighbors, and it must end at the same time as or before the start time of at least one of its downstream neighbors. During downstream propagation therefore the XorNode sets the lower-bound on its start time domain based on the minimum earliest finish time of its upstream neighbors. This start time is then propagated further downstream. Similarly, during upstream propagation the upper-bound on the end time of a XorNode is set based on the maximum latest start time of its downstream neighbors and this value is further propagated upstream.

Another perspective from which to view the temporal relationship enforced by the XorNodes is to examine the pairs of XorNodes. A pair of XorNodes that correspond to an alternative enforce a minimum interval of time between themselves. This minimum interval corresponds to the length of the shortest path between them (where path length is computed as the sum of the minimum durations of the temporal nodes on the path).

7.5.2 Deriving Implied PEX Commitments from Temporal Propagation

In standard temporal propagation algorithms, if it is discovered that a variable's domain has been emptied, a dead-end is derived. When PEX variables are present, emptying a domain may not be a dead-end. Rather it may indicate that a particular PEX variable must have a value of 0.

When a domain is emptied in temporal propagation, the PEX value of the temporal node with the emptied domain is examined. If the PEX value is less than 1, the node is marked to indicate that the PEX value has been determined to be 0 and temporal propagation does not continue from that node. After temporal propagation, a separate propagation algorithm examines all the temporal nodes to determine if any have been marked to indicate the derivation of an implied PEX constraint. If such a temporal node is found, a new unary PEX constraint is asserted, and the usual PEX propagation and the temporal propagation that follows PEX propagation (see below) is done.

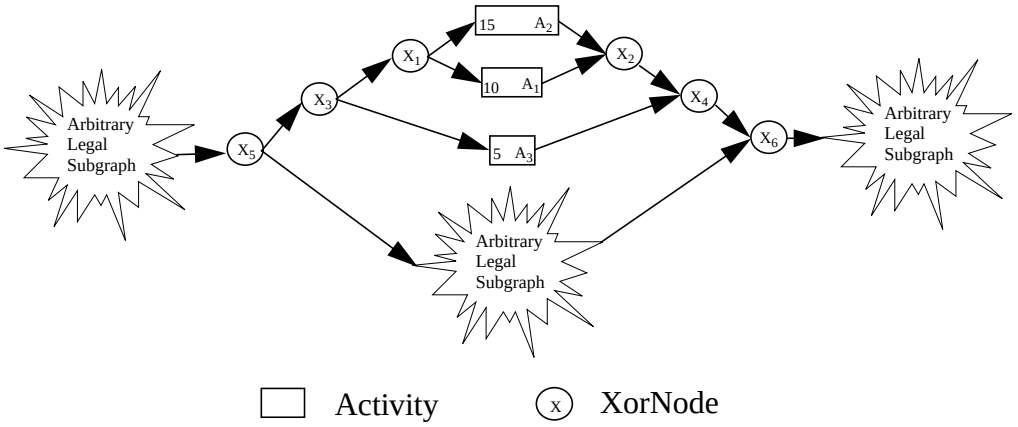


Figure 108. An Example of Cascading Temporal Propagation.

If the domain of a temporal node with PEX of 1 is emptied, a dead-end is derived as in the standard temporal propagation. To see why this is the case, imagine emptying the start time domain of an activity, *A*, with a PEX value of 1. By definition, it cannot have any enclosing XorNodes. Assume that the presence of some activity, *B*, with a PEX value of less than 1 caused the empty domain. The temporal propagation from *B* to *A* must occur through one of the XorNodes enclosing *B*. If that propagation empties *A*, then it must be the case that not only *B* but *all* its alternatives are inconsistent with *A*. Therefore, it is a true dead-end. Note that this reasoning holds even if activity *B* is nested inside a number of alternatives: the temporal propagation from *B* to *A* must pass through the outer-most XorNodes enclosing *B*. This XorNode, *X*, must have a PEX value of 1. Since all the alternatives of *X* are inconsistent with *A*, we have a true dead-end.

7.5.3 Temporal Propagation After PEX Propagation

After the PEX propagation described in Section 7.4, possibly including a cascade of PEX propagation, temporal propagation must be performed to re-establish a temporally consistent network.

Returning to our example in Figure 105, recall that a PEX commitment assigned activity A_1 a PEX value of 1. This resulted in three cascades of PEX propagation with the following three sets of enclosing XorNode pairs: X_3-X_4, X_2-X_6, and X_1-X_7.

Following PEX propagation, temporal propagation must be performed once for each PEX propagation cascade. Starting from the first cascade, temporal propagation must proceed upstream from the first upstream XorNode (*e.g.*, X_3) and downstream from the first downstream XorNode (X_4). It is then necessary to perform temporal propagation upstream from the second upstream XorNode (X_2), downstream from the second downstream XorNode (X_6) and finally from the third pair of XorNodes (X_1 and X_7). These cascades of temporal propagation must be done for each cascade of PEX propagation to ensure a temporally consistent network.

Figure 108 presents an example of a network where a cascade of temporal propagation is necessary to ensure consistency. Assume that a newly posted PEX constraint assigns the PEX variable of activity A_1 to 1. Due to the temporal relationship already enforced by the XorNodes X_1 and X_2, there is no need for propagation upstream from X_1 or downstream from X_2; these nodes already enforce an interval of at least 10 time units between them. However, the PEX propagation also assigns activity A_3 to a PEX value of 0. This means that the time gap between X_3 and X_4 must now be increased to at least 10 time units. While A_3 was still a possible activity, 5 time units was a consistent interval of separation between X_3 and X_4; however, now the gap must be at least 10

time units to incorporate A_1. So while it was not necessary to temporally propagate from the first pair of XorNodes, it is necessary to temporally propagate from the second pair and possibly also the third pair (X_5 and X_6). It is also possible to create temporal graphs where propagation is required from arbitrary pairs of the XorNodes in the PEX propagation cascade.

Note that while the temporal propagation occurs after all PEX cascades have been completed, it is necessary, during PEX propagation, to record the pairs of enclosing XorNodes for later use by temporal propagation. This is achieved with a queue of XorNode pairs populated during PEX propagation and used during temporal propagation.

7.5.4 A Note on Temporal Propagation Algorithms

These techniques for temporal propagation in the presence of PEX variables rely on non-PEX temporal propagation algorithms. The ODO shell has a variety of temporal propagation algorithms all of which have the same worst-case complexity, but vary in average-time performance. Rather than using the temporal propagation algorithm that appears relatively standard in the temporal network research [Cesta and Oddi, 1996; Dechter et al., 1991], we use an algorithm that performs upstream and downstream propagation separately and makes use of the GOR algorithm [Cherkassky et al., 1994] for the incremental construction of the depth-first search tree.

The PEX and temporal propagation algorithms depend on identification of the “direction” of temporal constraints: it must be known whether a constraint is an upstream or a downstream constraint. The standard temporal propagation algorithms in other scheduling systems do not appear to make the distinction between upstream and downstream constraints at a node, and furthermore, this distinction becomes suspect with more complex binary temporal constraints. For example, if A_2 must execute while A_1 is executing, is A_1 upstream or downstream of A_2? The question arises therefore about incorporating these propagation techniques into other systems.

For most of the PEX and temporal propagation presented above, the distinction between upstream and downstream constraints is not crucial to the correctness of the algorithm, but rather reduces the average run-time complexity. For example, to find the upstream and downstream XorNodes from a temporal node we follow any path upstream and then follow any path downstream. If we did not make the distinction between upstream and downstream constraints, the XorNodes could still be found; however, it might be the case that all the constraints at a node had to be followed in order to identify the enclosing XorNodes. Recall that in a legal network, we are guaranteed that any path of temporal constraints from a node must pass through one of its enclosing XorNodes. The algorithm therefore would simply find the first XorNode on each path and stop when the XorNodes found were different nodes.

The logical semantics of a XorNode, however, requires a distinction between the upstream and downstream constraints. In a final solution, a XorNode must enforce the condition that only one of its upstream neighbors exists (that is, has a PEX value of 1) and only one of its downstream neighbors exists. It seems, therefore, that to incorporate PEX into the standard temporal network algorithms, it will be necessary to add some notion of upstream and downstream constraints. However, it is only the XorNode that must make that distinction. Therefore, it is believed that the changes to the standard propagation techniques can be encapsulated within the behaviour of the XorNodes.

7.6 Incorporating PEX in Scheduling Heuristics

With the ability to represent and propagate the fact that some temporal nodes may not be present in a final schedule, it is necessary to extend the heuristic search techniques to allow direct reasoning about an activity's probability of existence.

7.6.1 Texture-based Heuristics

The obvious extension is to expand the type of commitments that a heuristic can make to include setting the PEX variable of an activity. This, indeed, is the basic extension to existing heuristics. Texture-based heuristics, as have been developed in this dissertation, are based on an analysis of the constraint graph to identify the critical points where a commitment should be made, followed by a commitment that attempts to decrease that criticality. PEX values represent potentially important search information and therefore should be embedded in the texture measurements to ensure that they are taken into account in the estimation of criticality at a search state.

The PEX variables affect texture-based heuristics in two ways. First, it is desirable to incorporate the PEX variables into the underlying texture measurement estimation technique. The criticality of an activity may be very different if it has a 0.125 probability of executing than if it has a 0.5 probability. Second, once textures have been calculated, the presence of PEX variables expands the types of heuristic commitments that can be made. Rather than sequencing the most critical pair of activities, perhaps setting the PEX variable of one of the activities to 0 is a better heuristic commitment.

7.6.1.1 Adding PEX to Texture Curves

The sole modification to texture measurements to incorporate PEX variables is to apply a vertical scaling factor to the individual activity demand. If the PEX value of activity A is 0.5, the individual demand at any time point t, $est_A \leq t < lft_A$, is half what the demand would be if the PEX value was equal to 1. This modification fits with our semantic interpretation of individual demand to be the probabilistic demand of an activity at a time point. Because we interpret a PEX value as an activity's probability of existence, an activity that has only a 50% likelihood of existing has half the probabilistic demand of an identical activity that will definitely exist. This is the same way that the probability of existence of an activity is taken into account in the KBLPS scheduler [Fox, 1999].

Recall that the individual demand (Section 4.3.2.1), $ID(A,R,t)$, is (probabilistically) the amount of resource R, required by activity A, at time t. It was calculated in Equation (14) (Chapter 4, p. 56). The modification to this calculation that incorporates PEX, $ID_{PEX}(A,R,t)$ simply multiplies by A_{PEX}, the PEX value of activity A:

$$ID_{PEX}(A, R, t) = A_{PEX} \times ID(A, R, t) \tag{36}$$

Clearly, when the PEX value is equal to 1, ID_{PEX} is equal to ID.

This modification of texture measurements represents a generalization of the work presented in Chapter 4 and Chapter 5. Two factors achieve this generalization:

1. Recall that the four variations on texture measurements (SumHeight, VarHeight, JointHeight, and TriangleHeight) calculate the individual demand curve in exactly the same way. The difference among the variations concerns the method of aggregation of the individual curves. The change to the texture measurement calculation resulting from the incorporation of PEX is limited to the calculation of the individual demand curve of each activity. Therefore, reasoning about PEX can be done with each of the texture measurement variations.
2. In the standard job shop problem all activities have a PEX value of 1. In the modified calculation of individual activity demand, the demand of an activity with a PEX value of 1 is identical to the individual demand curve presented in Chapter 4. Therefore, when applied to job shop scheduling problem, the texture measurements (and heuristic commitments) that incorporate PEX will be identical to the texture measurements that do not incorporate PEX.

7.6.1.2 New Heuristic Commitments

Independent of the texture measurement used, the basic texture algorithm remains the same (with ID_{PEX} replacing ID) up to the point where a commitment is chosen. That is, the individual demands are calculated and aggregated, and the resource and time point with highest criticality is identified. In non-PEX texture-based heuristics, the two most critical activities on the critical resource that were not already sequenced were then identified. This identification was done on the basis of the highest individual demand for the resource at the critical time point (Section 4.3.2.1). A commitment sequencing these two activities was then created and asserted (Section 4.3.2.2).

To incorporate PEX values, we follow the same intuition as in our non-PEX, texture-based heuristics: make a commitment that will tend to most reduce the criticality of the most critical resource. Given, the most critical resource, R^*, and the most critical time point, t^*, we identify the activity on R^* with the highest individual demand at t^*. If the most critical activity has a PEX value of 1, it must execute on R^* and so the heuristic commitment is the same as in our non-PEX heuristics. The critical activity is sequenced with the next most critical activity that also has a PEX of 1.

Consider the situation where the most critical activity has a PEX value of less than 1. The most critical activity on R^*, therefore, has alternatives and the greatest reduction in the criticality of R^* at t^* will result in choosing to execute one of those alternatives rather than the critical activity. Because we follow a least commitment approach, rather than explicitly choosing one of those alternatives, we simply choose *not* to execute the critical activity by posting a heuristic commitment setting its PEX value to 0.

Note that such a commitment will increase the PEX value of the alternatives to our critical activity. However, those alternatives must be executing on resources of equal or less criticality than R^*. (Recall that R^* and t^* are the most critical resource and time point at this problem state).

More formally, then, the procedure for generation of heuristic commitment techniques in the presence of PEX values is as follows. We identify three activities:

1. The activity, A, with PEX value, A_{PEX}, $0 < A_{PEX} < 1$, and with the highest individual demand for the critical resource, R^*, at the critical time point, t^*, of all activities with PEX values between 0 and 1.
2. The pair of activities, B and C, that are not currently sequenced, with PEX values $B_{PEX} = 1$ and $C_{PEX} = 1$, and with the highest individual demand for the critical resource at the critical time point of all activities with a PEX value of 1. Without loss of generality, assume that the individual demand of activity B at the critical time point is greater than or equal to the individual demand of C at the critical time point.

```
1:  if (ID_PEX(A, R*, t*) >= ID_PEX(B, R*, t*)) || (B or C do not exist))
2:     commitment = assign A_PEX to 0
3:  else if (ID_PEX(B, R*, t*) > ID_PEX(A, R*, t*)) || (A does not exist))
4:     commitment = heuristically-sequence(B, C)
```

Figure 109. Pseudocode for Determining Heuristic Commitment.

We then determine the heuristic commitment as displayed in Figure 109. The heuristic commitment is found by comparing the individual demand for *A* at *t** with that of *B*. If the individual demand of *A* is higher, then it is the most critical activity. Since *A* has a possibility of not existing, the heuristic decision to reduce criticality is to remove it from the schedule by setting its PEX value to 0. On the other hand if *B* has the higher individual demand, then it is necessary to sequence *B* and *C* to attempt to reduce criticality. We use the same sequencing heuristics presented in Chapter 4 (Section 4.3.2.2).

If the heuristic PEX commitment is retracted via a complete retraction technique, we soundly post its opposite, setting the PEX value of *A* to 1. Similarly, if the sequencing commitment is retracted, we post the opposite sequence.

The texture-based heuristics investigated in this chapter are SumHeightPEX and VarHeightPEX, modifications, respectively of the SumHeight and VarHeight heuristics to take into account the PEX variables.

7.6.2 Other Heuristics

To form a basis of comparison for the quality of the texture-based heuristics for PEX, we can modify other heuristics to incorporate the PEX variables. Here we present the modifications to CBASlack and LJRand heuristic.

7.6.2.1 CBASlackPEX

Recall that the CBASlack heuristic (Section 2.3.4) identifies the pair of activities on the same resource that are not sequenced and that have the minimum biased-slack measurement. These two activities are sequenced to preserve the maximum amount of slack. To adapt the CBASlack heuristic to activities with PEX values, we follow the intuition that a commitment should preserve the maximum amount of slack.

The adaptation first specifies that the biased-slack measurement is calculated only for activity pairs such that both activities have a PEX value greater than 0. The following three conditions then apply to the activity pair:

1. If both activities have a PEX value of 1, follow the CBASlack heuristic and post the sequencing constraint that preserves the most slack.
2. If one activity, *A*, has a PEX value of 1 and the other, *B*, has a PEX value of less than 1, the greatest amount of slack will be preserved by setting the PEX value of *B* to 0. *B* will be completely removed from competition with *A* thereby maximizing the resulting slack between the critical activity pair.
3. If both activities have a PEX value of less than 1, the greatest amount of slack will be preserved by setting the PEX value of the activity with the longest duration to 0. Again, by removing the activity with maximum duration, we maximize the resulting slack between the critical activity pair.

If any of these commitments is retracted, we can post its opposite (the other sequence for case 1 or setting the PEX value to 1 in cases 2 and 3) to guarantee a complete search.

7.6.2.2 LJRandPEX

Recall that the LJRand heuristic (Section 2.3.5), finds the smallest earliest finish time of all the unscheduled activities and then identifies the set of activities which are able to start before this time. One of the activities in this set is selected randomly and scheduled at its earliest start time. When backtracking, the alternative commitment is to update the earliest start time of the activity to the minimum earliest finish time of all other activities on that resource.

Our modification of LJRand to incorporate PEX variables, LJRandPEX, performs the following steps:

1. Find the smallest earliest finish time of all activities with PEX greater than 0.
2. Identify the set of activities with PEX values greater than 0 that can start before the minimum earliest finish time.
3. Randomly select an activity, *A*, from this set.
4. Assign *A* to start at its earliest start time and assign its PEX variable, A_{PEX}, to 1.

The alternative commitment, should backtracking undo the commitment on activity *A*, is to update the earliest start time of *A* to the minimum earliest finish time of all other activities with PEX > 0, on the same resource as *A*. Note that the alternative does not contain a PEX commitment, which means that subsequent heuristic and implied commitments can still assign A_{PEX} to either 1 or 0. This ensures completeness of the search. With chronological backtracking, if the commitment on activity *A* is undone, it has been derived that *A* cannot start at its earliest start time in any schedule. The alternative then is that *A* starts later or that *A* does not execute at all. For a complete search we need to preserve these two alternatives when backtracking.

7.6.3 The Information Content of Heuristic Commitment Techniques

One of the key differences among the heuristics that incorporate PEX is the extent of that incorporation. The texture-based heuristics use the PEX value to calculate the underlying individual demand of an activity and so it has a deep impact on the heuristic commitments that are made. LJRandPEX and CBASlackPEX in contrast only use the PEX variable as a three-value variable: 0, 1, or neither-0-nor-1. In their heuristic commitment process, these heuristics do not use the fact that one activity may have a PEX value of 0.125 while another a PEX value of 0.5. We expect that, because the texture-based heuristics take into account the information represented by the value of the PEX variable, that they will outperform heuristics that do not use that information.

While it is hard to see how the PEX value could be more deeply incorporated into the LJRand heuristic, in case 3 of the CBASlackPEX heuristic we may choose to set to 0 the PEX value of the activity that maximizes the product of the duration and PEX value. We do not incorporate the PEX value more deeply into the CBASlackPEX heuristic, in this dissertation, in order to explicitly evaluate the usefulness of maintaining PEX values as described. If CBASlackPEX performs as well as (or better) than the texture-based heuristics, we will be able to simplify the PEX representation to the three values required by CBASlackPEX. It may be that much of the propagation infrastructure presented in this chapter can then be discarded without reducing the quality of the heuristic commitment technique.

7.7 Incorporating PEX in Propagators

The power of propagators is well-recognized in the constraint-directed scheduling literature. It appears likely, therefore, that the use of propagators in problems with PEX variables will lead to improved search performance. One difficultly in the application of existing propagation techniques to problems with PEX variables, is that propagators derive sound commitments based on the temporal relations among sets of activities that *must* execute. To maintain the soundness of a propagator's commitments, we must take into account the fact that those activities with a PEX value of less than 1 will not necessarily execute. In this section we examine each of the propagators used for job shop scheduling in Chapter 4 and Chapter 5 to determine if they can be extended to activities with a PEX value of less than 1.

7.7.1 Constraint Based Analysis

Constraint-Based Analysis (CBA) (Section 2.4.1) examines pairs of activities to determine if a sequence between them is implied by their current time windows. If such a sequence is found, a new precedence constraint is posted.

Assume that CBA has examined activities *A* and *B*, and determined that the sequence *A* before *B* is implied by the activity time windows. Clearly, if the activities examined both have a PEX value of 1, the CBA commitment can be asserted. If one of the activities, say *A*, has a PEX value of less than 1, however, posting a new precedence constraint will have two effects.

1. The temporal network will now be illegal. There will now be a path from *A* that does not include its enclosing XorNodes.
2. Temporal propagation may remove some possible start times from the domain of activity *B*. It may be that later in the search it is decided that *A* will not execute. In such a case, the possible start times removed from *B* as a result of the CBA implied commitment need to be re-inserted into the domain of *B*.

While it may be possible to get around both of these problems, we see no clear way forward. As a result, while we will continue to use CBA on all activities with a PEX value of 1, we do not include the other activities in our CBA propagation.

7.7.2 Edge-finding

Two types of edge-finding (exclusion (Section 2.4.2) and not-first/not-last (Section 2.4.3)) have been used in this dissertation. Recall that the reasoning in edge-finding is based on examining the set of activities on a resource and deriving new temporal constraints that enforce new upper and lower bounds on the end and start times of activities.

Clearly, edge-finding can be used with activities with a PEX value of 1. Imagine the situation, however, where all activities on a resource but one, *A*, have a PEX value of 1.

1. If edge-finding derives a dead-end, we can soundly infer that activity *A* cannot execute and therefore set A_{PEX} to 0.
2. If edge-finding derives new unary temporal constraints on *A*, they can be soundly asserted. Clearly if *A* is to execute, it must be consistent with the rest of the activities that must execute on the same resource. Therefore, any unary temporal constraints on *A* are sound: they must be true if *A* is to execute and if *A* does not execute they do not affect any other activities. (Note

```
1:  procedure PEXEdgeFinding
2:     list-of-commitments = EdgeFinding
3:     if list-of-commitments is not empty
4:        return list-of-commitments
5:
6:     for each activity A such that 0 < A_PEX < 1
7:        temporarily set A_PEX = 1
8:        tmp-list = EdgeFinding
9:        for each commitment in tmp-list
10:          if dead-end
11:             list-of-commitments.insert(set A_PEX to 0)
12:          else if new commitment on activity A
13:             list-of-commitments.insert(commitment)
14:          else
15:             discard commitment
16:       reset A_PEX
17:
18:    return list-of-commitments
```

Figure 110. Pseudocode for PEX-Edge-Finding.

that the temporal propagation via XorNodes guarantees that a unary temporal constraint on an activity with a PEX value less than 1 will not over-constrain the temporal network).

3. If edge-finding derives any unary temporal constraints on activities other than *A*, they must be discarded. Since it is possible for *A* not to execute, and since the temporal constraints are derived on the assumption that *A* would execute, we cannot soundly constrain the activities that have PEX values of 1.

This reasoning leads to the PEX-edge-finding algorithms presented in Figure 110. Note that this function uses the usual edge-finding algorithms as sub-routines (line 2). Given that the standard edge-finding worst-case complexity is $O(n^2)$, where n is the number of activities on a resource, the PEX-edge-finding worst-case complexity is $O(n^3)$. It is possible that this time-complexity can be improved by clever implementation. It is certainly the case that the average time performance can be improved by specializing the code for PEX-edge-finding rather than simply using the existing edge-finding code as a sub-routine. These optimizations remain for future work.

7.8 Empirical Evaluation

The empirical evaluation of the PEX techniques, in this dissertation, focuses on problems with alternative process plans, and problems with both alternative process plans and alternative resources. The PEX techniques presented are applicable, without extension, to both types of problems as an activity with alternative resources is simply treated as a nested alternative within a process plan.

7.8.1 Experimental Design

The primary purpose of the experimental evaluation is to determine the efficacy of using PEX values as part of the information underlying heuristic commitments. As noted above, among the heuristic commitment techniques, only SumHeightPEX and VarHeightPEX use the actual value of the PEX variable in each search state to inform its decision making. An important component of the empirical evaluation is to determine if using the extra information represented by PEX values

results in better heuristic commitments and overall search performance. A second purpose is the evaluation of the PEX edge-finding techniques. Given the relatively high time-complexity of the propagators, we want to evaluate their efficacy in terms of overall search performance.

Two of the obvious interesting parameters for alternative process plan problems is the number of alternatives in each process plan and the size of the overall scheduling problems. In Experiment 1 we look at the former independent variable while examining the latter in Experiment 2. Both experiments use a fully crossed design with makespan factor as the second independent variable.

The overall pattern we expect in Experiment 1 is that as the number of alternatives increases, the requirement for direct reasoning about the alternatives will become more important to good algorithmic performance. For Experiment 2, we focus on a more global factor of difficulty: the problem size. Our primary interest is to see how the algorithms with higher complexity components (such as PEX-edge-finding) are able to scale in terms of problem solving ability in relation to the lower complexity algorithms.

In our final experiment, Experiment 3, we incorporate alternative resources into the alternative process plan problem sets from Experiment 1 (see Section 7.10.1.1 for a detailed description of this incorporation). We allow activities to have up to six alternative resources. The primary impact of the addition of alternative resources is to greatly increase the range of PEX values across the activities in the problem. As with the increase in the number of alternatives per process plan, we expect the increase in the range of PEX values to favour the heuristics that make detailed use of them in their heuristic commitment techniques.

7.8.2 Instantiations of the ODO Framework

The eight algorithms used in the experiments are summarized in Table 9.

The general model for the experiments in this chapter matches the model used throughout the dissertation. Each algorithm is run until it finds a solution or until a 20 minute CPU time limit has been reached in which case failure is reported. The machine for all experiments is a Sun Ultra-Sparc-IIi, 270 Mhz, 128 M memory, running SunOS 5.6.

7.8.3 Statistical Analysis

As we have done throughout this dissertation, we measure statistical significance with the bootstrap paired-t test [Cohen, 1995] with $p \leq 0.0001$ (unless otherwise noted).

Our interest in these experiments is the comparison of the four heuristic commitment techniques (SumHeightPEX, VarHeightPEX, CBASlackPEX, and LJRandPEX) and an evaluation of the efficacy of PEX-edge-finding. The statistical analysis therefore compares the four heuristics with each other in two conditions: with and without PEX-edge-finding. To address the question of the usefulness of PEX-edge-finding, we also compare its use in four conditions corresponding to each of the heuristics. A summary of these groups is shown in Table 10.

Algorithm	Heuristic Commitment Technique	Propagators	Retraction Technique
SumHeightPropPEX	SumHeightPEX	All[a]	Chronological Backtracking
SumHeightPEX	SumHeightPEX	Non-PEX Propagators[b]	Chronological Backtracking
VarHeightPropPEX	VarHeightPEX	All	Chronological Backtracking
VarHeightPEX	VarHeightPEX	Non-PEX Propagators	Chronological Backtracking
CBASlackPropPEX	CBASlackPEX	All	Chronological Backtracking
CBASlackPEX	CBASlackPEX	Non-PEX Propagators	Chronological Backtracking
LJRandPropPEX	LJRandPEX	All	Chronological Backtracking
LJRandPEX	LJRandPEX	Non-PEX Propagators	Chronological Backtracking

a. Temporal propagation, PEX edge-finding exclusion, PEX edge-finding not-first/not-last, and CBA.

b. Temporal propagation, edge-finding exclusion, edge-finding not-first/not-last, and CBA.

Table 9. **The Eight Algorithms Used in the Alternative Process Plan Experiments.**

7.9 Alternative Process Plans

7.9.1 Experiment 1: Scaling with the Number of Alternatives

7.9.1.1 Problems

We expect that the number of alternatives in each process plan will have an impact on the search performance. To examine algorithm performance under varying numbers of alternatives, we constructed four problem sets with varying maximum numbers of alternatives in each process plan. To illustrate this construction we use a problem set with a maximum of three alternatives.

Each problem begins with an underlying job shop problem. All problems in this experiment have 10 activities per process plan and 10 resources. For a problem with a maximum of three alternatives per process plan, we generate 30 jobs of 10 activities each. These 30 jobs are then transformed into 10 process plans with alternatives. For each job we randomly choose the number of alternatives, *k*, it will have uniformly from the interval [0, 3]. We then combine randomly chosen portions of the next *k* jobs with our original job to produce a single process plan with *k* alternatives. The combination process randomly chooses the to-be-combined portion and the location in the original process plan where the alternative is to be inserted. The only requirements are that each path between a pair of XorNodes representing an alternative must have the same number of

Comparison Group	Purpose
SumHeightPropPEX, VarHeightPropPEX, CBASlackPropPEX, LJRandPropPEX	Compare heuristics when used with PEX-edge-finding.
SumHeightPEX, VarHeightPEX, CBASlackPEX, LJRandPEX	Compare heuristics when used without PEX-edge-finding.
SumHeightPropPEX, SumHeightPEX	Evaluate PEX-edge-finding with SumHeightPEX heuristic.
VarHeightPropPEX, VarHeightPEX	Evaluate PEX-edge-finding with VarHeightPEX heuristic.
CBASlackPropPEX, CBASlackPEX	Evaluate PEX-edge-finding with CBASlackPEX heuristic.
LJRandPropPEX, LJRandPEX	Evaluate PEX-edge-finding with LJRandPEX heuristic.

Table 10. **The Groups of Algorithms Used in the Statistical Tests.**

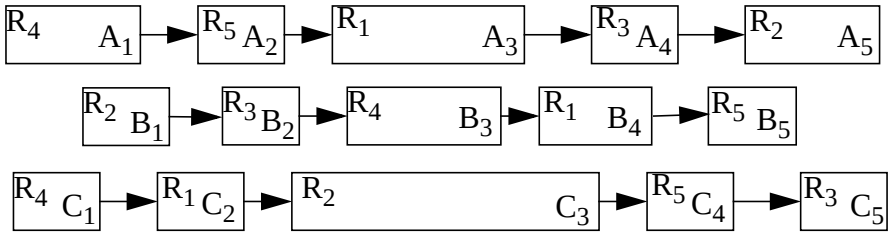
activities and that number must be greater than 1.The latter requirement avoids problem structures that are better categorized as alternative resources (see Section 7.10 and Section 7.11.4) while the former requirement avoids situations where an entire ten-activity process plan can have a two-activity alternative. Such a situation would lead to scheduling problems with essentially a single easy decision to make (choose the shorter process plan). We prefer to generate harder problems and so avoid such situations. Figure 111 illustrates the combination of three process plans into a single process plan with two alternatives.

Sets of problems with a maximum of one, three, five, and seven alternatives per process plan were generated. Each set contains 20 problems. For each problem the job lower bound was calculated taking into account the alternatives. This calculation finds the shortest path (where path length is determined by the sum of the durations of activities on a path) in each job. We then use a makespan factor spanning 1.0 to 1.5 (in steps of 0.1) to generate a total of six problem sets of 20 problems each for each number of alternatives.

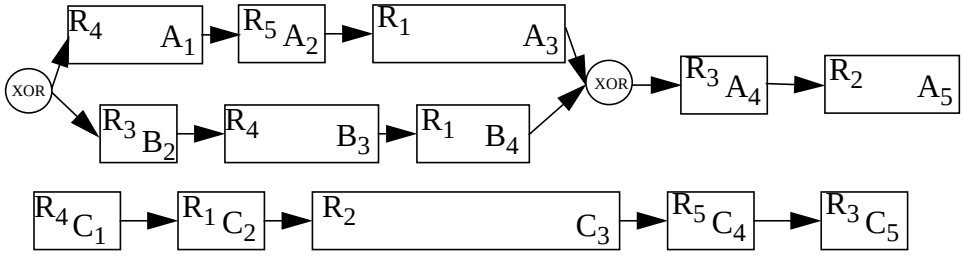
One of the results of this method of problem generation is that a solution to a problem will have exactly the same number of executing activities as the underlying job shop problem. Regardless of the number of alternatives, the final solutions for the problems in this experiment have 100 executing activities. Prior to scheduling, however, each problem has a different number of activities depending on the randomly chosen number and size of alternatives. Table 11 shows the characteristics of the problem sets in terms of the number of activities in the problem definition.

7.9.1.2 Results

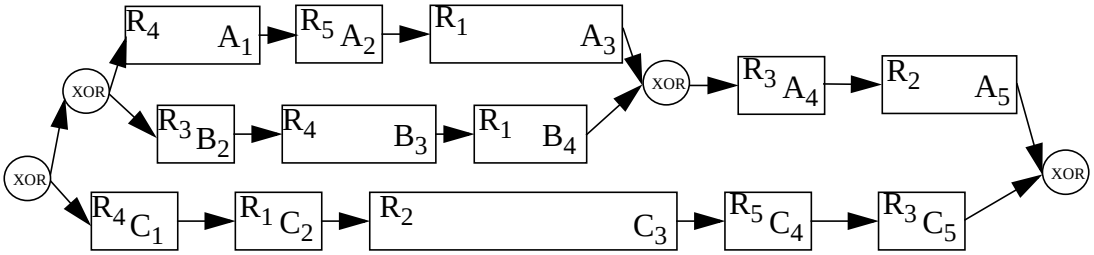
Complete results of Experiment 1 can be found in Section D.1 of Appendix D. In particular the results for each problem size can be found in the following sections of Appendix D: Overall – Section D.1.1, One Alternative – Section D.1.2, Three Alternatives – Section D.1.3, Five Alternatives – Section D.1.4, Seven Alternatives – Section D.1.5.



Original Process Plans



Process Plans After First Combination (B_1 and B_5 are discarded)



Final Process Plans (After Second Combination)

Figure 111. Generating a Single Process Plan with Two Alternatives.

The proportion of the problems in each set for which each algorithm times-out is shown in Figure 112. Slightly obscured by the graph is the result that SumHeightPropPEX and VarHeight-PropPEX do not time-out on any problems across all problem sets, and that there is no difference between CBASlackPropPEX and CBASlackPEX. Statistical analysis indicates that regardless of the use of PEX-edge-finding, the algorithms using VarHeightPEX, SumHeightPEX, and CBASlackPEX each time-out on significantly fewer problems than the corresponding algorithm

Problem Set (Maximum Number of Alternatives)	**Number of Activities Per Problem**		
	Minimum	**Mean**	**Maximum**
1	116	131.6	156
3	142	171.8	200
5	172	204.7	251
7	167	224.2	280

Table 11. **The Characteristics of the Problems in Experiment 1.**

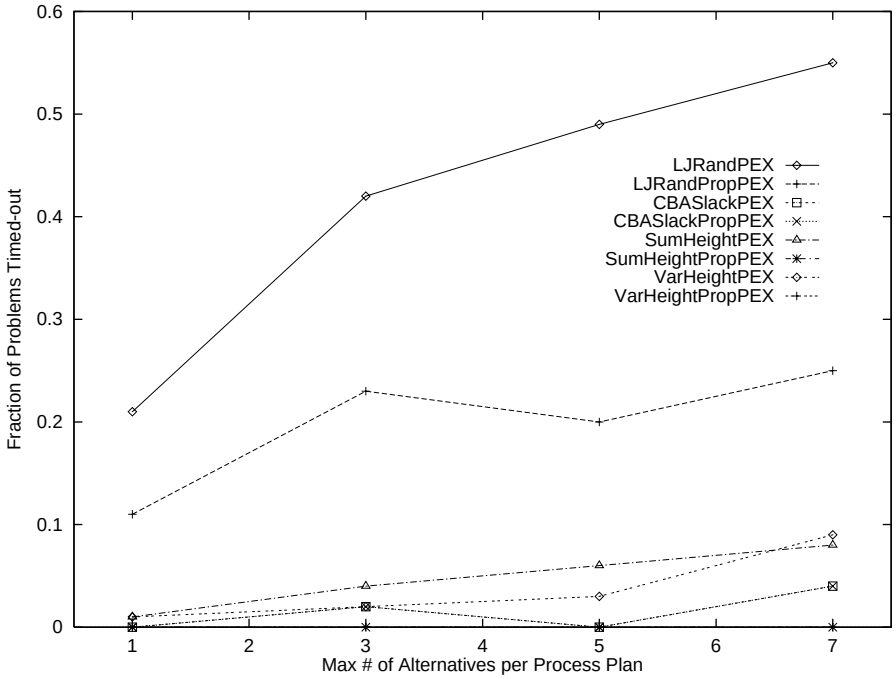


Figure 112. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out.

using LJRandPEX. With PEX-edge-finding there are no significant differences among SumHeightPropPEX, VarHeightPropPEX, and CBASlackPropPEX, while without PEX-edge-finding the only significant difference not involving LJRandPEX is that CBASlackPEX times-out on significantly fewer problems than SumHeightPEX ($p \leq 0.0005$). In terms of the usefulness of PEX-edge-finding, SumHeightPropPEX, VarHeightPropPEX, and LJRandPropPEX time-out on significantly fewer problems than SumHeightPEX, VarHeightPEX, and LJRandPEX respectively. There is no significant difference in performance between CBASlackPEX and CBASlackPropPEX.

Figure 112 indicates that the greatest differences among the algorithms are on the problem set with seven alternatives. The proportion of problems timed-out for each makespan factor of that problem set is shown in Figure 113. Statistical analysis indicates that, independent of the use of PEX-edge-finding, LJRandPEX times-out on significantly more problems than each of the other heuristics and there are no significant differences among the three other algorithms. In terms of the propagation, SumHeightPropPEX outperforms SumHeightPEX ($p \leq 0.005$), VarHeightPropPEX outperforms VarHeightPEX ($p \leq 0.0005$), LJRandPropPEX significantly outperforms LJRandPEX, and there is no difference between CBASlackPropPEX and CBASlackPEX.

Turning to mean CPU time, Figure 114 displays the mean CPU time across all problem sets while Figure 115 displays the same data for the seven-alternative problems. Overall, there is no significant difference between SumHeightPropPEX and VarHeightPropPEX. Both algorithms use significantly less mean CPU time than CBASlackPropPEX, which in turn uses significantly less mean CPU time than LJRandPropPEX. When PEX-edge-finding is not used, there is no difference among SumHeightPEX, VarHeightPEX, and CBASlackPEX, but all three are significantly better than LJRandPEX.

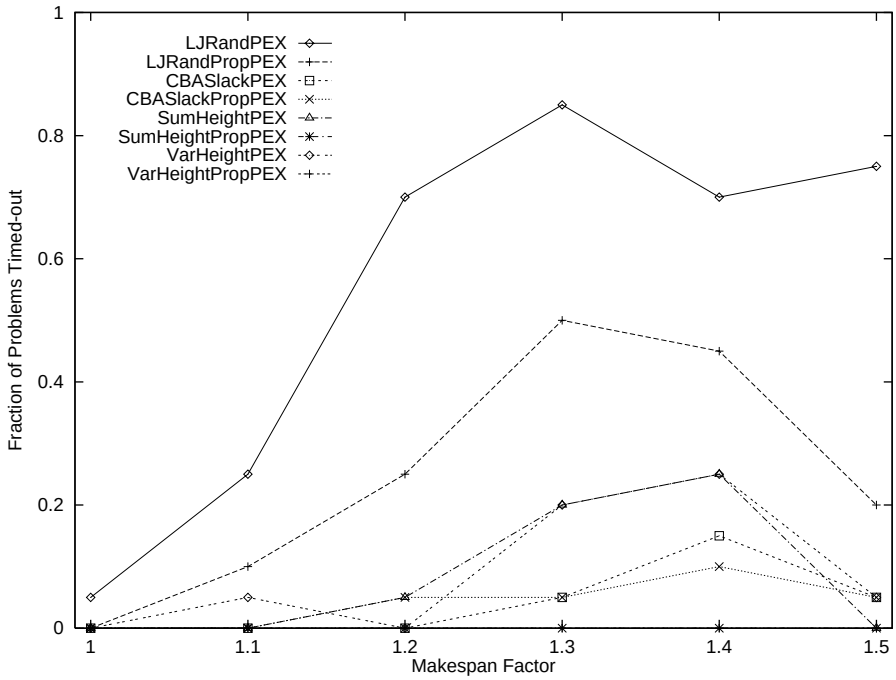


Figure 113. The Fraction of Problems with Seven Alternatives for which Each Algorithm Timed-out.

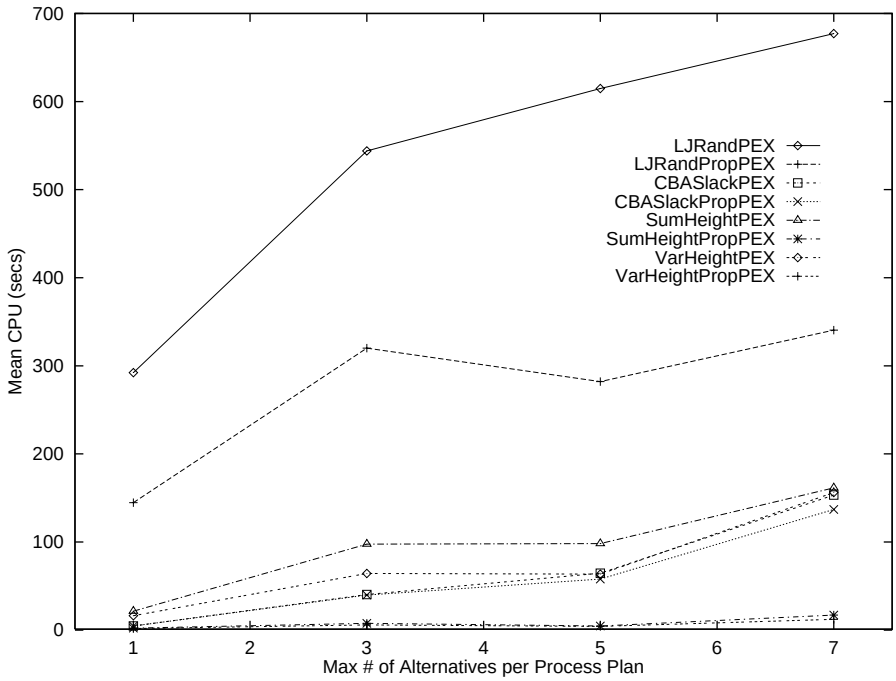


Figure 114. The Mean CPU Time in Seconds for Each Problem Set.

Holding the heuristic component constant, we see that SumHeightPropPEX, VarHeightPropPEX, and LJRandPropPEX all incur a lower mean CPU time than their corresponding non-PEX-edge-finding algorithms, while there is no difference between CBASlackPropPEX and CBASlackPEX.

Figure 115 echoes the overall results: while there is no difference between SumHeightPropPEX and VarHeightPropPEX, both are significantly better than CBASlackPropPEX which in turn is

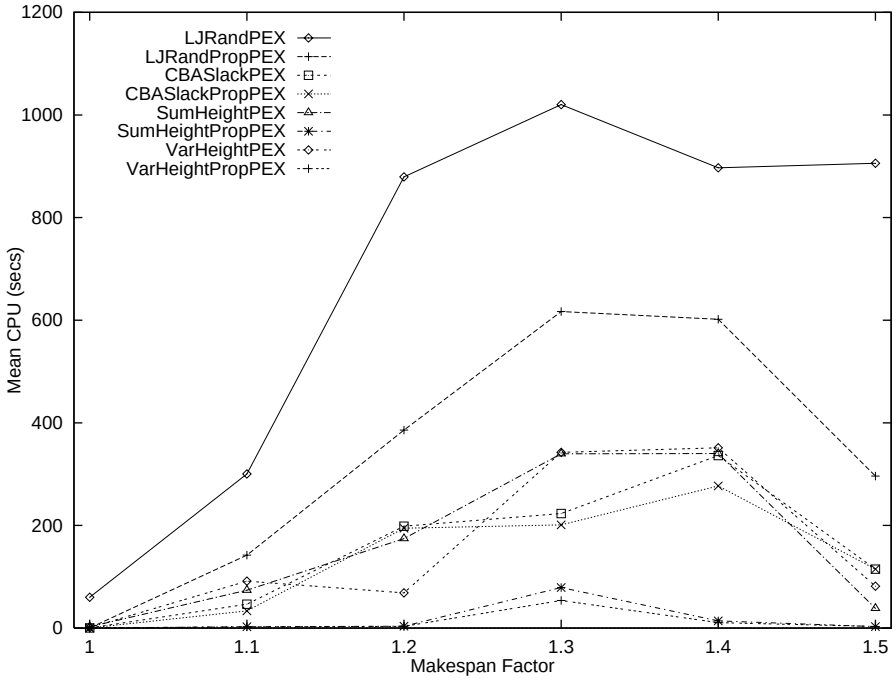


Figure 115. The Mean CPU Time in Seconds for the Problems with Seven Alternatives at Each Makespan Factor.

significantly better than LJRandPropPEX. Without PEX-edge-finding, the differences among SumHeightPEX, VarHeightPEX, and CBASlackPEX heuristics disappear while all are still better than LJRandPEX. PEX-edge-finding proves to significantly reduce CPU time when used with SumHeightPEX, VarHeightPEX, and LJRandPEX, but not with CBASlackPEX.

Other Search Statistics

Other search statistics reveal the following results:

- VarHeightPropPEX makes significantly fewer backtracks ($p \leq 0.001$), commitments ($p \leq 0.005$), and heuristic commitments ($p \leq 0.0005$) than SumHeightPropPEX. SumHeightPropPEX makes significantly fewer backtracks, commitments, and heuristic commitments than CBASlackPropPEX which in turn makes significantly fewer backtracks, commitments, and heuristic commitments than LJRandPropPEX.
- VarHeightPEX, SumHeightPEX, and CBASlackPEX each make significantly fewer backtracks, commitments, and heuristic commitments than LJRandPEX. In addition, VarHeightPEX makes significantly fewer backtracks ($p \leq 0.005$) and heuristic commitments ($p \leq 0.005$) than SumHeightPEX. The only other significant differences are in the overall commitments, where VarHeightPEX makes significantly fewer than ($p \leq 0.005$) SumHeightPEX which in turn makes significantly fewer than CBASlackPEX ($p \leq 0.005$).
- With each heuristic, the use of PEX-edge-finding results in significantly fewer backtracks and heuristic commitments ($p \leq 0.0005$ when CBASlackPEX is the heuristic). In terms of total commitments, LJRandPropPEX is not significantly different from LJRandPEX, while the difference is significant for the other three heuristic commitment techniques ($p \leq 0.005$ for CBASlackPropPEX versus CBASlackPEX).

Problem Set (Size of Underlying Jobshop Problem)	Number of Activities Per Problem		
	Minimum	Mean	Maximum
5✕5	36	61.7	85
10✕10	172	204.7	251
15✕15	356	430.9	524
20✕20	652	738.8	857

Table 12. **The Characteristics of the Problems in Experiment 2.**

7.9.1.3 Summary

Experiment 1 indicates that:

- The LJRandPEX heuristic when used with or without PEX-edge-finding performs significantly worse than the other heuristics across all the problem sets.
- While there is no significant difference in terms of the number of problems timed-out among SumHeightPropPEX, VarHeightPropPEX, and CBASlackPropPEX, all other search statistics indicate VarHeightPropPEX and SumHeightPropPEX perform significantly better than CBASlackPropPEX.
- There is little performance difference among VarHeightPEX, SumHeightPEX, and CBASlack-PEX.
- PEX-edge-finding improves scheduling performance when used with the VarHeightPEX, Sum-HeightPEX, and LJRandPEX heuristics. No such improvement was found with CBASlackPEX heuristic.

7.9.2 Experiment 2: Scaling with Problem Size

The purpose of Experiment 2 is to examine the scaling behaviour of the algorithms as the problem size gets larger, but the number of alternatives remains fixed.

7.9.2.1 Problems

The problems used in this experiment are generated in the same way as the problems in Experiment 1. The difference is that rather than changing the maximum number of alternatives in different problem sets, we hold that parameter fixed at five while varying the size of the underlying job shop problem. All problems in Experiment 1 had a 10✕10 underlying job shop problem. For this experiment, we look at problems whose underlying job shop problems are of sizes 5✕5, 10✕10, 15✕15, and 20✕20. The problem set from Experiment 1 with five alternatives is used again in Experiment 2. As with Experiment 1, we use the job lower bound and varying makespan factors to create six sets of 20 problems each at each problem size.

To be more specific, in generating a 20✕20 problem, we start with 100 jobs each with 20 activities and 20 resources. These are combined, as discussed above, into a problem with 20 process plans with alternatives. A solution to such a problem contains 400 activities with PEX value equal to 1.

Due to the problem generation, different problem instances of the same size problems have slightly varying numbers of activities. Table 12 shows the characteristics of the problem sets.

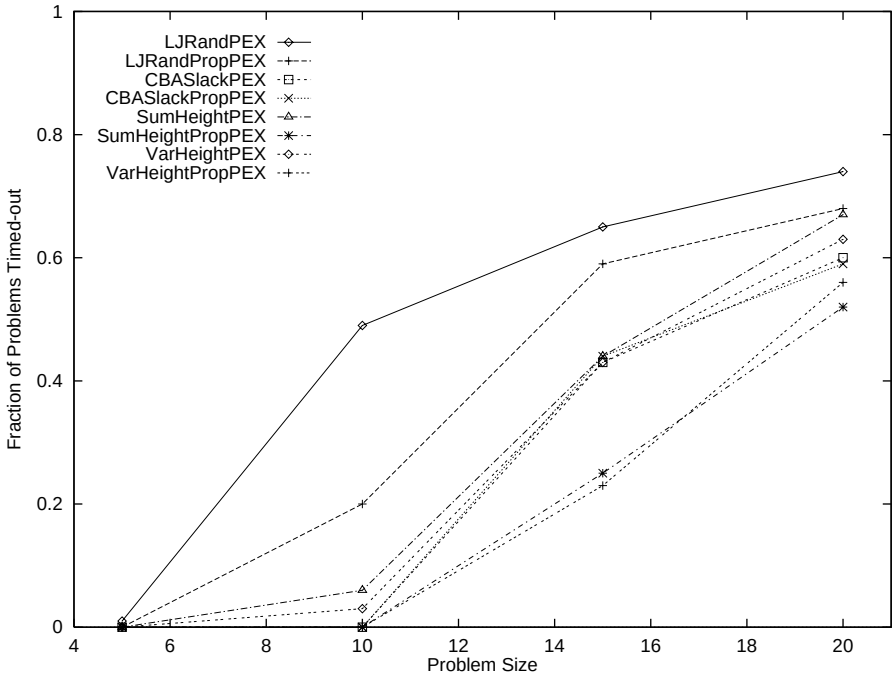


Figure 116. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out.

7.9.2.2 Results

Complete results of Experiment 2 can be found in Section D.1 of Appendix D. In particular the results for each problem size can be found in the following sections of Appendix D: Overall – Section D.1.1, 5×5 – Section D.1.2, 10×10 – Section D.1.3, 15×15 – Section D.1.4, 20×20 – Section D.1.5.

The proportion of problems timed-out for each algorithm is shown in Figure 116. SumHeightPropPEX and VarHeightPropPEX time-out on significantly fewer problems than CBASlackPropPEX which in turn times-out on significantly fewer problems than LJRandPropPEX. Without PEX-edge-finding, the results are slightly different as CBASlackPEX times-out on significantly ($p \leq 0.001$) fewer problems than SumHeightPEX which in turn times-out on significantly fewer problems than LJRandPEX. VarHeightPEX times-out on significantly fewer problems than LJRandPEX, but there are no significant differences between VarHeightPEX and SumHeightPEX or between VarHeightPEX and CBASlackPEX. CBASlackPropPEX shows no significant difference when compared to CBASlackPEX while SumHeightPropPEX, VarHeightPropPEX, and LJRandPropPEX time-out on significantly fewer problems than their respective non-PEX-edge-finding algorithms.

Focusing on the 20×20 problems, we see the fraction of each problem set that times-out in Figure 117. The only significant differences here are that the LJRandPEX algorithms are significantly worse than the other algorithms with or without PEX-edge-finding ($p \leq 0.005$) and SumHeightPropPEX is significantly better than SumHeightPEX.

Turning to the mean CPU time results, the overall results are shown in Figure 118 while the results for the 20×20 problem set are shown in Figure 119. For the overall results, SumHeightPropPEX and VarHeightPropPEX each incur significantly less CPU time than CBASlackPropPEX which in turn incurs significantly less CPU time than LJRandPropPEX. Without PEX-

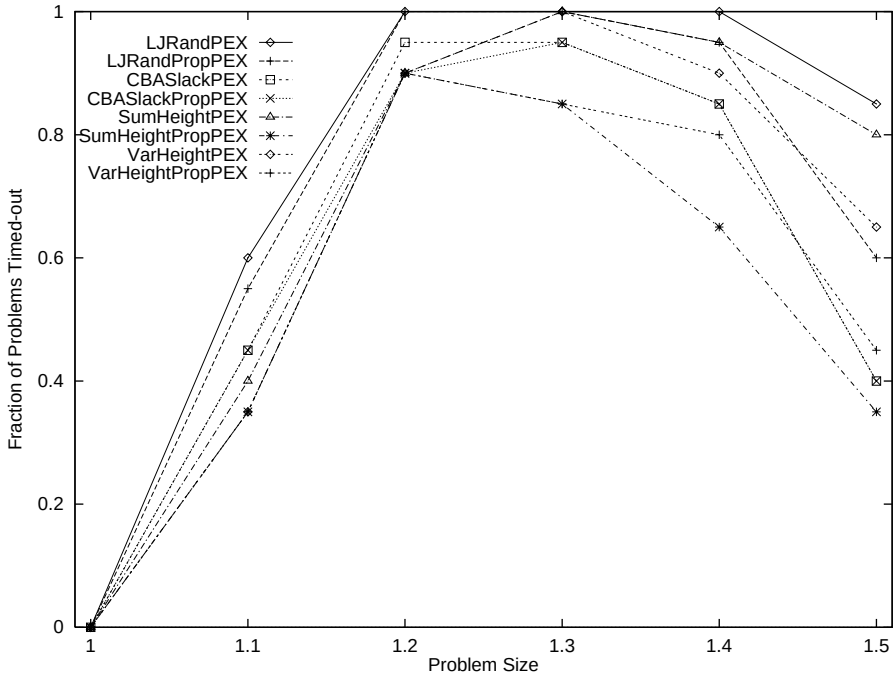


Figure 117. The Fraction of the 20×20 Problems at Each Makespan Factor for which Each Algorithm Timed-out.

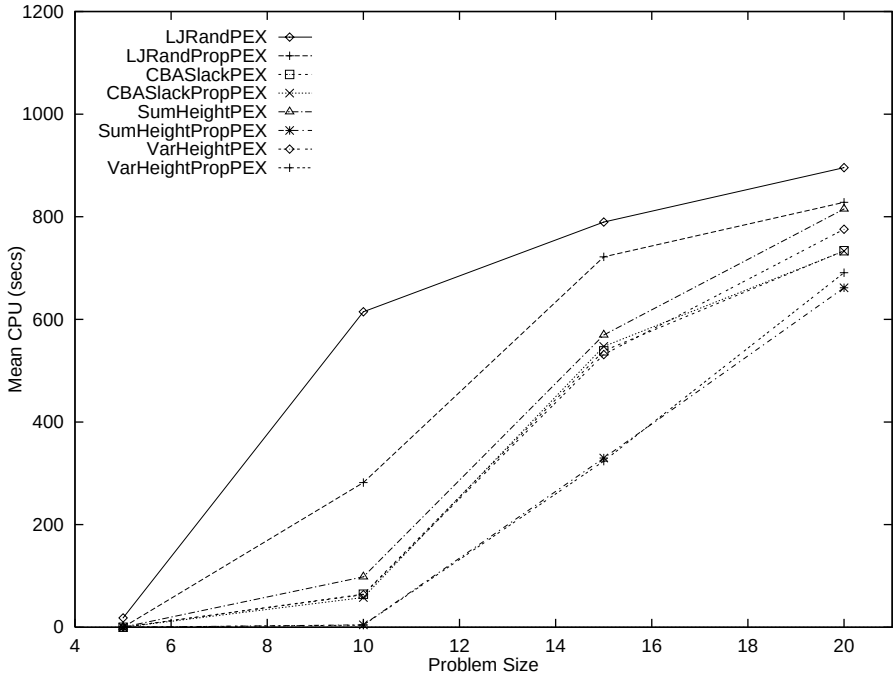


Figure 118. The Mean CPU Time in Seconds for Each Problem Set.

edge-finding, however, CBASlackPEX uses significantly less CPU time than SumHeightPEX ($p \leq 0.0005$) which in turn uses significantly less CPU time than LJRandPEX. There are no significant overall differences between VarHeightPEX and SumHeightPEX, or between VarHeightPEX and CBASlackPEX. Evaluation of PEX-edge-finding shows, overall, that using it results in significantly lower CPU time with SumHeightPEX, VarHeightPEX, and LJRandPEX, but not with CBASlackPEX.

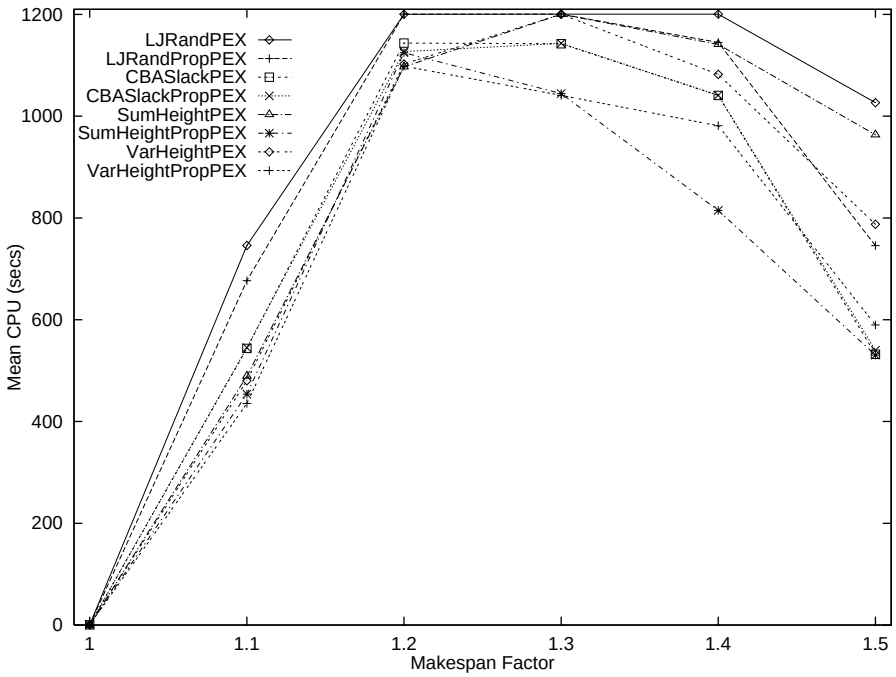


Figure 119. The Mean CPU Time in Seconds for the 20×20 Problems at Each Makespan Factor.

The results for the 20×20 problems show that regardless of the use of PEX-edge-finding, there are no significant differences among SumHeightPEX, VarHeightPEX, and CBASlackPEX while all three are significantly better than LJRandPEX ($p \leq 0.0005$). Use of PEX-edge-finding exhibits lower mean CPU time for SumHeightPEX and LJRandPEX ($p \leq 0.005$ for LJRandPEX condition), but shows no significant difference for VarHeightPEX or CBASlackPEX.

Other Search Statistics

A summary of the other search statistics is as follows:

- SumHeightPropPEX and VarHeightPropPEX each make significantly fewer backtracks, commitments, and heuristic commitments than both CBASlackPropPEX and LJRandPropPEX. There are no significant differences in these statistics between CBASlackPropPEX and LJRandPropPEX, or between SumHeightPropPEX and VarHeightPropPEX.
- VarHeightPEX makes significantly fewer backtracks ($p \leq 0.001$) and heuristic commitments ($p \leq 0.005$) than SumHeightPEX while making significantly fewer backtracks, commitments, and heuristic commitments than either CBASlackPEX or LJRandPEX. SumHeightPEX and CBASlackPEX each make significantly fewer backtracks and heuristic commitments than LJRandPEX. In addition, SumHeightPEX makes fewer commitments than either CBASlackPEX or LJRandPEX. There are no other significant differences among the non-PEX-edge-finding algorithms.
- The algorithms using PEX-edge-finding all make significantly fewer backtracks and heuristic commitments than their counterparts that do not use the propagator. In terms of total commitments, however, the only significant difference is that CBASlackPropPEX makes significantly fewer than CBASlackPEX.

Number of Alternative Resources for an Activity	Probability
1	0.03125
2	0.5
3	0.25
4	0.125
5	0.0625
6	0.03125

Table 13. **The Distribution of Alternative Resources for the Problems in Experiment 3.**

7.9.2.3 Summary

The results of Experiment 2 indicate that:

- With PEX-edge-finding, the algorithms using the VarHeightPEX and SumHeightPEX heuristics outperform the one using CBASlackPEX which in turn outperforms the one using LJRandPEX.
- Without PEX-edge-finding, however, the relative performance changes: CBASlackPEX outperforms SumHeightPEX which in turn outperforms LJRandPEX. VarHeightPEX shows few significant differences with either SumHeightPEX or CBASlackPEX while significantly outperforming LJRandPEX.
- PEX-edge-finding typically results in better overall performance when used with SumHeightPEX, VarHeightPEX, and LJRandPEX. There is little difference between CBASlackPropPEX and CBASlackPEX.

7.10 Combining Alternative Process Plans and Alternative Resources

In our final experiment, we look at problems containing both alternative process plans and alternative resources.

7.10.1 Experiment 3: Scaling with the Number of Alternatives

7.10.1.1 Problems

All problems for this experiment are transformations of the problems used in Experiment 1 above. In those problems, each activity has only one resource alternative and a single possible duration.

Alternative resources were added to each activity by randomly generating the total number of resource alternatives by following the distribution shown in Table 13. The original resource requirement and duration on that resource are preserved in the new problem. In addition, the new resource alternatives (if any) are randomly chosen with uniform probability from among all the other resources in the problem. The duration of the activity on each new alternative resource is generated by multiplying the activity's original duration by a randomly chosen factor in the domain [1.0, 1.5] and then rounding to the nearest integer value.

These transformations result in problems such that:

- The original job lower bound calculated in Experiment 1 is still a valid lower bound. All alternative resources require the activity to have at least as large a duration as in the original problem; therefore, the shortest path in a job, including resource alternatives, remains the same.
- There is likely to be widely varying PEX values between activities. The theoretical range on the PEX value of an activity, *A*, at the beginning of a problem with seven alternatives is shown in Expression (37) (below). The minimum PEX value is possible for an activity that represents one of six possible resource alternatives while the original activity (without resource alternatives) was nested inside the seven process plan alternatives. Widely ranging PEX values represent non-uniformities in problem structure: an activity with a high PEX value is far more likely to execute than an activity with a low PEX value. We would expect, therefore, that heuristics that reason explicitly about the PEX value (*i.e.*, SumHeightPEX and VarHeightPEX as opposed to the other heuristic commitment techniques) will be able to make higher quality commitments.

$$\frac{1}{3} \times 2^{-8} \leq A_{PEX} \leq 1 \tag{37}$$

7.10.1.2 Results

Complete results of Experiment 3 can be found in Section D.1 of Appendix D. In particular the results for each problem size can be found in the following sections of Appendix D: Overall – Section D.1.1, One Alternative – Section D.1.2, Three Alternatives – Section D.1.3, Five Alternatives – Section D.1.4, Seven Alternatives – Section D.1.5.

The fraction of problems in each problem set that each algorithm timed-out on is displayed in Figure 120. These results indicate, regardless of the use of PEX-edge-finding, that SumHeightPEX outperforms VarHeightPEX which is better than LJRandPEX which in turn outperforms CBASlackPEX. In addition, each heuristic times-out on significantly fewer problems when using PEX-edge-finding than without it.

The results for mean CPU time are displayed in Figure 121. These results are consistent with the timed-out results on all accounts. Comparing heuristics shows that SumHeightPEX has a significantly lower mean CPU time than VarHeightPEX. VarHeightPEX incurs significantly less CPU time than LJRandPEX which in turn has a significantly lower mean CPU time than CBASlackPEX. The PEX-edge-finding results indicate that each heuristic incurs a significantly lower mean CPU time when PEX-edge-finding is used.

Other Search Statistics

The other search statistics evaluated (number of backtracks, number of commitments, and number of heuristic commitments) agree in the relative ranking of the performance of each heuristic: regardless of the PEX-edge-finding condition, SumHeightPEX significantly outperforms VarHeightPEX which significantly outperforms LJRandPEX which in turn significantly outperforms CBASlackPEX. The only exception to this pattern is in comparing the number of heuristic commitments made by LJRandPEX and CBASlackPEX where there is no significant difference.

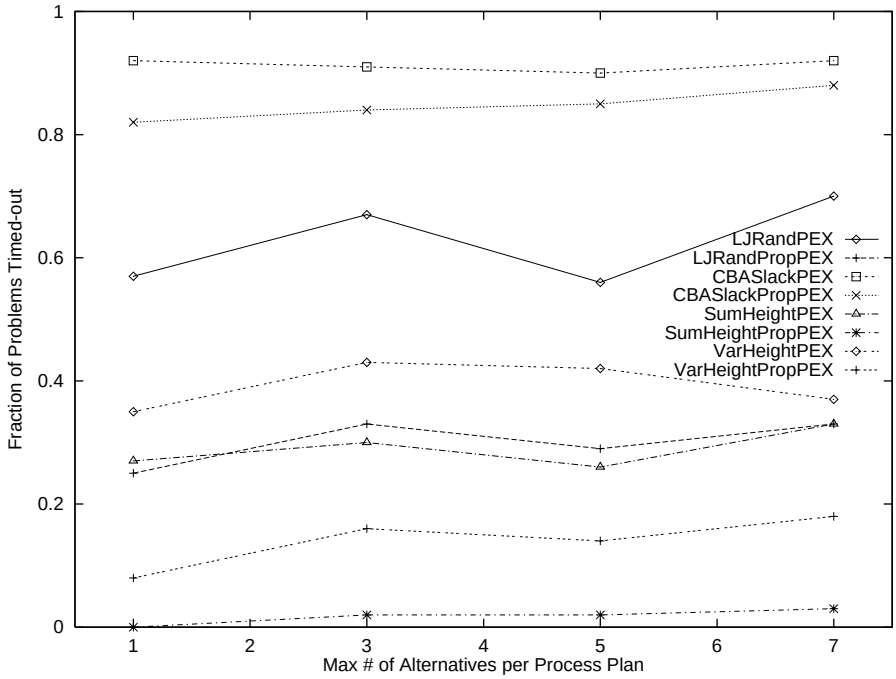


Figure 120. The Fraction of Problems in Each Problem Set for which Each Algorithm Timed-out.

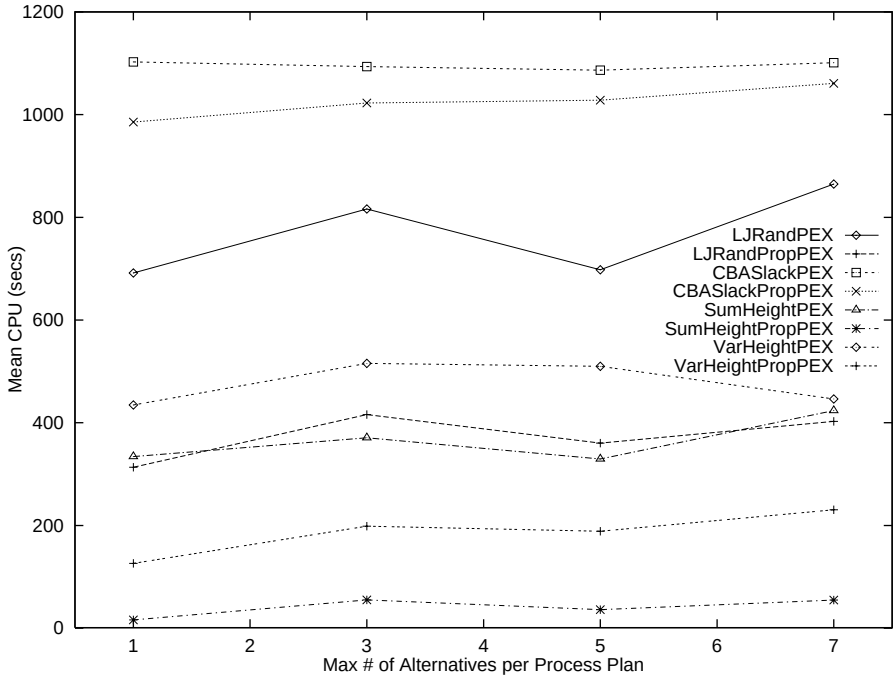


Figure 121. The Mean CPU Time in Seconds for Each Problem Set.

All heuristics exhibited significantly fewer backtracks and fewer heuristic commitments when used with PEX-edge-finding. Interestingly, VarHeightPEX, CBASlackPEX, and LJRandPEX all made significantly more overall commitments when using PEX-edge-finding than when not using it. There is no significant difference in overall commitments between PEX-edge-finding conditions when SumHeightPEX is the heuristic commitment technique.

7.10.1.3 Summary

Experiment 3 strongly indicates that, independent of the propagator condition, SumHeightPEX outperforms VarHeightPEX which outperforms LJRandPEX which in turn outperforms CBASlackPEX. In addition, PEX-edge-finding results in better performance regardless of the heuristic used.

7.11 Discussion

Our discussion of the experiments in this chapter first looks at the two goals of the experiments: evaluation of the heuristics and evaluation of PEX-edge-finding. We then turn to broader issues raised by these experiments.

7.11.1 Heuristics

Overall, while SumHeightPEX and VarHeightPEX outperform CBASlackPEX with PEX-edge-finding, especially in Experiment 3, their comparison without PEX-edge-finding is less clear-cut. In Experiment 1 there are few differences, in Experiment 2, CBASlackPEX is superior to SumHeightPEX, while in Experiment 3 SumHeightPEX is superior to VarHeightPEX which is superior to CBASlackPEX. LJRandPEX is outperformed by all other heuristics (regardless of the use of PEX-edge-finding) in Experiments 1 and 2, but outperforms CBASlackPEX (again regardless of the use of PEX-edge-finding) in Experiment 3.

Experiment 3 clearly demonstrates the result of exploiting the extra information represented by the PEX values in the texture measurement upon which the heuristic commitment technique is based. The widely ranging PEX values represent a non-uniformity in the problem structure that is successfully exploited by the SumHeightPEX and VarHeightPEX heuristics. In the other experiments, though the range of PEX values is not as wide, SumHeightPropPEX and VarHeightPropPEX are still the best scheduling algorithms of the ones tested.

The comparison between CBASlackPEX and SumHeightPEX is interesting:

- Experiment 1 shows no difference regardless of the use of PEX-edge-finding.
- Experiment 2 shows that CBASlackPEX is better than SumHeightPEX without PEX-edge-finding, but worse with PEX-edge-finding.
- Experiment 3 shows that CBASlackPEX is much worse than SumHeightPEX both with and without PEX-edge-finding.

These results demand an explanation both for why CBASlackPEX performs so poorly in Experiment 3 and why it performs so well in Experiments 1 and 2.

7.11.1.1 Why is CBASlackPEX So Good?

We know, based on the results of Chapter 4, that on some job shop scheduling problem sets without resource-level non-uniformities, CBASlack is able to outperform SumHeight. No such non-uniformities were examined in this chapter as our primary interest was the evaluation of reasoning about alternatives. One possibility as to reasons for the good performance of CBASlackPEX returns to our reasoning for Chapter 4: no resource-level non-uniformity exists for our experimental problem sets therefore the heuristic that does not specifically look for such non-uniformity achieves better overall performance.

A second explanation for the good quality of CBASlackPEX is the quality of the underlying biased-slack heuristic. When there are no resource-level non-uniformities, a heuristic based on the identification of minimum-slack activity pairs and the subsequent commitment to maintain as much slack as possible, has been shown to perform well in job shop scheduling. Some of the quality of that heuristic is maintained when PEX is added. Again, in the absence of resource-level non-uniformities, a heuristic that identifies the pair of activities with the smallest time in which they may be sequenced and seeks to maximize that time (through a PEX commitment or a sequencing commitment) seems to be a reasonable heuristic to establish both a selection of activity alternatives and a sequencing of activities that satisfies the resource constraints.

7.11.1.2 Why is CBASlackPEX So Bad?

As discussed below (Section 7.11.3), one explanation for the superior performance of SumHeightPEX as compared to CBASlackPEX is the extra information that the former incorporates into its heuristic decision making. Another possible (and not incompatible) explanation, that in addition may help to explain the relative results of CBASlackPEX and LJRandPEX, is that the ability of biased-slack calculation to identify critical activity pairs is degraded by the presence of PEX values.

In the original CBASlack heuristic, only activity pairs for which CBA propagation has not inferred a sequence are examined to find the pair with minimum biased-slack. When at least one member of an activity pair has a PEX value of less than 1, however, the CBA propagator cannot infer a sequence and therefore, regardless of the activity time windows, the biased-slack of the activity pair is evaluated. Activity pairs with little overlap or even disjoint time windows will tend to have a very low biased-slack and therefore the heuristic will tend to focus on such activity pairs. To take the extreme case of an activity pair with disjoint time windows, the activities are not actually competing with each other for a resource reservation and so can hardly be a truly critical pair, yet they have a very small biased-slack. Furthermore, the commitment to set the PEX value of one of the activities to 0, is made on the intuition that, if such a commitment is not made, there is a high likelihood that the two activities will later over-capacitate the resource. But this cannot be the case with disjoint activities, and the likelihood that an activity pair with a small overlap will lead to an over-capacity is relatively small.

This reasoning points to a modification of the CBASlackPEX heuristic to calculate the biased-slack only for those activity pairs that would not have an implied sequence if they both had a PEX value of 1. Such a modification remains as future work.

7.11.1.3 SumHeightPEX and VarHeightPEX

In the experiments in this chapter, VarHeightPEX achieves equal or better performance than all other non-texture-based heuristics tested. Given the wider applicability of the VarHeight texture measurement, as demonstrated in Chapter 6, these results confirm the practical utility of the probability of breakage measure of criticality.

A comparison of the performance VarHeightPEX and SumHeightPEX shows that, in the first two experiments in this chapter, VarHeightPEX achieved about the same level of performance as SumHeightPEX. In fact, on a few performance statistics (*e.g.*, the number of backtracks in Experiment 2) VarHeightPEX performs significantly better than SumHeightPEX. In Experiment 3, however, SumHeightPEX is significantly better on all performance measures. These results parallel the results seen in Chapter 5: on the randomly generated problems VarHeight and SumHeight performed at about the same level while when specific structure was added in the form of resource-

level non-uniformities, SumHeight performed significantly better. In this chapter, the added structure is the widening of the range of PEX values resulting from the addition of alternative resources to the alternative process plan problems.

As in Chapter 5 (see Section 5.7), these performance differences raise a number of questions. Is it the case that the aggregate demand estimation carried out by SumHeightPEX reveals useful structural information that is not found by the probability of breakage estimation of VarHeightPEX? It was argued in Section 5.7 that SumHeight makes more use of the magnitude of the possible resource over-capacity than VarHeight. Given the importance of the PEX values in scaling the magnitude of the individual demands, it may be the case that making deeper use of the aggregate magnitude allows SumHeightPEX to be more sensitive to the PEX values than VarHeightPEX.

As we also noted in Chapter 5, the existence of structural information captured by aggregate demand, but not by the probability of breakage, is a speculative point. Other factors such as the accuracy and computational complexity of estimation algorithms, and how well a measurement of criticality correlates with true criticality likely play complex and interdependent roles in the overall heuristic search performance. The unraveling of such factors, together with the deeper analysis of the information examined by a measure of criticality and estimated by a texture measurement, remain key areas of future work.

7.11.2 PEX-Edge-Finding

Through almost all experiments and experimental conditions, PEX-edge-finding was shown to be beneficial to the overall problem solving ability of an algorithm. The only exception is when the CBASlackPEX heuristic is used: little difference was seen between the CBASlackPEX algorithm with or without PEX-edge-finding except in Experiment 3 where PEX-edge-finding proved beneficial.

In general, these results are as expected. Given the significant increase in the performance of scheduling algorithms with the use of edge-finding propagators [Nuijten, 1994], we expect that some gain is likely with the PEX-edge-finding variation.

Our intuitions as to why PEX-edge-finding (and indeed propagation, in general) should improve search performance rests on two impacts of propagation. First, propagation techniques reduce the search space by removing search alternatives that would otherwise have to be searched through. Second, and more central to this dissertation, is the idea that propagators improve the search information upon which heuristics are based. PEX-edge-finding improves both the information represented in the PEX values (by pruning inconsistent activity alternatives) and the information represented in the time windows of activities (because it infers unary temporal constraints on activities with PEX values of less than one). SumHeightPEX benefits from both improvements. The former is reflected in the PEX values of the remaining activities while the latter is represented in the individual activity demand which, of course, depends on the time window as well as on the PEX value of each activity.

The results of the CBASlackPEX heuristic do not match our expectations as there is little if any benefit from the use of PEX-edge-finding in the first two experiments. While we have no clear explanation for these results, at least two possibilities can be suggested.

1. CBASlackPEX does not use PEX values. As discussed above (Section 7.6.3), CBASlackPEX does not make direct use of the PEX values in forming its heuristic commitments. Rather it treats the PEX variable as three-valued: 0, 1, or neither. Since PEX-edge-finding improves

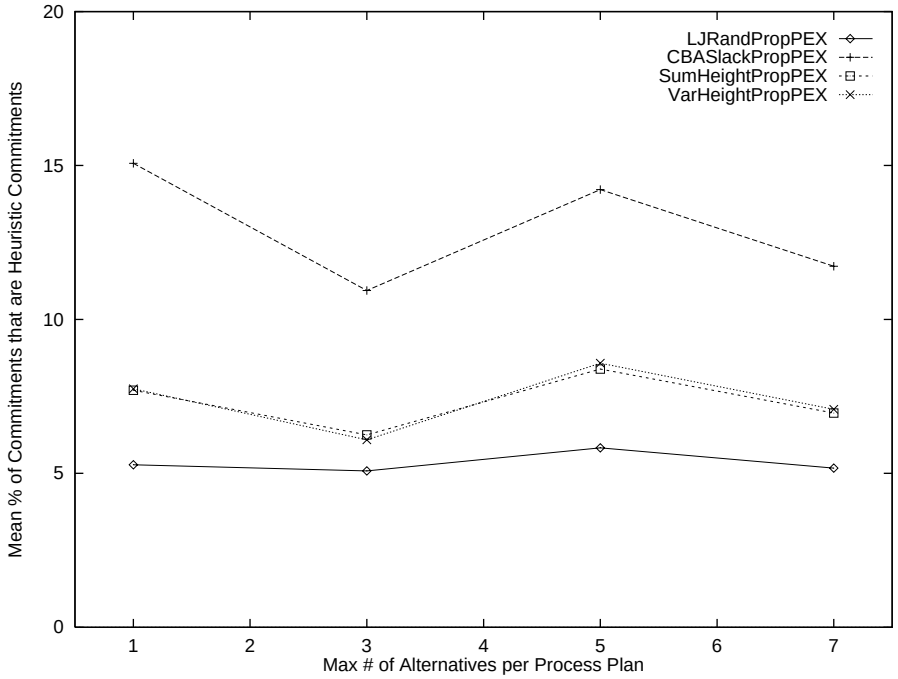


Figure 122. The Mean Percentage of Commitments in Each Problem Set Made by the Heuristic Commitment Technique for Experiment 1.

information represented in the PEX values (when alternatives are removed), perhaps CBASlackPEX does not benefit because it does not make use of this extra information.

2. A CBASlackPEX commitment results in less propagation. In the job shop problems in Chapter 4, we observed that a relatively large percentage of the CBASlack commitments were heuristic commitments as compared to those for SumHeight. We argued (Section 4.9.1.3) that this arose from the fact that SumHeight tended to make a commitment where many activities were competing for a resource and time point while CBASlack simply looked for the most tightly constrained pair of activities. The subsequent propagation from a heuristic commitment therefore would tend to be higher when SumHeight is used. The same pattern can be seen here as shown in Figure 122. Using the results from Experiment 1, we see that SumHeightPropPEX and VarHeightPropPEX make a significantly smaller percentage of heuristic commitments than CBASlackPropPEX. While a CBASlackPEX algorithm using PEX-edge-finding incurs the computational cost, the benefits are not apparent because the heuristic commitments do not result in significant propagation.

7.11.3 Exploiting Non-uniformities of Problem Structure

As noted in Section 7.6.3, of all heuristics described and tested in this chapter, only SumHeightPEX and VarHeightPEX make use of the actual value of the PEX variables in each search state. One explanation for the dramatic difference in relative heuristic quality between the first two experiments and the third is the fact that the non-uniformity in PEX values is much greater in the third experiment. Just as the SumHeight heuristic exploits the non-uniformity at the resource-level in job shop problems (Experiment 3 of Chapter 4), SumHeightPEX exploits the non-uniformities among PEX values. This explanation supports the central thesis of this dissertation that the analysis of the constraint graph to reveal dynamic information can then be used for high-quality heuristics.

7.11.4 Alternative Resources

The experimental evaluation of the techniques presented in this chapter focuses on problems with alternative process plans, and problems with both alternative process plans and alternative resources. Exactly the same techniques, however, are also applicable to scheduling problems with alternative resources but without alternative process plans. Each activity with resource alternatives can be transformed into a set of alternative activities (subject to a pair of XorNodes) such that there is one activity for each resource alternative.

A separate evaluation of the PEX techniques performed in collaboration with other researchers [Davenport et al., 1999], examines such problems. While these experiments (and the non-PEX techniques for alternative resources) are not part of this dissertation, the results do speak to the efficacy of the PEX techniques and so we briefly examine them here.

In addition to the PEX techniques (PEX-edge-finding and VarHeightPEX), [Davenport et al., 1999] examines techniques for alternative resource problems that decompose the problem into two phases. In the first phase, the resource assignments are made for each activity based on texture measurement information about each of the resource alternatives. The second phase then treats the problem as a pure scheduling problem and sequences activities as discussed in Chapter 4. While the PEX techniques are motivated by the intuition that high-quality heuristic commitments require representation and analysis of the dynamic information at each search state, the non-PEX techniques are more motivated by the intuition that the computational cost of representing and reasoning about such search knowledge can be prohibitively high. It may be more useful, therefore, to make a different trade-off between the quality of heuristic commitments, and the computational overhead of representing and reasoning about the information upon which the heuristic commitments are made.

The experimental results for backtrack-free search indicate that the PEX-based techniques result in substantially higher quality solutions. In some problems, the PEX techniques are able to find solutions more than 20% closer to the optimal than any other techniques. As expected, however, the cost of representing and reasoning about PEX can sometimes be prohibitive. In the largest problems (*e.g.,* 800 activities with each activity having eight alternative resources), the PEX representation requires about 7200 temporal nodes (activities and XorNodes). This problem size proved too large in terms of memory requirements to successfully complete the experiments. The non-PEX techniques, however, are able to generate solutions to such large problems ranging from 50% to 120% above the lower bound for the problems.

Detailed discussion of the non-PEX techniques as well as the experimental results are beyond the scope of this dissertation. See [Davenport et al., 1999] for a full description.

7.12 Conclusion

In this chapter, we introduced the notion of the probability of existence (PEX) of an activity and used it to expand constraint-directed scheduling representation and reasoning to account for alternative activities: an activity present in the original problem definition does not necessarily have to be scheduled to achieve a solution.

The modeling of PEX required extensions to the constraint representation of activities and of the temporal network. Algorithms for the propagation of PEX values are created along with modifications to temporal propagation to account for the fact that an activity might not exist in a final solu-

tion. In addition, heuristic commitment techniques and two edge-finding propagators are extended to account for PEX values.

Experimental results indicated that incorporating PEX values into the texture measurements upon which the heuristic commitment techniques are based results in significantly higher quality commitments and better overall search performance. Performance differences are especially large when there is a wide range of PEX values in a problem. Experimental results also validated the use of PEX-edge-finding which, in most cases, leads to significantly better overall search performance.

Chapter 8 Conclusions and Future Work

8.1 Contributions

8.1.1 Major Contributions

The major contributions of this dissertation concern the analysis of scheduling algorithms, the demonstration of the importance of the knowledge of problem structure in heuristic search, and the extension of constraint-directed scheduling, in general, via the use of the criticality of constraints. This extension is specifically demonstrated in the context of scheduling with inventory and scheduling with alternative activities.

8.1.1.1 Analysis and Categorization of Scheduling Algorithms

We performed an analysis of algorithms in order to determine the main factors contributing to their ability to solve scheduling algorithms. This analysis was based on the ODO framework and its categorization of the three key components of constraint-directed scheduling algorithms: heuristic commitment techniques, propagators, and retraction techniques. By adopting such a component-based view of existing scheduling algorithms, we were able to rigorously examine claims of superior scheduling performance among sets of algorithms. More importantly, our conceptual (and implementational) model of scheduling algorithms enabled us to isolate and vary specific components (*e.g.*, heuristic commitment techniques) while holding all other components constant. This allowed us to attribute specific algorithmic performance claims to components rather than to the global scheduling algorithm. This is a significant step as it allows a deeper, more detailed understanding of scheduling algorithms, the analysis of their performance on scheduling problems, and the investigation of underlying reasons for such performance.

8.1.1.2 The Importance of Problem Structure in Heuristic Search

Based on our analysis of scheduling algorithms, we focused specifically on heuristic commitment techniques for constraint-directed scheduling. The central hypothesis we investigated, and were able to demonstrate, was that as a problem becomes more complex, knowledge of the problem structure has a dominant role in guiding heuristic search to a solution. This hypothesis is originally due to early work of Simon [Simon, 1973], and has been re-examined from a constraint perspective by Fox et al. [Fox, 1983; Fox et al., 1989; Sadeh, 1991] and from the perspective of phase transition in combinatorial search in the work of [Gent et al., 1996b].

In particular, we were able to demonstrate that knowledge of the problem structure leads to superior heuristic search performance in three specific classes of problems: job shop scheduling, scheduling with inventory, and scheduling with alternative activities.

In the context of job shop scheduling, we showed that higher quality heuristic search is attained by the dynamic identification of critical resource constraints. When experimental problems were manipulated to specifically insert resource-level non-uniformities (*i.e.*, bottlenecks), the heuristic that specifically identified and exploited such non-uniformities was shown to provide significantly better overall scheduling performance.

In scheduling problems with representation of inventory storage, production, and consumption, we demonstrated that the best overall heuristic search performance is achieved by a texture-based heuristic that measures the criticality of both inventory and resource constraints in each problem state. Such heuristics are able to dynamically focus attention on the most critical constraint, regardless of type. Furthermore, as we increased the complexity of the inventory relationships among activities in the experimental problems, the magnitude of superiority of the texture-based heuristic also increased.

Finally, we examined problems where the activities in the original problem definition did not necessarily have to execute in a solution. In such problems, we explicitly represented estimates of the probability that an activity in the original problem definition would exist in a final schedule. We demonstrated that heuristics incorporating the structural information represented by the probability estimates are able to significantly outperform other, less knowledge-based, heuristics. As with the inventory problems, in the alternative activity problems we were able to demonstrate that the performance of the structurally sensitive heuristics rises as the complexity of choices among alternative activities increases.

8.1.1.3 The Criticality of Constraints

Though the original intent of constraint-directed scheduling was to enable reasoning about any type of constraint represented in the problem [Fox, 1983], to date, the analysis of problem structure via the use of texture measurements has focused primarily on resource constraints. To widen the scope of application of principled constraint-directed heuristic techniques and focus more on the original intent of constraint-directed scheduling, we introduced a more general concept of criticality that is applicable to any type of constraint. We discussed the possibility of different measurements of the criticality of a constraint and suggested four requirements for any such measurement. We presented the probability of breakage of a constraint as a measure of constraint criticality and showed that it meets our requirements. Three estimation techniques for the probability of breakage were introduced and incorporated into the texture-based heuristic commitment techniques of ODO. It was demonstrated that heuristic commitment techniques based on two of the estimates are able to perform as well as contention, the more narrowly defined measure of criticality used in previous texture-based heuristic techniques.

8.1.1.4 Scheduling with Inventory

We extended the ODO representation of constraint-directed scheduling problems to represent the consumption and production of inventory as well as minimum and maximum constraints on inventory storage capacity. Based on this expanded representation, we also extended the estimation techniques for the probability of constraint breakage to inventory constraints. This extension enables the estimation of the criticality of both resource and inventory constraints, and the integration of inventory constraint reasoning into the overall texture-based heuristic approach of ODO. Specifically, the ability to estimate the criticality of both resource and inventory constraints allows the construction of heuristics that are able to dynamically and opportunistically reason about the most critical constraint in a problem state, regardless of whether the constraint is a resource or inventory constraint. It was demonstrated that such heuristic commitment techniques are able to

outperform non-integrated heuristics as well as heuristics that do not directly reason about inventory constraints at all.

8.1.1.5 Scheduling with Alternate Activities

We extended the ODO framework to include the representation of alternative activities. With such an extension, the set of problems that can be represented and solved within ODO is expanded to include problems with alternative resources and alternative process plans. The representation of alternative activities greatly refines the concept of the probability of existence (PEX) of an activity first used in the KBLPS scheduling system [Saks et al., 1993; Fox, 1999]. Further, we introduced techniques for the propagation of PEX values through a temporal network, and modified the temporal propagation techniques to account for the possibility that an activity may not execute in a final schedule. Finally, the resource criticality texture measurements were generalized to take into account activities that may not exist in the final schedule. Heuristics based on these texture measurements are able to dynamically identify the most critical resource constraint in a problem state taking into account activities that must execute as well as activities that have a non-zero probability of execution. The incorporation of the magnitude of the probability of existence of each activity together with the opportunistic, dynamic focusing abilities leads to significantly better overall search performance.

8.1.2 Other Contributions

Other contributions of this dissertation include:

- The creation of two new propagation techniques for inventory scheduling. Experiments showed a significant positive impact when inventory propagation is used.
- The extension of two existing propagators (edge-finding exclusion and edge-finding not-first/not-last) so that they may be soundly applied to problems with alternative activities. Empirical results indicated a significant positive benefit from the use of the modified propagators.
- The object-oriented re-implementation of the ODO constraint-directed scheduling framework.

8.2 Future Work

Throughout this dissertation we have briefly commented on aspects of the research that remain for future work. In this section, we provide a more detailed treatment of the avenues of research suggested by the work in this dissertation.

8.2.1 Heuristics for Constraint-Directed Search

Given that the central thesis of this dissertation concerns the formulation of heuristic commitment techniques for scheduling, it is not surprising that a number of areas of future work involve basic issues about heuristic commitment techniques for use in constraint-directed search and scheduling. In this section, we look at some of these basic issues before turning to more specific issues related to scheduling.

8.2.1.1 Problem Structure and Criticality

The hypothesis explored in this dissertation is that as problems become more complex, knowledge of the problem structure becomes the dominant factor in achieving successful heuristic search behavior. The approach we have taken is to use texture measurements to distill problem structure

into a measure of criticality of individual constraints. While we have shown this approach to be successful in terms of significantly improving heuristic search performance in a variety of types of scheduling problems, there are a number of issues that remain to be explored.

Is Constraint Criticality the Appropriate Distillation of Problem Structure?

From the original formulation of texture measurements [Fox et al., 1989], the intuition has been that high-quality heuristics arise from the identification of a critical point on which a heuristic commitment technique should focus its effort. Criticality, whether with respect to a variable [Sadeh, 1991] or a constraint (Chapter 5), means expectation of failure. In a search state, variables which we expect to have an empty domain or constraints which we expect to be unsatisfied, are judged to be critical.

The intuition behind identifying and focusing on critical variables and constraints is, essentially, the fail-first principle [Haralick and Elliot, 1980]: decisions where failure is likely should be examined early in the search so that they do not later lead to exponential backtracking. The fail-first principle, however, has recently been called into question by work in CSP that shows that improving a heuristic's ability to identify where failure is likely can result in *lower* quality heuristic commitments and higher overall search cost [Smith and Grant, 1997]. Clearly, this work needs to be extended in the context of CSP, scheduling, and heuristic search in general. Is it the case that the same results can be found in constraint-directed scheduling? Given the pervasiveness of the fail-first principle, finding alternative intuitions for heuristic quality and overall search performance is a critical area for future research in heuristic search.[1]

Is the Focus on Individual Constraints Appropriate?

As noted, the texture-based heuristics in this dissertation use problem structure to estimate the criticality of individual constraints. While previous work has looked at analogs for criticality of activities, variables, and values [Fox et al., 1989; Sadeh, 1991], there has been little work that investigates the notion of criticality of larger components of a constraint-based search problem.

To focus specifically on scheduling, for example, perhaps there is information in the structure of the constraint subgraph involving a subset of resources that indicates that the subgraph containing several constraints is critical in some way. Alternatively, perhaps, a particular sub-interval of the scheduling horizon tends to have more overall contention for resources. Higher level measures of criticality may allow heuristics to focus on sub-problems, reducing computational complexity while perhaps maintaining or improving search performance. Such considerations are especially relevant to very large-scale scheduling problems where some sort of abstraction or decomposition of the problem is necessary as even algorithms with $O(n^2)$ time-complexity in a search state are prohibitively expensive when run on the full problem.

What Other Aspects of Problem Structure can be Exploited?

The texture measurements investigated in this dissertation, including the implicit texture measurement underlying the CBASlack heuristic (see Section 4.9.1.2), are all loosely based on the notion of variables competing to be assigned to values that are in conflict from the perspective of constraints in the problem state.

---

1. Some preliminary ideas regarding such intuitions are discussed in Section 4.9.1.3.

Given the abstract form of a constraint-based search problem (*i.e.*, assignment of values to variables such that all constraints are satisfied), it may be that such competition represents the fundamental structural aspect of constraint-based problem search problems. However, it has been shown that the overall graph structure (*e.g.*, graph shape such as trees [Freuder, 1982] and other characteristics such as measures of the number of constraints, variables, and values [Gent et al., 1996b]), can have significant impact on heuristic search performance.

While we continue to believe that the conceptual competition among variables is a fundamental structural component in constraint-based problems, the identification and exploitation of other structural information is an interesting direction for future research.

Can Non-uniformities of Problem Structure be Better Exploited?

In each of the experimental chapters of this dissertation, we showed that high quality heuristics can be based on the exploitation of non-uniformities in the structure of a search state. Texture measurements are algorithms that can help reveal these non-uniformities. We showed improved overall search performance based on the exploitation of non-uniformities in the context of job shop scheduling (Chapter 4), scheduling with inventory (Chapter 6), and scheduling with activity alternatives (Chapter 7).

An important question suggested by these results is: Can we design texture measurements to explicitly search for non-uniformities on a variety of levels? In Chapter 4, we showed a difference between non-uniformities at a resource-level and those at the level of activity pairs. Given more expressive problem domains with inventories, alternative process plans, etc., there are many potential sources of non-uniformity. In such problems, we should not have to rely on only one type of non-uniformity (*e.g.*, resource bottlenecks) to inform our heuristics. Rather we should be able to formulate texture measurement techniques to analyze non-uniformities at a variety of levels and reveal the non-uniformities most likely to result in high quality heuristic commitments.

There are two general approaches to such texture measurements:

1. Extend the notion of constraint criticality to a variety of levels and focus on the most critical constraint in a search state regardless of the level. For example, we may be able to adapt the CBASlack heuristic to provide an estimate of the probability of breakage of the binary constraint on each activity pair. These estimates can then be compared with estimates of the probability of breakage of the resource constraints (perhaps with the VarHeight texture) in order to identify the most critical constraint regardless of type.
2. Calculate measures of criticality at a variety of levels and use a measure of the non-uniformity at each level to guide selection of the heuristic commitment technique. Such a technique can be viewed as a meta-texture measurement: by comparing the non-uniformities of a number of texture measurements at different levels, the meta-technique is able to dynamically identify the most non-uniform level at each search state. A prosaic example of such a meta-texture measurement is a technique for job shop scheduling that calculates both SumHeight and CBASlack at each search state. By comparing a measure of the variance of the biased-slack values of the activity pairs with the variance of criticality among the resources, the technique could dynamically choose which heuristic to use.

It is unclear, *a priori*, if either of these approaches will necessarily lead to superior performance. The former approach is more in the spirit of the work done in this dissertation as it extends a measure of criticality to a wider space of constraint types. However, the latter approach, in using a measure of variance of criticality, exploits additional structural information, that we have shown

(Section 4.9.1.2) to be relevant to search performance. Investigation of these approaches remains for future work.

8.2.1.2 The Quality versus Cost Trade-off

A critical component of our investigation of texture measurements and texture-based heuristics is the practicality of such techniques. While an extremely expensive texture measurement may lead to a search that minimizes the size of the overall search tree, it would not be useful in any practical sense: too much effort would be expended at each search state.

Overall, the goal for heuristic search techniques is the minimization of the time and effort expended to solve a problem. Given the theoretical difficulty of scheduling in general, a heuristic needs to balance the computational expense of finding a commitment at a search state with the quality of that commitment. The trade-off embodied by the texture measurement-based heuristics investigated in this dissertation is to spend a low-polynomial effort (*e.g.*, $O(n^2)$ or $O(n^3)$) at each search state to find high-quality commitments that will tend to quickly converge on a solution. The hope is that the exponential expense of visiting a large portion of the search space will be avoided. In contrast, other heuristics try to spend small effort at each search space (*e.g.*, linear or constant time) in order to cover more search states. A central issue, then, is the cost of the heuristic commitments versus their quality in terms of being able to find a solution while visiting relatively few search states.

In this dissertation, we have addressed the cost versus quality trade-off by running our experiments with a CPU time bound, reporting mean CPU times, and, in some cases, investigating the algorithms when the size of the problems is varied. Nonetheless, deeper analysis of the scaling behaviour and of the quality versus cost trade-off is necessary. For example, the modeling of alternative activities with the use of PEX variables results in an explosion in the number of activities in a problem representation. While the information in the extra activities contributes greatly to the quality of heuristic decisions and, in our experiments in Chapter 7, to the overall search performance, for larger problems, the overhead of the extra activities may be prohibitive. Indeed, for the alternative resource problems examined in [Davenport et al., 1999], the PEX techniques were not able to solve some of the larger problems due to memory exhaustion.

Investigation of the applicability of high-knowledge heuristics in very large problems, creation of techniques for improving the scaling behaviour of such heuristics without significantly reducing the quality of commitments, and achievement of a better understanding of the quality versus cost trade-off all remain as interesting future research.

8.2.2 Models of Scheduling

The expansion of constraint-directed scheduling techniques to more realistic scheduling problems is one of the key motivations for the work in this dissertation. The two main characteristics of realistic scheduling problems that we examined are the presence of inventory and the presence of alternative activities. Each characteristic, their combination, and other models of scheduling problems suggest further research issues.

8.2.2.1 Scheduling with Inventory

The inventory problems examined in this dissertation are characteristic of batch environments: inventory is consumed at the start of an activity and produced at its end. While there are significant industrial applications for such functionality (*e.g.*, in the pharmaceutical industries), there is

also a need to represent and reason about activities in a more continuous environment. The characteristics of such continuous scheduling problems include:

- Activities produce and consume inventory at varying rates over their entire duration.
- Rate-matching may be required between activities in a producer/consumer relationship to ensure that inventory constraints are satisfied.
- Durations of activities may vary depending on the rates at which inventory is produced and/or consumed.

To our knowledge, such inventory constraints have not been addressed in the constraint-directed scheduling literature.

8.2.2.2 Scheduling with Alternative Activities

Representation and reasoning about alternative activities opens a number of areas of future research. While we have shown that the use of PEX variables can lead to superior scheduling performance, there are a number of issues surrounding the scaling of the PEX approach that need to be addressed. In particular, the increase in the number of activities in the representation of a problem results in scaling issues as discussed in [Davenport et al., 1999]. Investigation of reduction of these scaling problems and, perhaps, methods to reduce the alternative activities that actually need to be represented form important future tasks. Given the positive impact of the PEX-edge-finding propagator, techniques to reduce its average and worst-case time-complexity will also contribute to overall scheduling performance and help to combat scaling issues.

The PEX approach to alternative activities scheduling problems can be viewed as a bottom-up approach. Alternative activities are fully represented in the scheduling problem and choosing alternatives is fully integrated into the scheduling algorithm. In contrast, a top-down approach to scheduling with alternatives can be seen in number of places in the AI research literature [Wagner et al., 1997; Wagner et al., 1998; Kott and Saks, 1998; Sadeh et al., 1998; Davenport et al., 1999]. In such systems, high-level and, in some cases, domain dependent, knowledge and heuristics are brought to bear on the problem of pruning alternatives before the actual scheduling takes place. While these techniques avoid many of the scaling problems of the PEX approach, the heuristics are not informed by as much detailed information as the PEX heuristics. As a result, the quality of the heuristic decisions and overall solution tends to be inferior to the PEX solution [Davenport et al., 1999] in those problems where the PEX approach is not overwhelmed by the size of the required representation. Obviously, there is a trade-off here in terms of the effort required to represent and reason about the detailed alternative information and the quality of the heuristic decisions that can be made. Further research is required to better understand this trade-off and, perhaps, to find ways to combine the top-down approach with the bottom-up approach. For example, it may be that better overall performance can be achieved by making some of the alternative choices based on high-level information and allowing the rest of them to be decided during the scheduling process.

Reasoning about activity alternatives either with a PEX-based formulation or with a more top-down approach blurs the distinction between scheduling and planning. One of the major assumptions about the alternative scheduling problems addressed in this dissertation is that all possible activities are known before scheduling begins. Imagine, however, a problem with demands for a number of finished goods and a number of process plans with which each can be produced. Each finished good process plan may require a number of other inventories to be produced and these inventories, in turn, have alternative process plans which themselves require further inventories to be executed. This cascade of process plans terminates eventually with raw material supply events.

The problem now begins to look more like planning (with scheduling constraints) than scheduling. A set of goals must be achieved; however, actions to achieve goals potentially conflict with one another and may introduce further sub-goals.

The combination of scheduling and planning techniques to address such problems has been begun in work such as [Saks, 1992; Kott and Saks, 1998; Sadeh et al., 1998]; however, much remains to be investigated.

8.2.2.3 Alternative Activities and Inventory

While the modeling of alternative process plans in this dissertation originally arose out of the need to represent multiple recipes for the production of an inventory, we have not addressed scheduling with both inventory and activity alternatives in the same problem. Such a combination presents a number of challenging issues for future work. For example, recall that inventory bound propagation (and, indeed, the cumulative constraint propagation due to [Simonis and Cornelissens, 1995]) required that the upper and lower bounds on the inventory levels must be known. If activities which produce and consume an inventory may not necessarily exist in a final solution, the bounds on inventory levels will be much weaker, reducing the efficacy of the propagator. Other issues surrounding what commitments are valid on producers and consumers with PEX values of less than 1 and the incorporation of PEX into the inventory texture measurements also remain to be explored.

8.2.2.4 Scheduling with Multi-capacity Resources

The resources in the scheduling problems investigated in this dissertation were unary capacity resources. An interesting area for future application of texture measurement-based heuristic commitment techniques is to multi-capacity resources: resources that can execute more than one activities at a time point. While there has been work looking at the extension of propagators to multi-capacity resources [Nuijten and Aarts, 1997; Caseau and Laburthe, 1996] there has been little work that has examined heuristic techniques for such resources with the notable exception of [Cesta et al., 1998].

Two of the texture measurements proposed in this dissertation for the estimation of the probability of breakage of unary resource constraints (TriangleHeight and VarHeight) are directly applicable to multi-capacity resource constraints. Issues to be investigated include the type of heuristic commitments to be made based on the texture information and termination criteria for the multi-capacity resources.

8.2.2.5 Scheduling and Optimization

Scheduling is not simply the satisfaction of constraints but also the optimization of various cost functions stemming from sources through-out an enterprise. In this dissertation, we have focussed on heuristic techniques for the satisfaction of constraints rather than for the optimization. There are simple techniques for the application of satisfaction technology to optimization problems (*i.e.*, repeated resolving with a new constraint specifying that a solution must be found with lower cost than the best solution found so far) and there has been work done on the biasing of texture measurements by the cost information [Sadeh, 1991].

The structural approach taken in this dissertation suggests the representation of cost information as part of the constraint graph and the formulation of heuristic techniques informed by satisfaction information and cost information. Such a combination of information sources may be done based on an approach similar to [Sadeh, 1991] or based on higher level reasoning where the cost and satisfaction information are independently measured and combined at the level of the heuristic com-

mitment technique. Trade-offs among these approaches as well as their application to sophisticated optimization functions characteristic of real world problems remain to be investigated.

8.3 Conclusion

The central thesis of this dissertation is that an understanding of the structure of a problem leads to high-quality heuristic problem solving performance in constraint-directed scheduling. Exploration of this thesis has a history in the artificial intelligence literature [Simon, 1973; Fox, 1983; Fox et al., 1989; Sadeh, 1991] and this dissertation is a continuation of such investigations. Our methods for gaining an understanding of problem structure focus on *texture measurements*: algorithms that implement dynamic analyses of each search state. Texture measurements distill structural information from the constraint graph which is then used as a basis for heuristic decision making. Empirical results indicate that under a number of conditions in a variety of types of scheduling problems, superior search performance is achieved from texture measurement-based heuristic commitment techniques over simpler, less knowledge-intensive heuristics.

In particular, in this dissertation:

- We created and investigated a number of new texture measurements and texture measurement-based heuristics. The texture measurements are able to achieve a deeper understanding of the problems structure resulting in novel knowledge-based heuristic commitment techniques that outperform existing heuristics.
- We expanded the scope of problems that can be addressed, in general, by constraint-directed scheduling techniques. Specifically, we demonstrated representation and generic solutions techniques for scheduling with inventories and scheduling with alternative activities.

Appendix A Index of Important Terms

Appendix B Detailed Results for the Experiments in Chapter 4 and Chapter 5

In this appendix we provide tables of results for each of the experiments in Chapter 4 and Chapter 5. Because the chapters use the same set of experiments, with different but overlapping algorithms, we choose to combine the results into a single appendix.

B.1 Experiment 1

	Makespan Factor					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	0.95	0.75	0.25	0.20	0.00	0.00
LJRandLDS	0.90	0.20	0.05	0.00	0.00	0.00
CBASlackChron	0.50	0.20	0.05	0.00	0.00	0.00
CBASlackLDS	0.50	0.05	0.00	0.00	0.00	0.00
SumHeightChron	0.35	0.25	0.05	0.10	0.00	0.00
SumHeightLDS	0.40	0.05	0.00	0.00	0.00	0.00
JointHeightChron	0.40	0.25	0.10	0.00	0.00	0.00
JointHeightLDS	0.55	0.10	0.05	0.00	0.00	0.00
TriangleHeightChron	0.60	0.35	0.30	0.20	0.05	0.10
TriangleHeightLDS	0.65	0.15	0.00	0.00	0.00	0.00
VarHeightChron	0.40	0.15	0.10	0.05	0.00	0.00
VarHeightLDS	0.45	0.05	0.00	0.00	0.00	0.00

Table B.1. **Experiment 1: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	1172.31	902.07	417.22	241.69	47.80	1.71
LJRandLDS	1092.76	342.87	113.32	6.79	2.10	1.62
CBASlackChron	685.55	265.46	85.53	3.61	3.81	3.85
CBASlackLDS	645.43	82.21	12.60	3.93	3.62	3.69
SumHeightChron	604.37	319.34	90.25	127.28	55.35	1.83
SumHeightLDS	569.47	113.12	16.90	3.19	2.45	2.06
JointHeightChron	685.54	398.33	155.20	42.38	3.68	3.86
JointHeightLDS	702.33	157.78	71.17	5.13	3.83	4.27
TriangleHeightChron	781.20	535.39	388.35	249.67	70.90	121.96
TriangleHeightLDS	916.95	250.33	108.47	20.26	7.34	6.27
VarHeightChron	625.40	216.87	121.79	77.16	9.32	1.95
VarHeightLDS	602.16	74.11	8.20	3.78	2.66	2.24

Table B.2. **Experiment 1: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	17916.10	11279.50	4996.85	2158.20	277.95	2.90
LJRandLDS	11004.25	2357.75	431.65	7.25	0.75	0.30
CBASlackChron	8592.40	2080.30	476.85	0.10	0.00	0.00
CBASlackLDS	5253.25	124.10	3.35	0.20	0.00	0.00
SumHeightChron	5898.10	2143.95	642.00	1180.00	315.70	0.15
SumHeightLDS	4636.50	341.50	26.50	1.10	0.45	0.25
JointHeightChron	6298.70	3153.10	987.05	449.15	0.10	0.10
JointHeightLDS	3557.60	131.85	34.45	0.65	0.05	0.05
TriangleHeightChron	8298.65	5191.35	2966.60	2114.00	541.50	589.35
TriangleHeightLDS	8520.95	715.10	160.20	15.25	4.00	2.30
VarHeightChron	5623.75	1397.75	949.80	678.35	77.20	0.10
VarHeightLDS	4214.20	167.70	6.30	1.35	0.65	0.20

Table B.3. **Experiment 1: Mean Number of Backtracks.**

	Makespan Factors					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	632030.90	355228.55	138393.35	63349.60	8317.25	1478.10
LJRandLDS	884110.40	314582.25	110333.70	6897.15	2174.55	1732.55
CBASlackChron	257096.15	69079.10	18061.70	1092.75	1054.60	1030.15
CBASlackLDS	359378.95	39194.40	4118.25	1310.35	1054.60	1030.15
SumHeightChron	284642.65	91840.75	18587.15	26122.35	7765.70	908.65
SumHeightLDS	384581.70	70865.10	10954.45	1781.10	1279.35	1028.70
JointHeightChron	296081.50	152016.15	33164.95	20500.35	967.30	883.50
JointHeightLDS	297666.90	38863.75	14545.10	1422.10	985.85	938.60
TriangleHeightChron	314407.85	149825.40	79109.70	55540.70	8353.80	17199.70
TriangleHeightLDS	541786.65	127583.55	51561.25	8757.75	3098.00	2549.30
VarHeightChron	286907.80	63704.85	22499.30	15568.00	2422.25	861.80
VarHeightLDS	385622.10	41533.85	4596.95	1864.90	1211.95	983.35

Table B.4. **Experiment 1: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	35866.25	22619.95	10106.45	4461.00	695.45	143.00
LJRandLDS	48933.00	18936.10	6598.50	514.05	199.00	163.90
CBASlackChron	17376.95	4588.10	1567.15	736.95	817.80	852.50
CBASlackLDS	38241.45	8377.60	1795.65	788.30	817.80	852.50
SumHeightChron	11839.15	4371.70	1410.70	2510.95	827.20	228.10
SumHeightLDS	20501.20	6216.60	1189.50	309.35	273.25	269.60
JointHeightChron	12644.95	6388.85	2120.90	1076.00	228.25	262.90
JointHeightLDS	15836.70	3579.60	1722.75	261.95	237.65	275.55
TriangleHeightChron	16636.55	10456.75	6037.50	4352.90	1243.35	1358.10
TriangleHeightLDS	29603.70	10440.90	5226.45	1173.70	518.95	429.60
VarHeightChron	11288.65	2883.75	2026.70	1528.90	360.55	241.65
VarHeightLDS	18733.85	4192.55	610.15	354.15	296.65	268.45

Table B.5. **Experiment 1: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**
LJRandChron	5.68	7.03	8.56	9.98	10.51	10.60
LJRandLDS	5.78	7.13	8.33	9.64	10.47	10.69
CBASlackChron	7.80	23.28	49.29	66.61	76.31	81.23
CBASlackLDS	10.83	24.51	51.81	64.97	76.31	81.23
SumHeightChron	4.40	6.26	10.85	17.46	23.02	29.39
SumHeightLDS	5.02	8.84	14.15	19.68	23.85	30.40
JointHeightChron	4.36	6.12	12.78	16.34	26.07	33.66
JointHeightLDS	5.33	9.61	14.99	19.43	26.47	33.68
TriangleHeightChron	5.34	7.55	9.67	11.88	17.33	20.87
TriangleHeightLDS	5.31	8.75	10.95	14.62	18.24	21.26
VarHeightChron	4.15	6.60	12.27	19.44	24.07	30.78
VarHeightLDS	4.86	9.08	14.28	21.13	26.09	30.39

Table B.6. **Experiment 1: Mean Percentage of Heuristic Commitments.**

B.2 Experiment 2

B.2.1 Overall Results

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	0.00	0.05	0.49	0.74
LJRandLDS	0.00	0.04	0.32	0.59
CBASlackChron	0.00	0.00	0.14	0.43
CBASlackLDS	0.00	0.00	0.10	0.41
SumHeightChron	0.00	0.00	0.14	0.55
SumHeightLDS	0.00	0.00	0.14	0.45
JointHeightChron	0.00	0.00	0.19	0.54
JointHeightLDS	0.00	0.00	0.15	0.44
TriangleHeightChron	0.00	0.00	0.38	0.76
TriangleHeightLDS	0.00	0.00	0.23	0.61
VarHeightChron	0.00	0.00	0.13	0.54
VarHeightLDS	0.00	0.00	0.11	0.44

Table B.7. **Experiment 2 Overall: Fraction of Problems Timed-out.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	0.05	83.45	611.48	905.18
LJRandLDS	0.07	51.96	430.32	779.41
CBASlackChron	0.04	1.02	207.18	559.44
CBASlackLDS	0.05	1.93	174.11	542.19
SumHeightChron	0.06	0.79	233.99	693.50
SumHeightLDS	0.06	1.60	219.66	613.32
JointHeightChron	0.06	1.15	282.48	685.41
JointHeightLDS	0.07	2.24	238.60	603.96
TriangleHeightChron	0.06	1.71	502.26	927.92
TriangleHeightLDS	0.07	6.35	332.96	840.38
VarHeightChron	0.06	0.83	201.18	688.19
VarHeightLDS	0.06	1.30	213.30	587.55

Table B.8. **Experiment 2 Overall: Mean CPU Time in Seconds.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	0.71	1786.84	3423.08	1634.89
LJRandLDS	2.68	909.67	1248.08	521.65
CBASlackChron	0.33	8.91	1322.34	1160.19
CBASlackLDS	0.70	29.07	899.16	431.15
SumHeightChron	0.23	5.69	1307.50	1306.44
SumHeightLDS	0.50	17.81	914.20	517.29
JointHeightChron	0.23	6.58	1440.39	1196.31
JointHeightLDS	0.52	19.01	712.78	299.53
TriangleHeightChron	0.36	20.84	2712.12	1760.55
TriangleHeightLDS	0.85	95.05	1208.59	618.39
VarHeightChron	0.23	4.94	1081.49	1245.58
VarHeightLDS	0.54	11.04	841.72	450.99

Table B.9. **Experiment 2 Overall: Mean Number of Backtracks.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	117.71	50040.29	134340.77	83750.09
LJRandLDS	151.10	41215.21	240421.00	290963.65
CBASlackChron	84.41	759.41	52401.93	56436.91
CBASlackLDS	88.70	1539.82	72814.64	152319.51
SumHeightChron	82.52	760.41	75895.08	102212.40
SumHeightLDS	86.51	1530.21	105905.96	207047.68
JointHeightChron	82.39	794.08	83001.96	94682.91
JointHeightLDS	86.21	1640.57	86221.82	120648.86
TriangleHeightChron	84.46	1230.36	107269.03	92024.03
TriangleHeightLDS	93.01	4759.56	142179.84	247036.61
VarHeightChron	82.71	708.28	67481.51	101032.77
VarHeightLDS	87.71	1157.54	102591.03	189910.15

Table B.10. **Experiment 2 Overall: Mean Number of Commitments.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	18.36	3647.13	6964.90	3424.19
LJRandLDS	23.25	3014.11	13531.25	14658.27
CBASlackChron	25.02	259.86	3395.77	3836.01
CBASlackLDS	25.72	311.55	5435.21	15717.15
SumHeightChron	15.89	89.42	2749.25	2761.29
SumHeightLDS	16.28	127.92	4212.67	8439.11
JointHeightChron	16.35	103.39	3039.41	2584.19
JointHeightLDS	16.77	143.02	3600.96	5153.57
TriangleHeightChron	14.08	100.57	5526.99	3634.79
TriangleHeightLDS	14.84	295.44	6891.36	11648.28
VarHeightChron	15.87	91.41	2305.49	2652.26
VarHeightLDS	16.34	112.29	3799.50	7623.74

Table B.11. **Experiment 2 Overall: Mean Number of Heuristic Commitments.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
LJRandChron	15.46	10.26	6.79	4.91
LJRandLDS	15.75	10.59	7.17	5.33
CBASlackChron	31.65	44.66	38.72	26.44
CBASlackLDS	31.89	43.94	38.06	27.45
SumHeightChron	20.62	15.88	7.93	3.65
SumHeightLDS	20.71	16.37	9.09	4.97
JointHeightChron	21.22	18.31	9.54	4.59
JointHeightLDS	21.37	18.21	10.54	6.07
TriangleHeightChron	17.95	11.57	6.39	4.21
TriangleHeightLDS	17.87	11.58	7.01	4.66
VarHeightChron	20.57	16.68	8.49	3.89
VarHeightLDS	20.67	16.79	9.31	5.22

Table B.12. **Experiment 2 Overall: Mean Percentage of Heuristic Commitments.**

B.2.2 5✕5 Results

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LJRandLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JointHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JointHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TriangleHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TriangleHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table B.13. **Experiment 2, 5✕5 Problems: Fraction of Problems Timed-out.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	0.04	0.07	0.05	0.05	0.05	0.05	0.05
LJRandLDS	0.05	0.16	0.06	0.06	0.06	0.06	0.06
CBASlackChron	0.03	0.04	0.04	0.04	0.05	0.05	0.05
CBASlackLDS	0.04	0.05	0.05	0.04	0.05	0.04	0.05
SumHeightChron	0.04	0.06	0.06	0.06	0.06	0.06	0.06
SumHeightLDS	0.05	0.06	0.05	0.06	0.06	0.06	0.06
JointHeightChron	0.04	0.06	0.06	0.07	0.07	0.07	0.08
JointHeightLDS	0.05	0.07	0.06	0.06	0.07	0.07	0.08
TriangleHeightChron	0.05	0.06	0.06	0.06	0.07	0.07	0.07
TriangleHeightLDS	0.05	0.08	0.06	0.07	0.07	0.07	0.08
VarHeightChron	0.05	0.06	0.06	0.07	0.07	0.07	0.07
VarHeightLDS	0.05	0.07	0.06	0.07	0.06	0.07	0.07

Table B.14. **Experiment 2, 5✕5 Problems: Mean CPU Time in Seconds.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	1.25	2.80	0.45	0.15	0.20	0.10	0.05
LJRandLDS	3.30	14.30	0.65	0.15	0.20	0.10	0.05
CBASlackChron	0.90	1.25	0.15	0.00	0.00	0.00	0.00
CBASlackLDS	2.15	2.55	0.20	0.00	0.00	0.00	0.00
SumHeightChron	0.85	0.65	0.00	0.10	0.00	0.00	0.00
SumHeightLDS	2.05	1.30	0.00	0.15	0.00	0.00	0.00
JointHeightChron	0.80	0.75	0.00	0.05	0.00	0.00	0.00
JointHeightLDS	2.00	1.60	0.00	0.05	0.00	0.00	0.00
TriangleHeightChron	0.95	1.10	0.15	0.20	0.10	0.00	0.00
TriangleHeightLDS	2.55	2.95	0.15	0.20	0.10	0.00	0.00
VarHeightChron	0.85	0.65	0.05	0.05	0.00	0.00	0.00
VarHeightLDS	2.05	1.55	0.10	0.10	0.00	0.00	0.00

Table B.15. **Experiment 2, 5✕5 Problems: Mean Number of Backtracks.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	96.80	146.20	121.15	114.90	114.85	114.75	115.35
LJRandLDS	112.10	325.00	134.65	121.55	126.00	119.85	118.55
CBASlackChron	87.30	102.50	91.75	83.00	77.55	75.20	73.60
CBASlackLDS	98.65	118.55	94.35	83.00	77.55	75.20	73.60
SumHeightChron	86.35	98.95	89.00	84.00	77.70	72.95	68.70
SumHeightLDS	96.40	112.00	89.05	88.80	77.70	72.95	68.70
JointHeightChron	85.75	98.45	91.05	83.95	75.90	72.75	68.90
JointHeightLDS	95.60	114.50	91.05	84.80	75.90	72.75	68.90
TriangleHeightChron	89.40	101.70	89.35	85.80	79.05	75.10	70.80
TriangleHeightLDS	105.55	137.70	90.30	90.05	81.55	75.10	70.80
VarHeightChron	85.90	99.15	89.00	84.55	78.85	73.05	68.45
VarHeightLDS	95.15	119.55	91.25	87.70	78.85	73.05	68.45

Table B.16. **Experiment 2, 5×5 Problems: Mean Number of Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	8.90	20.30	18.80	18.55	19.95	20.40	21.60
LJRandLDS	12.10	45.60	20.10	19.80	22.00	21.00	22.15
CBASlackChron	2.95	10.95	18.20	26.75	34.10	39.50	42.70
CBASlackLDS	4.30	14.40	18.30	26.75	34.10	39.50	42.70
SumHeightChron	2.90	8.75	12.10	17.90	20.40	23.60	25.60
SumHeightLDS	4.10	10.10	12.10	18.05	20.40	23.60	25.60
JointHeightChron	2.95	9.25	12.55	17.60	20.65	24.50	26.95
JointHeightLDS	4.15	11.05	12.55	17.55	20.65	24.50	26.95
TriangleHeightChron	3.00	8.70	10.35	13.35	18.15	21.30	23.70
TriangleHeightLDS	4.90	12.70	10.20	13.05	18.00	21.30	23.70
VarHeightChron	2.90	8.55	12.05	18.65	19.65	23.75	25.55
VarHeightLDS	4.10	10.55	11.95	18.85	19.65	23.75	25.55

Table B.17. **Experiment 2, 5×5 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	8.54	14.26	15.46	16.08	17.33	17.80	18.73
LJRandLDS	10.56	14.96	15.15	16.07	17.30	17.54	18.66
CBASlackChron	3.41	10.74	19.97	32.71	44.09	52.61	58.01
CBASlackLDS	4.40	11.70	19.72	32.71	44.09	52.61	58.01
SumHeightChron	3.36	8.91	13.75	21.72	26.66	32.52	37.42
SumHeightLDS	4.36	9.05	13.74	21.25	26.66	32.52	37.42
JointHeightChron	3.43	9.51	13.93	21.19	27.45	33.82	39.23
JointHeightLDS	4.44	9.72	13.93	21.00	27.45	33.82	39.23
TriangleHeightChron	3.38	8.60	11.76	15.98	23.27	28.64	34.03
TriangleHeightLDS	4.54	8.70	11.51	15.13	22.54	28.64	34.03
VarHeightChron	3.37	8.80	13.74	22.36	25.38	32.78	37.57
VarHeightLDS	4.42	9.01	13.37	22.13	25.38	32.78	37.57

Table B.18. **Experiment 2, 5×5 Problems: Mean Percentage of Heuristic Commitments.**

B.2.3 10×10 Results

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	0.00	0.35	0.00	0.00	0.00	0.00	0.00
LJRandLDS	0.05	0.20	0.00	0.00	0.00	0.00	0.00
CBASlackChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JointHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JointHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TriangleHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TriangleHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightChron	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightLDS	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table B.19. **Experiment 2, 10×10 Problems: Fraction of Problems Timed-out.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	10.37	489.29	34.63	47.06	1.77	0.54	0.52
LJRandLDS	80.77	277.60	2.55	0.88	0.82	0.60	0.53
CBASlackChron	0.47	2.45	0.92	0.80	0.83	0.80	0.89
CBASlackLDS	0.94	8.22	1.01	0.78	0.83	0.81	0.90
SumHeightChron	0.35	2.06	0.68	0.61	0.63	0.60	0.61
SumHeightLDS	0.74	6.58	1.27	0.69	0.67	0.62	0.62
JointHeightChron	0.40	2.60	1.01	0.94	0.95	1.02	1.11
JointHeightLDS	0.89	8.55	2.02	1.10	0.95	1.05	1.11
TriangleHeightChron	0.43	4.40	2.86	1.87	0.78	0.79	0.85
TriangleHeightLDS	1.18	35.61	2.52	1.75	1.20	1.11	1.06
VarHeightChron	0.38	1.75	0.73	0.80	0.69	0.69	0.74
VarHeightLDS	0.76	4.37	1.10	0.73	0.70	0.69	0.76

Table B.20. **Experiment 2, 10×10 Problems: Mean CPU Time in Seconds.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	254.10	10509.50	732.40	983.30	28.20	0.35	0.05
LJRandLDS	1905.40	4447.55	12.50	1.20	0.80	0.20	0.05
CBASlackChron	9.50	47.35	5.45	0.05	0.00	0.00	0.00
CBASlackLDS	22.75	176.30	4.40	0.05	0.00	0.00	0.00
SumHeightChron	4.55	31.30	2.45	0.80	0.60	0.10	0.05
SumHeightLDS	12.55	104.55	6.70	0.45	0.30	0.05	0.05
JointHeightChron	4.90	36.00	3.95	1.15	0.00	0.05	0.00
JointHeightLDS	13.30	110.10	9.05	0.55	0.00	0.05	0.00
TriangleHeightChron	5.75	73.50	40.95	22.85	1.30	0.95	0.55
TriangleHeightLDS	21.10	613.40	24.05	3.90	1.35	1.05	0.50
VarHeightChron	4.55	24.75	2.45	2.75	0.05	0.00	0.05
VarHeightLDS	11.65	61.55	3.75	0.25	0.05	0.00	0.05

Table B.21. **Experiment 2, 10×10 Problems: Mean Number of Backtracks.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	4941.80	293603.90	20090.50	29010.75	1181.90	728.60	724.60
LJRandLDS	36776.15	244851.00	3096.90	1129.55	1083.00	820.55	749.30
CBASlackChron	520.20	1898.65	740.85	565.05	542.85	527.60	520.65
CBASlackLDS	821.80	6751.80	1033.10	580.95	542.85	527.60	520.65
SumHeightChron	477.00	2052.00	720.40	604.60	536.30	480.10	452.50
SumHeightLDS	805.55	6382.55	1341.15	668.15	563.65	488.70	461.75
JointHeightChron	484.90	2239.65	792.30	601.25	515.30	478.00	447.15
JointHeightLDS	810.60	6924.30	1578.45	718.95	515.30	489.25	447.15
TriangleHeightChron	506.60	3310.75	2145.15	1051.00	560.35	542.10	496.60
TriangleHeightLDS	1070.75	26297.50	2276.00	1407.15	883.10	756.30	626.15
VarHeightChron	477.30	1725.35	691.05	633.60	511.40	475.85	443.40
VarHeightLDS	779.25	4181.05	1079.70	603.15	522.75	475.85	461.00

Table B.22. **Experiment 2, 10×10 Problems: Mean Number of Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	513.75	21078.20	1551.30	2054.90	146.60	91.80	93.35
LJRandLDS	4058.60	16295.80	286.35	127.75	129.20	104.70	96.40
CBASlackChron	21.50	176.80	219.15	287.85	341.35	378.65	393.75
CBASlackLDS	53.10	501.35	227.25	285.40	341.35	378.65	393.75
SumHeightChron	9.85	89.75	71.65	81.85	110.80	123.90	138.15
SumHeightLDS	25.20	270.75	113.65	95.10	121.00	129.45	140.30
JointHeightChron	10.65	103.05	78.30	102.55	121.20	147.05	160.95
JointHeightLDS	26.95	300.45	130.20	110.40	121.20	151.00	160.95
TriangleHeightChron	12.20	168.10	130.05	111.05	84.50	94.15	103.95
TriangleHeightLDS	41.70	1343.90	142.20	151.45	128.65	129.95	130.20
VarHeightChron	10.00	78.80	69.65	98.90	113.90	125.10	143.50
VarHeightLDS	24.50	186.00	92.55	99.80	115.15	125.10	142.95

Table B.23. **Experiment 2, 10×10 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	4.95	7.88	9.96	11.25	12.30	12.61	12.89
LJRandLDS	6.10	7.95	10.45	11.84	12.19	12.71	12.89
CBASlackChron	3.35	13.49	33.83	51.40	63.10	71.82	75.64
CBASlackLDS	5.04	10.58	30.85	50.54	63.10	71.82	75.64
SumHeightChron	1.72	5.03	11.14	14.55	21.44	26.34	30.92
SumHeightLDS	2.57	5.65	11.08	15.14	22.29	26.85	31.01
JointHeightChron	1.91	5.80	11.16	18.03	24.01	31.03	36.21
JointHeightLDS	2.90	5.59	10.37	17.19	24.01	31.21	36.21
TriangleHeightChron	1.93	5.03	7.54	11.85	15.44	17.86	21.33
TriangleHeightLDS	2.84	4.77	6.92	11.63	15.39	18.14	21.39
VarHeightChron	1.72	5.13	10.85	17.24	22.69	26.56	32.59
VarHeightLDS	2.50	5.41	10.71	17.58	22.68	26.56	32.10

Table B.24. **Experiment 2, 10×10 Problems: Mean Percentage of Heuristic Commitments.**

B.2.4 15×15 Results

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	0.20	1.00	1.00	0.70	0.40	0.15	0.00
LJRandLDS	0.25	0.95	0.70	0.35	0.00	0.00	0.00
CBASlackChron	0.05	0.45	0.40	0.10	0.00	0.00	0.00
CBASlackLDS	0.05	0.40	0.25	0.00	0.00	0.00	0.00
SumHeightChron	0.00	0.25	0.50	0.10	0.10	0.05	0.00
SumHeightLDS	0.05	0.55	0.35	0.00	0.00	0.00	0.00
JointHeightChron	0.05	0.30	0.60	0.25	0.05	0.05	0.00
JointHeightLDS	0.05	0.50	0.50	0.00	0.00	0.00	0.00
TriangleHeightChron	0.05	0.70	0.70	0.50	0.35	0.15	0.20
TriangleHeightLDS	0.05	0.90	0.60	0.05	0.00	0.00	0.00
VarHeightChron	0.00	0.25	0.50	0.10	0.00	0.05	0.00
VarHeightLDS	0.00	0.45	0.30	0.00	0.00	0.00	0.00

Table B.25. **Experiment 2, 15×15 Problems: Fraction of Problems Timed-out.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	270.93	1200.09	1200.16	894.84	511.63	199.91	2.80
LJRandLDS	417.52	1164.30	892.92	487.20	38.47	8.07	3.73
CBASlackChron	71.21	619.05	567.49	171.62	6.67	6.82	7.43
CBASlackLDS	108.40	661.52	408.49	18.66	7.28	6.98	7.41
SumHeightChron	20.86	567.25	630.91	208.04	142.10	63.76	5.03
SumHeightLDS	69.65	787.14	620.32	44.45	7.34	4.71	3.99
JointHeightChron	70.33	607.58	779.60	336.20	68.80	74.70	40.13
JointHeightLDS	124.04	796.09	667.78	56.69	9.84	8.87	6.90
TriangleHeightChron	63.47	922.59	864.20	668.52	439.84	275.39	281.81
TriangleHeightLDS	78.71	1142.03	805.84	236.50	35.45	15.99	16.20
VarHeightChron	16.01	435.71	664.70	205.45	17.50	64.35	4.53
VarHeightLDS	96.73	734.73	587.24	57.20	7.91	4.45	4.84

Table B.26. **Experiment 2, 15X15 Problems: Mean CPU Time in Seconds.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	1809.15	7542.30	6514.35	4692.35	2466.05	936.40	0.95
LJRandLDS	2514.25	3673.60	1795.65	716.85	32.80	2.85	0.55
CBASlackChron	490.40	4071.45	3706.95	987.15	0.35	0.05	0.00
CBASlackLDS	765.10	3820.45	1694.00	14.05	0.40	0.15	0.00
SumHeightChron	134.50	3218.90	3571.75	1185.50	647.95	384.40	9.50
SumHeightLDS	437.70	3804.05	2081.60	71.60	3.05	0.90	0.50
JointHeightChron	398.30	3260.65	4065.75	1591.15	308.20	321.35	137.35
JointHeightLDS	554.40	2895.90	1491.15	45.60	1.45	0.85	0.10
TriangleHeightChron	347.60	5365.20	4740.50	3580.55	2303.45	1368.45	1279.10
TriangleHeightLDS	380.80	5114.55	2483.20	436.30	31.25	7.75	6.30
VarHeightChron	95.70	2385.85	3612.25	1059.70	66.25	347.60	3.05
VarHeightLDS	558.70	3389.90	1868.05	71.20	3.20	0.55	0.45

Table B.27. **Experiment 2, 15X15 Problems: Mean Number of Backtracks.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	75041.85	322515.35	269275.90	169136.60	76866.90	25333.80	2215.00
LJRandLDS	156407.45	644276.45	543594.90	302841.85	26494.15	6238.30	3093.90
CBASlackChron	16979.60	158992.80	147837.00	37756.15	1806.40	1749.40	1692.15
CBASlackLDS	33906.55	265835.05	193566.80	10558.75	2254.45	1888.75	1692.15
SumHeightChron	7344.05	207740.40	207824.30	61161.35	29110.40	15860.15	2224.90
SumHeightLDS	24893.75	359935.55	323037.05	24928.70	4060.80	2439.60	2046.25
JointHeightChron	20071.15	194098.20	241894.60	100282.90	9246.50	11957.00	3463.35
JointHeightLDS	38984.05	282683.10	254651.35	20003.15	3082.25	2447.05	1701.80
TriangleHeightChron	18467.85	253020.30	187792.95	130776.55	82505.30	44632.80	33687.45
TriangleHeightLDS	28752.65	468912.35	360811.60	106484.90	16655.65	7266.20	6375.55
VarHeightChron	6331.30	163033.65	227737.15	51691.80	6320.95	15572.15	1683.60
VarHeightLDS	35742.65	342046.80	302199.25	29862.00	4171.40	2114.45	2000.65

Table B.28. **Experiment 2, 15×15 Problems: Mean Number of Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	3624.40	15127.15	13104.30	9506.65	5104.25	2072.40	215.15
LJRandLDS	9078.00	32128.20	30888.25	19809.95	1980.35	543.80	290.20
CBASlackChron	985.50	8379.05	8008.00	2804.65	1052.10	1211.05	1330.05
CBASlackLDS	2275.25	16147.20	14573.10	1416.40	1090.45	1214.05	1330.05
SumHeightChron	267.00	6465.90	7230.70	2513.00	1497.50	1009.60	261.05
SumHeightLDS	1000.20	12221.55	13564.80	1613.90	431.45	345.15	311.65
JointHeightChron	795.80	6552.20	8230.40	3333.10	843.05	915.60	605.75
JointHeightLDS	1510.75	10480.50	10776.80	1344.45	366.40	380.55	347.25
TriangleHeightChron	694.45	10768.40	9566.30	7270.60	4745.10	2910.40	2733.65
TriangleHeightLDS	1212.60	19112.20	18088.75	7035.85	1372.90	699.10	718.15
VarHeightChron	189.40	4801.00	7312.80	2273.35	324.60	944.15	293.15
VarHeightLDS	1202.00	10820.15	11540.90	1971.40	400.10	311.15	350.80

Table B.29. **Experiment 2, 15×15 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	3.53	5.02	5.27	6.70	8.13	9.15	9.72
LJRandLDS	4.61	5.01	6.07	7.42	8.45	9.13	9.53
CBASlackChron	3.11	5.08	19.57	36.27	58.75	69.51	78.72
CBASlackLDS	4.52	6.55	19.59	33.65	55.43	67.96	78.72
SumHeightChron	1.30	3.10	4.04	6.70	10.55	14.80	15.05
SumHeightLDS	1.96	3.51	5.41	8.55	12.42	15.61	16.16
JointHeightChron	1.58	3.29	5.04	6.70	12.65	15.95	21.60
JointHeightLDS	2.30	4.01	5.85	8.93	13.63	17.36	21.68
TriangleHeightChron	1.36	4.30	5.36	6.01	7.74	9.65	10.29
TriangleHeightLDS	2.00	4.14	5.53	6.86	8.85	10.19	11.47
VarHeightChron	1.33	2.86	3.87	7.11	10.16	15.49	18.62
VarHeightLDS	1.92	3.23	5.02	8.55	11.52	16.07	18.85

Table B.30. **Experiment 2, 15×15 Problems: Mean Percentage of Heuristic Commitments.**

B.2.5 20×20 Results

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	0.25	1.00	1.00	1.00	0.90	0.80	0.25
LJRandLDS	0.30	1.00	1.00	1.00	0.75	0.10	0.00
CBASlackChron	0.00	0.80	0.95	0.85	0.35	0.05	0.00
CBASlackLDS	0.05	1.00	1.00	0.75	0.10	0.00	0.00
SumHeightChron	0.00	0.95	1.00	0.90	0.65	0.20	0.15
SumHeightLDS	0.00	0.95	1.00	0.90	0.30	0.00	0.00
JointHeightChron	0.05	0.95	1.00	1.00	0.50	0.20	0.05
JointHeightLDS	0.05	1.00	1.00	0.80	0.20	0.00	0.00
TriangleHeightChron	0.05	1.00	1.00	1.00	0.90	0.80	0.60
TriangleHeightLDS	0.05	1.00	1.00	1.00	0.70	0.45	0.10
VarHeightChron	0.00	0.95	1.00	1.00	0.45	0.20	0.20
VarHeightLDS	0.00	0.95	1.00	0.70	0.35	0.05	0.00

Table B.31. **Experiment 2, 20×20 Problems: Fraction of Problems Timed-out.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	356.16	1200.23	1200.33	1200.36	1109.01	962.37	307.82
LJRandLDS	531.95	1200.34	1200.34	1200.31	971.20	300.69	51.06
CBASlackChron	23.39	1071.23	1162.51	1043.70	487.44	91.53	36.25
CBASlackLDS	75.06	1200.18	1200.44	977.04	259.68	46.75	36.16
SumHeightChron	9.61	1146.96	1200.24	1104.60	817.95	321.06	254.09
SumHeightLDS	39.64	1180.06	1200.27	1130.60	587.41	112.71	42.54
JointHeightChron	74.99	1165.00	1200.28	1200.28	680.88	383.25	93.21
JointHeightLDS	135.31	1200.33	1200.56	1003.28	496.24	151.69	40.31
TriangleHeightChron	69.17	1200.25	1200.20	1200.23	1116.39	980.19	728.99
TriangleHeightLDS	108.30	1200.32	1200.38	1200.21	991.67	764.99	416.79
VarHeightChron	6.34	1145.16	1200.25	1200.25	694.20	313.76	257.40
VarHeightLDS	21.21	1164.07	1200.31	998.08	545.31	147.82	36.04

Table B.32. **Experiment 2, 20×20 Problems: Mean CPU Time in Seconds.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	936.55	2713.75	2328.50	2023.20	1719.70	1327.65	394.85
LJRandLDS	1299.40	1018.35	570.50	405.55	285.70	65.60	6.45
CBASlackChron	70.65	2805.95	2496.90	1906.90	750.75	90.20	0.00
CBASlackLDS	225.05	2086.65	482.95	184.00	37.10	2.30	0.00
SumHeightChron	20.85	2558.85	2403.00	1959.15	1331.80	473.20	398.20
SumHeightLDS	93.95	1775.95	948.65	551.35	221.65	24.15	5.35
JointHeightChron	158.45	2298.05	2194.60	2049.35	1060.45	527.40	85.90
JointHeightLDS	227.05	1073.35	453.15	252.00	76.60	13.65	0.90
TriangleHeightChron	166.85	2668.65	2532.20	2241.75	1908.30	1680.35	1125.75
TriangleHeightLDS	245.85	1684.35	984.30	640.90	416.95	250.35	106.05
VarHeightChron	12.10	2466.35	2303.70	2104.30	1044.40	464.35	323.85
VarHeightLDS	47.50	1636.85	824.00	440.45	174.35	30.65	3.10

Table B.33. **Experiment 2, 20×20 Problems: Mean Number of Backtracks.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	51274.10	169415.15	121536.00	101663.95	73483.10	52231.15	16647.15
LJRandLDS	87265.00	433612.45	473296.15	480260.30	407814.20	129362.25	25135.20
CBASlackChron	3503.70	130323.70	118853.15	91817.70	38772.95	7800.95	3986.20
CBASlackLDS	7873.10	299797.90	369620.70	296847.95	78554.10	9556.60	3986.20
SumHeightChron	3232.95	187801.40	179760.85	162333.80	109716.10	40752.00	31889.70
SumHeightLDS	9781.40	345452.65	428115.30	400692.80	211395.90	39058.60	14837.10
JointHeightChron	12140.95	194896.75	178460.45	153327.50	77036.00	38187.85	8730.85
JointHeightLDS	21036.05	248643.35	247893.90	198186.10	95463.05	27106.05	6213.50
TriangleHeightChron	9767.65	161200.45	120778.45	121404.45	105888.25	74017.00	51111.95
TriangleHeightLDS	17087.00	328627.60	365528.30	372444.55	305347.10	226552.70	113669.00
VarHeightChron	2639.80	197394.80	188131.45	180858.90	85846.30	31420.70	20937.45
VarHeightLDS	5953.00	343588.30	406770.25	336066.30	179637.25	46557.40	10798.55

Table B.34. **Experiment 2, 20×20 Problems: Mean Number of Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	1876.55	5482.20	4752.80	4180.85	3630.20	2911.30	1135.45
LJRandLDS	3689.60	16528.70	22154.35	26042.90	24193.15	8331.20	1668.00
CBASlackChron	139.30	6123.10	6061.15	5264.00	3562.75	2733.90	2967.90
CBASlackLDS	513.40	18188.10	38665.10	36480.10	10087.80	3117.65	2967.90
SumHeightChron	39.70	5161.95	4894.35	4038.40	2848.85	1216.60	1129.15
SumHeightLDS	198.75	9573.10	15322.55	18633.85	11364.40	2702.60	1278.50
JointHeightChron	316.85	4636.50	4480.40	4229.40	2357.20	1423.35	645.60
JointHeightLDS	597.45	7177.45	10062.75	9790.25	5701.50	2032.35	713.25
TriangleHeightChron	332.65	5385.30	5149.30	4599.30	3951.55	3532.65	2492.80
TriangleHeightLDS	618.90	10086.90	14850.10	18087.40	16236.05	13898.90	7759.70
VarHeightChron	22.20	4975.85	4687.05	4336.35	2314.90	1228.90	1000.60
VarHeightLDS	97.40	8720.15	14683.15	15670.05	9883.65	3331.50	980.25

Table B.35. **Experiment 2, 20×20 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor						
Algorithm	**1.0**	**1.05**	**1.1**	**1.15**	**1.2**	**1.25**	**1.3**
LJRandChron	2.76	3.33	4.16	4.25	5.63	6.66	7.55
LJRandLDS	3.44	3.82	4.68	5.42	6.03	6.69	7.21
CBASlackChron	1.79	4.68	5.20	9.19	30.95	58.70	74.54
CBASlackLDS	2.91	5.83	10.49	14.62	25.78	58.01	74.54
SumHeightChron	0.83	2.76	3.03	2.65	3.38	5.39	7.49
SumHeightLDS	1.35	2.74	3.58	4.73	6.01	7.44	8.94
JointHeightChron	1.03	2.40	2.57	2.91	4.65	7.12	11.45
JointHeightLDS	1.56	2.90	4.09	5.31	6.80	9.56	12.26
TriangleHeightChron	0.77	3.62	4.68	4.14	4.40	5.86	5.98
TriangleHeightLDS	1.47	3.06	4.08	4.88	5.40	6.21	7.49
VarHeightChron	0.55	2.55	2.69	2.75	3.79	6.48	8.42
VarHeightLDS	0.93	2.49	3.62	4.81	6.54	8.18	9.99

Table B.36. **Experiment 2, 20×20 Problems: Mean Percentage of Heuristic Commitments.**

B.3 Experiment 3

B.3.1 10×10 Results

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	0.00	0.05	0.40	0.75	0.75	0.30
LJRandLDS	0.00	0.00	0.05	0.60	0.90	0.50
CBASlackChron	0.00	0.00	0.05	0.20	0.20	0.00
CBASlackLDS	0.00	0.00	0.00	0.25	0.40	0.25
SumHeightChron	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightLDS	0.00	0.00	0.00	0.00	0.05	0.05
JointHeightChron	0.00	0.00	0.00	0.00	0.10	0.00
JointHeightLDS	0.00	0.00	0.00	0.15	0.30	0.10
TriangleHeightChron	0.00	0.00	0.00	0.40	0.40	0.05
TriangleHeightLDS	0.00	0.00	0.00	0.30	0.75	0.50
VarHeightChron	0.00	0.00	0.05	0.15	0.30	0.05
VarHeightLDS	0.00	0.00	0.00	0.30	0.45	0.25

Table B.37. **Experiment 3, 10×10 Problems: Fraction of Problems Timed-out.**

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	1.77	140.66	566.15	982.47	976.22	421.77
LJRandLDS	0.82	4.37	109.15	810.72	1106.72	710.95
CBASlackChron	0.83	4.14	67.26	390.57	418.97	109.48
CBASlackLDS	0.83	1.66	14.99	407.83	694.60	454.51
SumHeightChron	0.63	1.06	5.19	69.81	52.75	14.65
SumHeightLDS	0.67	1.49	7.16	161.93	266.06	99.14
JointHeightChron	0.95	1.47	17.36	185.42	321.17	50.68
JointHeightLDS	0.95	2.48	12.48	298.94	585.32	276.26
TriangleHeightChron	0.78	3.70	36.89	588.04	760.16	242.40
TriangleHeightLDS	1.20	4.26	24.06	529.08	1006.54	699.60
VarHeightChron	0.69	2.17	89.09	382.99	529.98	138.43
VarHeightLDS	0.70	1.84	7.93	456.30	796.65	383.37

Table B.38. **Experiment 3, 10×10 Problems: Mean CPU Time in Seconds.**

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	28.20	2547.15	8478.00	12868.70	10818.20	4418.40
LJRandLDS	0.80	18.45	784.60	7429.45	9900.50	6695.10
CBASlackChron	0.00	58.50	979.35	5621.70	5988.70	1445.40
CBASlackLDS	0.00	4.60	154.15	5707.75	9225.20	5723.40
SumHeightChron	0.60	6.20	65.35	970.55	678.95	170.25
SumHeightLDS	0.30	4.30	43.00	1904.00	3166.05	1063.25
JointHeightChron	0.00	6.55	217.40	2198.95	3562.00	543.60
JointHeightLDS	0.00	4.90	54.75	2734.35	5297.35	2483.80
TriangleHeightChron	1.30	46.60	494.45	7415.40	8292.30	2506.90
TriangleHeightLDS	1.35	19.30	157.30	4923.75	9254.25	6525.75
VarHeightChron	0.05	23.45	1324.40	5091.35	6275.00	1477.55
VarHeightLDS	0.05	4.75	40.05	5000.05	8068.20	3692.45

Table B.39. **Experiment 3, 10×10 Problems: Mean Number of Backtracks.**

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	1181.90	60063.60	358707.85	725282.40	818502.10	360083.00
LJRandLDS	1083.00	5311.55	119602.95	852255.25	1155480.60	690050.20
CBASlackChron	542.85	1860.10	20543.55	189578.40	221784.20	58515.70
CBASlackLDS	542.85	1395.35	12858.95	260329.70	411942.15	244692.40
SumHeightChron	536.30	995.75	4225.30	57178.25	41906.60	11613.45
SumHeightLDS	563.65	1531.20	6748.30	126720.15	200946.05	74231.35
JointHeightChron	515.30	966.05	9421.65	119322.75	203894.25	33353.50
JointHeightLDS	515.30	1635.75	8107.55	179440.45	350824.40	161005.20
TriangleHeightChron	560.35	2683.10	22244.95	402450.70	535954.55	189788.00
TriangleHeightLDS	883.10	3845.75	21476.15	443670.30	851184.35	561926.80
VarHeightChron	511.40	1445.25	28997.40	221487.85	365213.70	111845.75
VarHeightLDS	522.75	1770.30	7168.75	351584.75	672579.60	329784.95

Table B.40. **Experiment 3, 10×10 Problems: Mean Number of Commitments.**

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	146.60	5175.65	17010.35	25763.20	21645.20	8837.95
LJRandLDS	129.20	404.60	5421.45	29833.90	31488.95	16506.95
CBASlackChron	341.35	366.15	2164.15	11353.10	12016.55	2888.80
CBASlackLDS	341.35	321.00	1171.40	15443.25	23368.80	12826.75
SumHeightChron	110.80	80.30	193.40	1977.75	1364.25	338.50
SumHeightLDS	121.00	120.25	359.60	4826.45	7042.45	2283.85
JointHeightChron	121.20	89.95	502.75	4436.60	7129.50	1085.20
JointHeightLDS	121.20	149.25	504.10	7254.10	12687.15	5465.55
TriangleHeightChron	84.50	147.35	1033.85	14856.70	16592.20	5012.45
TriangleHeightLDS	128.65	234.90	946.40	15915.50	23785.00	14987.35
VarHeightChron	113.90	112.95	2702.15	10211.35	12562.00	2954.05
VarHeightLDS	115.15	134.80	426.65	13263.25	20230.85	8408.05

Table B.41. **Experiment 3, 10×10 Problems: Mean Number of Heuristic Commitments.**

	Number of Bottlenecks					
Algorithm	**0**	**2**	**4**	**6**	**8**	**10**
LJRandChron	12.30	8.73	5.30	3.63	2.60	2.05
LJRandLDS	12.19	8.16	5.31	3.65	2.66	2.07
CBASlackChron	63.10	31.06	16.07	7.24	5.08	4.04
CBASlackLDS	63.10	30.03	15.00	7.03	5.72	4.38
SumHeightChron	21.44	9.06	6.40	3.99	3.12	2.00
SumHeightLDS	22.29	9.01	6.29	4.03	3.39	2.28
JointHeightChron	24.01	9.99	6.01	3.95	3.21	2.20
JointHeightLDS	24.01	10.15	6.93	4.66	3.60	2.44
TriangleHeightChron	15.44	7.13	5.63	3.86	3.06	2.01
TriangleHeightLDS	15.39	7.05	5.12	3.61	2.88	2.11
VarHeightChron	22.69	8.69	6.14	4.10	3.37	2.02
VarHeightLDS	22.68	8.41	6.42	3.94	3.02	2.19

Table B.42. **Experiment 3, 10✕10 Problems: Mean Percentage of Heuristic Commitments.**

B.3.2 15✕15 Results

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	0.90	1.00	1.00	1.00	1.00	0.90	0.55
LJRandLDS	0.50	0.95	1.00	1.00	1.00	0.90	0.75
CBASlackChron	0.35	0.95	1.00	1.00	0.90	0.60	0.35
CBASlackLDS	0.00	0.65	1.00	1.00	1.00	0.70	0.60
SumHeightChron	0.25	0.50	0.90	0.95	0.65	0.35	0.15
SumHeightLDS	0.00	0.40	0.90	0.95	0.85	0.55	0.35
JointHeightChron	0.35	0.95	1.00	1.00	0.85	0.60	0.45
JointHeightLDS	0.00	0.55	1.00	1.00	0.90	0.85	0.65
TriangleHeightChron	0.70	0.95	1.00	1.00	0.85	0.70	0.45
TriangleHeightLDS	0.25	0.70	1.00	1.00	0.95	0.75	0.60
VarHeightChron	0.65	0.90	1.00	1.00	0.85	0.60	0.35
VarHeightLDS	0.10	0.60	0.90	1.00	1.00	0.80	0.65

Table B.43. **Experiment 3, 15✕15 Problems: Fraction of Problems Timed-out.**

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	1084.03	1200.15	1200.14	1200.08	1200.08	1080.21	723.41
LJRandLDS	732.11	1150.32	1200.16	1200.18	1200.14	1080.40	907.94
CBASlackChron	529.94	1172.72	1200.11	1200.08	1155.93	844.02	528.71
CBASlackLDS	108.55	974.89	1200.15	1200.10	1200.12	874.11	728.58
SumHeightChron	323.60	867.76	1109.13	1145.69	888.05	550.07	323.51
SumHeightLDS	92.30	690.64	1148.73	1197.91	1079.38	776.67	532.07
JointHeightChron	509.57	1140.61	1200.18	1200.06	1055.29	880.94	625.03
JointHeightLDS	95.87	858.71	1200.18	1200.18	1150.65	1039.56	835.70
TriangleHeightChron	860.39	1141.43	1200.12	1200.12	1115.70	912.94	650.05
TriangleHeightLDS	495.33	996.21	1200.24	1200.16	1152.45	986.20	798.05
VarHeightChron	804.58	1091.03	1200.12	1200.10	1126.58	922.56	605.40
VarHeightLDS	299.28	876.35	1138.85	1200.17	1200.12	968.48	837.33

Table B.44. **Experiment 3, 15×15 Problems: Mean CPU Time in Seconds.**

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	5340.80	5938.95	5823.70	5654.35	5439.15	4595.25	2873.20
LJRandLDS	1028.20	1906.10	2123.55	2687.65	3154.65	3022.55	2734.40
CBASlackChron	2613.90	5902.05	6435.65	6721.50	6192.20	4453.35	2678.95
CBASlackLDS	81.15	2252.60	3713.35	5279.45	5578.00	4101.15	3454.15
SumHeightChron	1653.00	4366.80	5701.55	5578.55	4331.15	2534.75	1451.85
SumHeightLDS	161.30	1909.10	3474.55	4462.90	4244.30	3142.90	2163.75
JointHeightChron	2386.70	5416.15	5561.45	5502.35	4523.15	3472.60	2399.25
JointHeightLDS	74.05	1206.70	1847.30	2367.80	2570.60	2603.65	2172.85
TriangleHeightChron	4108.15	5236.45	5282.15	5100.95	4514.70	3367.30	2325.10
TriangleHeightLDS	924.30	2079.45	2552.00	2786.75	2931.00	2541.35	2324.95
VarHeightChron	3760.25	5261.10	5229.25	5382.95	5035.65	3841.75	2401.30
VarHeightLDS	500.50	1873.95	2405.10	3439.20	3854.85	3117.30	2946.00

Table B.45. **Experiment 3, 15×15 Problems: Mean Number of Backtracks.**

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	218018.10	348004.55	373239.50	415959.50	452760.20	439076.15	312087.50
LJRandLDS	473876.60	723194.90	742052.35	721181.10	663025.15	573075.20	458852.30
CBASlackChron	94207.40	245729.45	282536.15	311654.10	284094.05	204779.70	133729.10
CBASlackLDS	53391.80	447845.40	558729.35	483931.60	416003.45	285535.55	221669.05
SumHeightChron	91089.85	281061.90	351226.40	393083.20	291164.70	182027.25	104050.45
SumHeightLDS	50756.90	366355.45	595528.75	557427.05	453595.10	318974.65	197801.30
JointHeightChron	129612.30	340604.15	343280.75	344416.00	308169.00	253734.20	174211.85
JointHeightLDS	31519.35	275561.10	377653.05	372771.20	348471.70	305001.90	229453.10
TriangleHeightChron	199454.30	305561.95	359197.55	406550.15	404112.70	328297.25	267448.90
TriangleHeightLDS	245182.65	492953.50	596648.10	594104.35	564066.10	459712.35	362983.45
VarHeightChron	168938.20	284734.00	333716.65	419334.75	412962.55	335791.45	228844.90
VarHeightLDS	159534.65	446445.20	596557.60	595169.45	564775.30	426232.20	352132.25

Table B.46. **Experiment 3, 15X15 Problems: Mean Number of Commitments.**

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	10774.00	11937.20	11691.05	11338.20	10898.10	9203.65	5751.90
LJRandLDS	26040.80	32291.10	26536.60	19494.25	15807.00	12029.70	8300.20
CBASlackChron	5951.40	12277.00	13245.90	13646.80	12542.30	8982.85	5388.55
CBASlackLDS	9114.25	56723.70	50280.75	35324.80	26871.40	15212.10	10236.50
SumHeightChron	3425.75	8829.85	11457.35	11190.95	8687.95	5082.30	2907.45
SumHeightLDS	2641.55	15101.95	22155.65	18406.05	14228.10	8806.25	5596.95
JointHeightChron	4904.20	10912.15	11180.60	11049.40	9089.65	6970.40	4811.20
JointHeightLDS	2106.60	14836.40	19167.60	15992.70	13183.85	9462.00	6925.50
TriangleHeightChron	8287.60	10521.65	10602.65	10222.15	9044.55	6746.80	4655.40
TriangleHeightLDS	10919.45	18373.00	19450.60	14310.30	12156.25	8852.60	6731.50
VarHeightChron	7609.05	10582.60	10507.50	10799.90	10098.10	7699.55	4811.10
VarHeightLDS	6668.05	17504.05	19710.15	17530.70	15963.70	11451.50	8244.15

Table B.47. **Experiment 3, 15X15 Problems: Mean Number of Heuristic Commitments.**

	Number of Bottlenecks						
Algorithm	**2**	**4**	**6**	**8**	**10**	**12**	**14**
LJRandChron	5.36	3.49	3.26	2.81	2.48	2.01	1.55
LJRandLDS	5.92	4.50	3.58	2.71	2.43	1.98	1.56
CBASlackChron	19.23	5.33	4.76	4.42	4.50	3.65	2.75
CBASlackLDS	20.13	13.21	9.01	7.18	6.45	4.48	3.33
SumHeightChron	4.57	3.69	3.39	2.87	2.98	2.37	2.08
SumHeightLDS	5.98	4.44	3.73	3.31	3.09	2.46	2.30
JointHeightChron	5.17	3.37	3.32	3.29	2.94	2.47	2.06
JointHeightLDS	7.46	5.75	5.15	4.35	3.76	2.83	2.38
TriangleHeightChron	4.62	3.56	2.99	2.55	2.32	1.92	1.54
TriangleHeightLDS	4.94	3.86	3.30	2.42	2.21	1.78	1.68
VarHeightChron	5.11	3.93	3.32	2.65	2.48	2.13	1.77
VarHeightLDS	4.86	4.05	3.29	2.95	2.84	2.50	2.01

Table B.48. **Experiment 3, 15×15 Problems: Mean Percentage of Heuristic Commitments.**

B.3.3 20×20 Results

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	0.90	1.00	1.00	1.00	0.85	0.50
LJRandLDS	0.75	1.00	1.00	1.00	0.85	0.60
CBASlackChron	0.35	1.00	1.00	1.00	0.65	0.30
CBASlackLDS	0.10	1.00	1.00	1.00	0.75	0.35
SumHeightChron	0.65	1.00	1.00	1.00	0.50	0.20
SumHeightLDS	0.30	1.00	1.00	1.00	0.70	0.35
JointHeightChron	0.50	1.00	1.00	1.00	0.70	0.35
JointHeightLDS	0.20	1.00	1.00	1.00	0.80	0.35
TriangleHeightChron	0.90	1.00	1.00	1.00	0.70	0.35
TriangleHeightLDS	0.70	1.00	1.00	1.00	0.80	0.35
VarHeightChron	0.45	1.00	1.00	1.00	0.70	0.40
VarHeightLDS	0.35	1.00	1.00	1.00	0.80	0.45

Table B.49. **Experiment 3, 20×20 Problems: Fraction of Problems Timed-out.**

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	1109.01	1200.29	1200.26	1200.27	1030.87	687.01
LJRandLDS	971.20	1200.31	1200.27	1200.27	1037.68	721.82
CBASlackChron	487.44	1200.20	1200.22	1200.20	860.48	393.16
CBASlackLDS	259.68	1200.48	1200.31	1200.19	952.27	481.55
SumHeightChron	817.95	1200.24	1200.23	1200.27	791.31	334.25
SumHeightLDS	587.41	1200.37	1200.34	1200.31	928.86	471.57
JointHeightChron	680.88	1200.33	1200.29	1200.32	936.34	442.41
JointHeightLDS	496.24	1200.72	1200.63	1200.53	1012.03	536.10
TriangleHeightChron	1116.39	1200.34	1200.29	1200.35	907.90	455.15
TriangleHeightLDS	991.67	1200.41	1200.50	1200.45	961.81	536.50
VarHeightChron	694.20	1200.33	1200.26	1200.20	942.14	513.08
VarHeightLDS	545.31	1200.39	1200.26	1200.38	962.43	644.40

Table B.50. **Experiment 3, 20×20 Problems: Mean CPU Time in Seconds.**

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	1719.70	2113.15	2097.90	2219.60	1790.55	1126.90
LJRandLDS	285.70	452.75	577.35	963.70	1111.35	956.40
CBASlackChron	750.75	2143.55	2482.30	2529.70	1789.30	755.30
CBASlackLDS	37.10	344.20	890.90	2056.05	1785.70	849.30
SumHeightChron	1331.80	2327.70	2330.95	2333.80	1507.90	600.35
SumHeightLDS	221.65	875.60	1238.10	1591.80	1402.45	738.35
JointHeightChron	1060.45	2112.45	2106.00	1871.00	1452.15	648.30
JointHeightLDS	76.60	360.35	477.75	651.10	764.40	460.10
TriangleHeightChron	1908.30	2058.90	1962.95	1906.45	1386.15	643.60
TriangleHeightLDS	416.95	730.75	785.85	923.45	881.95	613.95
VarHeightChron	1044.40	2009.20	2040.45	2090.95	1605.75	840.35
VarHeightLDS	174.35	632.85	843.05	1185.35	1213.35	910.65

Table B.51. **Experiment 3, 20×20 Problems: Mean Number of Backtracks.**

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	73483.10	129429.60	183842.20	214082.45	194256.60	139867.80
LJRandLDS	407814.20	480089.35	460838.25	419504.25	335978.70	202557.60
CBASlackChron	38772.95	100305.65	135353.20	139937.50	111221.25	51764.05
CBASlackLDS	78554.10	337240.40	330962.10	256123.95	172544.70	81556.65
SumHeightChron	109716.10	164669.15	170349.15	175959.55	124188.70	51522.80
SumHeightLDS	211395.90	420912.30	385081.70	324172.40	231555.15	98132.00
JointHeightChron	77036.00	154463.80	157087.40	153058.10	129207.75	60461.80
JointHeightLDS	95463.05	203344.10	200846.10	188709.20	158099.85	82167.80
TriangleHeightChron	105888.25	154722.65	188083.65	216783.80	177525.60	91748.20
TriangleHeightLDS	305347.10	388415.65	375359.45	372589.55	287373.90	138087.55
VarHeightChron	85846.30	149399.10	166752.35	187178.55	158776.20	83622.95
VarHeightLDS	179637.25	411476.45	400342.55	344159.20	254639.65	148565.75

Table B.52. **Experiment 3, 20×20 Problems: Mean Number of Commitments.**

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	3630.20	4318.50	4255.80	4471.30	3598.00	2258.95
LJRandLDS	24193.15	18553.65	12555.65	9218.05	5716.10	2992.55
CBASlackChron	3562.75	5473.25	5711.70	5324.10	3669.90	1542.25
CBASlackLDS	10087.80	39648.00	30228.30	17192.30	8362.45	2543.30
SumHeightChron	2848.85	4746.15	4742.65	4727.25	3044.00	1205.30
SumHeightLDS	11364.40	14584.80	12640.15	8354.70	5187.60	2080.70
JointHeightChron	2357.20	4330.05	4297.60	3801.20	2943.50	1310.50
JointHeightLDS	5701.50	10098.95	8559.45	6141.40	4334.45	1759.65
TriangleHeightChron	3951.55	4187.40	3972.55	3834.40	2784.15	1289.70
TriangleHeightLDS	16236.05	12802.10	8880.75	5643.35	3790.35	1969.20
VarHeightChron	2314.90	4092.15	4144.05	4227.65	3239.60	1693.70
VarHeightLDS	9883.65	13031.75	11342.50	9047.70	5970.50	2938.65

Table B.53. **Experiment 3, 20×20 Problems: Mean Number of Heuristic Commitments.**

	Number of Bottlenecks					
Algorithm	**0**	**4**	**8**	**12**	**16**	**20**
LJRandChron	5.63	3.75	2.42	2.16	1.72	1.00
LJRandLDS	6.03	3.87	2.73	2.22	1.57	0.97
CBASlackChron	30.95	5.83	4.35	3.83	2.90	1.50
CBASlackLDS	25.78	11.84	9.16	6.69	4.13	1.70
SumHeightChron	3.38	3.35	3.04	2.93	2.06	1.20
SumHeightLDS	6.01	3.46	3.27	2.56	1.90	1.23
JointHeightChron	4.65	2.96	2.81	2.52	1.96	1.13
JointHeightLDS	6.80	4.99	4.29	3.26	2.34	1.25
TriangleHeightChron	4.40	2.89	2.29	1.82	1.30	0.72
TriangleHeightLDS	5.40	3.31	2.39	1.52	1.15	0.80
VarHeightChron	3.79	2.89	2.79	2.49	1.87	1.24
VarHeightLDS	6.54	3.17	2.84	2.64	1.98	1.19

Table B.54. **Experiment 3, 20×20 Problems: Mean Percentage of Heuristic Commitments.**

Appendix C Detailed Results for the Experiments in Chapter 6

C.1 Experiment 1

	Number of Consumers per Process Plan				
Algorithm	**5**	**10**	**15**	**20**	**25**
CBASlackNoProp	0.60	0.85	0.90	0.85	0.90
CBASlackProp	0.30	0.50	0.55	0.65	0.60
GreedySumHeightNoProp	0.45	0.60	0.90	0.85	0.75
GreedySumHeightProp	0.25	0.20	0.60	0.45	0.50
VarHeightNoProp	0.40	0.40	0.50	0.70	0.75
VarHeightProp	0.05	0.10	0.30	0.25	0.25

Table C.1. **Experiment 1: Fraction of Problems Timed-out.**

	Number of Consumers per Process Plan				
Algorithm	**5**	**10**	**15**	**20**	**25**
CBASlackNoProp	724.16	1026.54	1113.83	1039.62	1080.08
CBASlackProp	360.63	604.89	678.18	784.33	725.86
GreedySumHeightNoProp	540.29	720.42	1080.24	1020.24	900.43
GreedySumHeightProp	339.63	240.94	720.73	540.93	600.89
VarHeightNoProp	480.58	482.46	639.46	840.65	900.52
VarHeightProp	63.87	122.23	363.86	301.52	305.17

Table C.2. **Experiment 1: Mean CPU Time in Seconds.**

	Number of Consumers per Process Plan				
Algorithm	**5**	**10**	**15**	**20**	**25**
CBASlackNoProp	9622.60	12556.35	12806.40	10802.00	9615.75
CBASlackProp	5410.75	7130.85	6667.75	6869.90	5836.35
GreedySumHeightNoProp	6473.65	7289.80	9823.35	8520.65	6208.15
GreedySumHeightProp	4235.25	2519.10	7417.15	5544.25	5393.15
VarHeightNoProp	5420.20	4639.40	5764.90	6538.05	6557.20
VarHeightProp	618.75	933.40	3665.80	2779.65	2246.40

Table C.3. **Experiment 1: Mean Number of Backtracks.**

	Number of Consumers per Process Plan				
Algorithm	**5**	**10**	**15**	**20**	**25**
CBASlackNoProp	64538.95	82914.05	126518.45	94886.55	100586.15
CBASlackProp	11861.70	52459.15	99881.20	94029.40	59560.50
GreedySumHeightNoProp	171862.20	206466.75	142097.40	139458.10	74183.90
GreedySumHeightProp	74861.55	52339.55	150908.35	78032.15	71649.65
VarHeightNoProp	122016.85	133475.10	123333.70	174536.55	136523.45
VarHeightProp	26334.50	28700.05	87663.85	67849.70	71671.50

Table C.4. **Experiment 1: Mean Number of Commitments.**

	Number of Consumers per Process Plan				
Algorithm	**5**	**10**	**15**	**20**	**25**
CBASlackNoProp	19427.15	25372.70	25851.05	21828.75	19478.75
CBASlackProp	10945.45	14448.60	13541.70	13941.45	11873.50
GreedySumHeightNoProp	17863.50	19597.75	36654.55	28158.90	23643.95
GreedySumHeightProp	7245.25	3839.45	11067.45	7681.65	7637.20
VarHeightNoProp	11859.05	9705.35	14331.15	16520.00	16544.35
VarHeightProp	1141.35	1861.90	5672.20	4040.30	3311.30

Table C.5. **Experiment 1: Mean Number of Heuristic Commitments.**

C.2 Experiment 2

C.2.1 5X5 Results

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	0.00	0.00	0.00	0.00
CBASlackProp	0.10	0.00	0.10	0.05
EorPProp	0.05	0.20	0.20	0.20
SumHeightProp	0.10	0.05	0.10	0.05
GreedyCBASlackProp	0.00	0.10	0.20	0.30
GreedyEorPProp	0.00	0.10	0.20	0.30
GreedySumHeightProp	0.00	0.10	0.20	0.30

Table C.6. **Experiment 2, 5X5 Problems: Fraction of Problems Timed-out.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	0.08	3.56	76.80	42.28
CBASlackProp	121.61	92.34	273.95	126.23
EorPProp	87.43	278.64	360.29	387.64
SumHeightProp	121.69	109.01	204.30	108.01
GreedyCBASlackProp	0.09	120.14	240.56	360.22
GreedyEorPProp	0.10	120.14	240.54	360.23
GreedySumHeightProp	0.11	120.15	240.57	360.23

Table C.7. **Experiment 2, 5X5 Problems: Mean CPU Time in Seconds.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	0.80	106.00	2235.30	987.00
CBASlackProp	8707.95	5232.25	12546.30	4801.65
EorPProp	5999.50	15229.30	14846.05	13281.60
SumHeightProp	8199.90	5998.60	9052.25	4125.40
GreedyCBASlackProp	1.05	4464.40	8210.15	10858.15
GreedyEorPProp	1.05	4467.45	8281.55	10622.30
GreedySumHeightProp	1.05	4323.20	8023.85	10476.65

Table C.8. **Experiment 2, 5X5 Problems: Mean Number of Backtracks.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	129.55	1759.80	40365.95	18852.25
CBASlackProp	17660.20	10660.95	25552.15	9861.60
EorPProp	19761.45	50581.05	81898.80	61735.75
SumHeightProp	16628.40	12188.65	18396.95	8505.50
GreedyCBASlackProp	137.05	65327.25	87867.00	138577.95
GreedyEorPProp	140.95	66054.70	88604.70	135551.55
GreedySumHeightProp	137.00	63346.60	86274.40	133535.20

Table C.9. **Experiment 2, 5✕5 Problems: Mean Number of Commitments.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	2.00	187.35	3095.40	1406.45
CBASlackProp	17418.10	10472.70	25108.65	9623.05
EorPProp	12001.00	30463.75	29698.05	26570.95
SumHeightProp	16401.95	12003.15	18115.65	8263.85
GreedyCBASlackProp	8.60	6961.65	12509.05	15298.70
GreedyEorPProp	10.75	6963.80	12643.70	14953.40
GreedySumHeightProp	8.50	6743.35	12226.85	14755.40

Table C.10. **Experiment 2, 5✕5 Problems: Mean Number of Heuristic Commitments.**

C.2.2 10✕10 Results

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	0.35	0.60	0.55	0.45
CBASlackProp	0.80	0.90	0.95	0.90
EorPProp	0.90	1.00	1.00	1.00
SumHeightProp	0.75	0.95	0.95	1.00
GreedyCBASlackProp	0.50	0.75	0.65	0.45
GreedyEorPProp	0.50	0.75	0.65	0.45
GreedySumHeightProp	0.50	0.75	0.65	0.45

Table C.11. **Experiment 2, 10✕10 Problems: Fraction of Problems Timed-out.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	421.69	721.87	664.08	545.57
CBASlackProp	963.50	1081.02	1140.54	1081.63
EorPProp	1080.07	1200.08	1200.13	1200.20
SumHeightProp	902.42	1140.38	1140.41	1200.21
GreedyCBASlackProp	600.94	901.15	782.17	545.11
GreedyEorPProp	600.95	901.16	782.20	545.26
GreedySumHeightProp	600.97	901.16	782.23	545.29

Table C.12. **Experiment 2, 10×10 Problems: Mean CPU Time in Seconds.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	3223.35	4557.05	2716.25	1662.35
CBASlackProp	8238.70	5669.75	4078.30	2687.35
EorPProp	9546.95	6579.85	4831.90	3359.10
SumHeightProp	7237.05	5326.90	3846.70	2993.20
GreedyCBASlackProp	5039.20	5604.20	3084.65	1458.05
GreedyEorPProp	4987.60	5550.20	3115.50	1456.35
GreedySumHeightProp	4909.25	5466.10	3067.00	1449.80

Table C.13. **Experiment 2, 10×10 Problems: Mean Number of Backtracks.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	120608.40	93338.60	50062.25	28944.05
CBASlackProp	121160.25	92469.20	60279.90	48588.65
EorPProp	177764.70	143691.35	67160.70	64437.40
SumHeightProp	142623.35	141704.50	92388.75	64644.55
GreedyCBASlackProp	99317.65	56370.50	30012.10	19237.10
GreedyEorPProp	97248.30	56028.00	30384.05	19092.30
GreedySumHeightProp	97000.55	55266.10	29742.60	19085.95

Table C.14. **Experiment 2, 10×10 Problems: Mean Number of Commitments.**

	Proportion of Maximum Consumers			
Algorithm	**0.2**	**0.4**	**0.6**	**0.8**
VarHeightProp	5219.45	6541.10	3892.80	2404.30
CBASlackProp	16716.25	11632.45	8484.80	5690.90
EorPProp	19139.60	13207.90	9713.25	6765.35
SumHeightProp	14597.85	10761.10	7834.10	6126.85
GreedyCBASlackProp	7619.00	7745.05	4416.40	2408.00
GreedyEorPProp	7578.05	7686.05	4475.40	2440.05
GreedySumHeightProp	7433.20	7560.00	4392.50	2397.10

Table C.15. **Experiment 2, 10✕10 Problems: Mean Number of Heuristic Commitments.**

C.3 Experiment 3

C.3.1 5✕5 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	0.10	0.25	0.05	0.05	0.20	0.05
CBASlackProp	0.00	0.40	0.75	0.80	0.85	0.95
EorPProp	0.25	0.70	0.70	0.70	0.40	0.70
SumHeightProp	0.05	0.15	0.50	0.65	0.80	0.95
GreedyCBASlackProp	0.30	0.80	0.60	0.50	0.35	0.20
GreedyEorPProp	0.30	0.80	0.60	0.50	0.35	0.25
GreedySumHeightProp	0.30	0.80	0.60	0.45	0.35	0.20

Table C.16. **Experiment 3, 5✕5 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	155.77	351.84	135.68	61.76	241.52	61.85
CBASlackProp	23.53	485.70	901.02	962.88	1020.36	1140.12
EorPProp	300.16	933.28	847.94	893.07	487.27	840.28
SumHeightProp	63.80	215.84	616.64	784.90	965.66	1140.79
GreedyCBASlackProp	360.13	960.18	720.53	600.65	421.12	241.66
GreedyEorPProp	360.12	960.17	720.53	600.61	421.27	302.59
GreedySumHeightProp	360.12	960.18	720.53	540.71	421.08	241.41

Table C.17. **Experiment 3, 5✕5 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	1808.15	4779.60	1594.05	701.25	3554.80	809.40
CBASlackProp	293.15	10016.60	18260.95	19116.35	18376.20	21472.10
EorPProp	5533.75	17924.00	17442.25	19570.90	10518.60	17856.25
SumHeightProp	757.60	3746.25	12309.40	15836.70	19226.75	23936.50
GreedyCBASlackProp	6925.75	15031.70	11621.30	7866.90	5399.80	2608.35
GreedyEorPProp	7107.70	15179.20	12129.45	8615.05	6320.40	4286.95
GreedySumHeightProp	6954.60	14797.95	11475.15	7293.05	5453.30	2649.40

Table C.18. **Experiment 3, 5✕5 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	68704.60	132915.75	51621.70	10586.05	12662.75	3303.35
CBASlackProp	11769.05	57899.40	45483.25	49528.20	55659.75	50989.85
EorPProp	123086.00	297681.10	163353.10	92972.75	46229.80	94641.65
SumHeightProp	34310.10	60448.00	50747.20	49482.85	61632.95	59166.45
GreedyCBASlackProp	124832.65	345913.90	151264.20	118466.90	62000.85	49171.80
GreedyEorPProp	129044.70	350686.15	147614.85	120238.70	71764.30	41062.05
GreedySumHeightProp	124970.10	341861.65	160748.80	113812.65	84849.60	58263.00

Table C.19. **Experiment 3, 5✕5 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	3054.00	7496.40	2878.85	1563.10	6436.90	1577.85
CBASlackProp	601.60	20156.70	36715.45	38434.60	36962.50	43159.90
EorPProp	11072.35	35868.70	34920.05	39182.10	21081.45	35753.45
SumHeightProp	1523.25	7555.45	24721.05	31789.80	38580.25	48003.85
GreedyCBASlackProp	9885.55	23016.45	19297.35	14221.85	10300.05	5394.40
GreedyEorPProp	10142.35	23241.00	20260.25	15675.70	12031.60	8696.05
GreedySumHeightProp	9919.90	22669.45	19050.20	13040.70	10311.60	5436.00

Table C.20. **Experiment 3, 5✕5 Problems: Mean Number of Heuristic Commitments.**

C.3.2 10×10 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	0.35	0.65	0.20	0.15	0.10	0.25
CBASlackProp	0.45	0.95	1.00	1.00	1.00	1.00
EorPProp	0.45	1.00	1.00	1.00	0.95	1.00
SumHeightProp	0.45	1.00	0.95	0.95	0.85	1.00
GreedyCBASlackProp	0.45	1.00	0.95	0.90	0.60	0.35
GreedyEorPProp	0.45	1.00	0.95	1.00	0.85	0.45
GreedySumHeightProp	0.45	1.00	0.95	0.95	0.75	0.55

Table C.21. **Experiment 3, 10×10 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	451.81	799.15	289.01	215.21	159.36	336.18
CBASlackProp	540.57	1142.96	1200.41	1200.32	1200.30	1200.33
EorPProp	540.51	1200.24	1200.28	1200.28	1140.96	1200.28
SumHeightProp	540.58	1200.27	1141.52	1141.71	1024.14	1200.35
GreedyCBASlackProp	540.56	1200.24	1141.40	1089.85	759.11	469.94
GreedyEorPProp	540.50	1200.29	1141.29	1200.45	1025.24	558.83
GreedySumHeightProp	540.57	1200.27	1141.36	1142.47	931.12	719.42

Table C.22. **Experiment 3, 10×10 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	1136.45	1060.20	362.30	251.90	153.55	416.45
CBASlackProp	752.55	1748.70	1722.80	1723.50	1755.20	1840.10
EorPProp	1068.75	2355.95	2200.20	2221.30	2215.95	2285.30
SumHeightProp	987.95	2021.40	1689.65	1641.60	1578.90	1828.50
GreedyCBASlackProp	1038.45	2241.15	1810.05	1423.50	918.70	411.25
GreedyEorPProp	1126.75	2328.80	1902.90	1695.35	1445.65	715.50
GreedySumHeightProp	1083.70	2228.00	1779.70	1448.35	1106.65	742.35

Table C.23. **Experiment 3, 10×10 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	35489.50	73509.60	15975.60	13104.55	5364.75	5812.65
CBASlackProp	48762.45	54940.90	36101.80	31459.45	18534.55	18421.90
EorPProp	19466.10	46229.95	49057.75	42568.50	23922.15	33975.60
SumHeightProp	55477.60	95784.45	91393.50	76414.65	36995.30	58857.55
GreedyCBASlackProp	36672.45	60400.30	51465.30	41319.50	35380.75	18619.90
GreedyEorPProp	39530.15	61041.95	45033.20	28948.65	24585.50	12512.65
GreedySumHeightProp	38224.80	59495.10	58266.85	45398.30	43678.00	31263.30

Table C.24. **Experiment 3, 10×10 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
VarHeightProp	1915.65	2182.55	1140.90	978.95	875.50	1408.60
CBASlackProp	1879.10	4678.60	4768.70	4727.85	4934.30	5082.65
EorPProp	2192.75	4857.55	4564.25	4611.85	4607.80	4742.35
SumHeightProp	2049.05	4354.70	3725.95	3727.25	3715.60	4170.50
GreedyCBASlackProp	1557.90	3570.55	3450.60	3493.90	2805.75	2303.05
GreedyEorPProp	1683.85	3710.45	3642.30	4009.30	3539.10	2138.15
GreedySumHeightProp	1623.15	3554.65	3377.00	3484.45	2870.80	2282.80

Table C.25. **Experiment 3, 10×10 Problems: Mean Number of Heuristic Commitments.**

Appendix D Detailed Results for the Experiments in Chapter 7

D.1 Experiment 1

D.1.1 Overall Results

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	0.00	0.02	0.00	0.04
CBASlackPropPEX	0.00	0.02	0.00	0.04
LJRandPEX	0.21	0.42	0.49	0.55
LJRandPropPEX	0.11	0.23	0.20	0.25
SumHeightPEX	0.01	0.04	0.06	0.08
SumHeightPropPEX	0.00	0.00	0.00	0.00
VarHeightPEX	0.01	0.02	0.03	0.09
VarHeightPropPEX	0.00	0.00	0.00	0.00

Table D.1. **Experiment 1 Overall: Fraction of Problems Timed-out.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	4.70	40.34	64.53	153.23
CBASlackPropPEX	4.67	39.79	57.96	136.86
LJRandPEX	292.23	543.90	614.82	677.15
LJRandPropPEX	144.71	320.14	282.05	340.55
SumHeightPEX	21.49	97.56	98.12	161.51
SumHeightPropPEX	2.27	7.62	4.56	17.05
VarHeightPEX	16.27	64.30	63.53	156.11
VarHeightPropPEX	1.82	5.82	4.13	12.27

Table D.2. **Experiment 1 Overall: Mean CPU Time in Seconds.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	94.93	636.41	1112.53	2593.71
CBASlackPropPEX	90.43	616.36	953.33	2189.79
LJRandPEX	6147.72	9512.61	9745.32	10429.66
LJRandPropPEX	2407.81	4339.43	3233.32	3769.48
SumHeightPEX	389.33	1584.38	1439.28	2281.82
SumHeightPropPEX	18.91	65.31	26.42	133.64
VarHeightPEX	260.34	909.32	837.02	1973.79
VarHeightPropPEX	10.58	42.65	18.47	80.35

Table D.3. **Experiment 1 Overall: Mean Number of Backtracks.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	2902.64	16072.19	28457.79	66834.32
CBASlackPropPEX	2781.42	15359.41	23993.05	58860.02
LJRandPEX	88666.53	94722.18	82219.18	88306.67
LJRandPropPEX	75694.16	124503.93	98543.12	118501.88
SumHeightPEX	5271.70	20157.82	17843.91	27095.17
SumHeightPropPEX	1192.01	3246.88	1955.11	7342.73
VarHeightPEX	3793.04	12011.95	11117.13	24445.48
VarHeightPropPEX	849.03	2534.23	1561.44	4937.83

Table D.4. **Experiment 1 Overall: Mean Number of Commitments.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	278.63	1347.07	2325.10	5270.64
CBASlackPropPEX	270.45	1306.36	2005.65	4465.02
LJRandPEX	13138.06	21138.97	21923.18	23219.40
LJRandPropPEX	4852.47	8709.38	6506.57	7573.18
SumHeightPEX	819.02	3207.13	2933.75	4612.62
SumHeightPropPEX	79.03	166.96	105.55	312.52
VarHeightPEX	561.47	1855.73	1730.81	3994.14
VarHeightPropPEX	61.97	121.59	90.42	205.57

Table D.5. **Experiment 1 Overall: Mean Number of Heuristic Commitments.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	14.96	11.29	14.66	12.15
CBASlackPropPEX	15.07	10.94	14.22	11.73
LJRandPEX	10.77	17.76	21.20	21.18
LJRandPropPEX	5.28	5.08	5.83	5.17
SumHeightPEX	9.32	9.97	12.41	12.47
SumHeightPropPEX	7.70	6.25	8.39	6.96
VarHeightPEX	9.01	9.34	12.20	11.65
VarHeightPropPEX	7.75	6.09	8.58	7.08

Table D.6. **Experiment 1 Overall: Mean Percentage of Heuristic Commitments.**

D.1.2 One-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
LJRandPEX	0.00	0.00	0.35	0.55	0.20	0.15
LJRandPropPEX	0.00	0.00	0.15	0.35	0.15	0.00
SumHeightPEX	0.00	0.00	0.00	0.05	0.00	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.00	0.00	0.00	0.05	0.00	0.00
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.7. **Experiment 1, 1-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.26	0.27	2.30	18.01	5.01	2.33
CBASlackPropPEX	0.25	0.26	2.34	18.32	4.60	2.22
LJRandPEX	0.26	15.37	447.51	730.86	342.06	217.30
LJRandPropPEX	0.26	0.26	193.43	432.18	229.28	12.83
SumHeightPEX	0.25	0.31	27.34	95.00	5.17	0.87
SumHeightPropPEX	0.25	0.25	1.80	8.01	1.87	1.45
VarHeightPEX	0.27	0.31	12.34	72.37	11.14	1.21
VarHeightPropPEX	0.27	0.27	1.66	5.37	1.70	1.64

Table D.8. **Experiment 1, 1-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.00	1.25	47.85	396.90	93.00	29.60
CBASlackPropPEX	1.00	1.00	43.80	394.80	77.75	24.20
LJRandPEX	1.00	381.60	9879.85	15385.75	6776.40	4461.75
LJRandPropPEX	1.00	1.00	3283.40	7091.50	3863.25	206.70
SumHeightPEX	1.00	1.95	504.30	1745.75	82.55	0.40
SumHeightPropPEX	1.00	1.00	15.65	88.85	6.80	0.15
VarHeightPEX	1.00	1.70	201.60	1173.35	180.65	3.75
VarHeightPropPEX	1.00	1.00	11.90	47.45	1.75	0.35

Table D.9. **Experiment 1, 1-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	135.25	160.60	1465.70	11223.35	3045.95	1385.00
CBASlackPropPEX	135.25	145.90	1342.65	11163.00	2679.25	1222.50
LJRandPEX	135.25	4502.25	133662.10	250940.35	103645.35	39113.90
LJRandPropPEX	135.25	145.90	89178.85	245309.25	112621.95	6773.75
SumHeightPEX	135.25	168.55	7905.95	20981.70	1887.30	551.45
SumHeightPropPEX	135.25	145.90	1001.90	4479.70	819.50	569.80
VarHeightPEX	135.25	162.25	3494.70	15209.00	3160.20	596.85
VarHeightPropPEX	135.25	145.90	813.35	2765.80	666.70	567.20

Table D.10. **Experiment 1, 1-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.50	108.10	873.65	356.75	332.80
CBASlackPropPEX	0.00	0.00	100.00	869.35	323.60	329.75
LJRandPEX	0.00	824.85	21052.35	32877.75	14667.80	9405.60
LJRandPropPEX	0.00	0.00	6577.45	14235.15	7799.70	502.55
SumHeightPEX	0.00	1.90	1017.85	3533.40	248.95	112.00
SumHeightPropPEX	0.00	0.00	40.60	221.65	100.00	111.95
VarHeightPEX	0.00	1.40	410.50	2388.00	445.30	123.60
VarHeightPropPEX	0.00	0.00	32.30	138.35	85.30	115.85

Table D.11. **Experiment 1, 1-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.17	4.17	12.83	26.57	46.01
CBASlackPropPEX	0.00	0.00	3.74	12.74	26.46	47.51
LJRandPEX	0.00	2.97	12.26	16.58	15.87	16.94
LJRandPropPEX	0.00	0.00	3.44	7.39	9.85	11.02
SumHeightPEX	0.00	0.47	5.25	12.59	16.29	21.32
SumHeightPropPEX	0.00	0.00	2.68	7.80	15.03	20.68
VarHeightPEX	0.00	0.42	4.35	10.47	16.13	22.67
VarHeightPropPEX	0.00	0.00	2.50	7.95	14.02	22.00

Table D.12. **Experiment 1, 1-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.1.3 Three-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.10	0.00	0.00
CBASlackPropPEX	0.00	0.00	0.00	0.10	0.00	0.00
LJRandPEX	0.00	0.05	0.40	0.80	0.70	0.60
LJRandPropPEX	0.00	0.00	0.25	0.65	0.20	0.25
SumHeightPEX	0.00	0.00	0.05	0.10	0.05	0.05
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.00	0.00	0.00	0.05	0.05	0.00
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.13. **Experiment 1, 3-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.10	0.25	18.55	149.99	15.92	57.25
CBASlackPropPEX	0.11	0.11	15.22	148.34	17.58	57.38
LJRandPEX	0.11	60.11	552.61	1014.07	910.80	725.71
LJRandPropPEX	0.11	0.11	383.85	831.11	390.59	315.05
SumHeightPEX	0.10	0.14	121.11	299.84	86.74	77.46
SumHeightPropPEX	0.10	0.11	3.23	9.53	28.96	3.78
VarHeightPEX	0.11	0.13	35.36	156.56	151.53	42.09
VarHeightPropPEX	0.12	0.11	2.25	13.04	17.35	2.04

Table D.14. **Experiment 1, 3-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.00	3.65	334.05	2060.15	302.25	1117.35
CBASlackPropPEX	1.00	1.00	247.30	2016.00	324.35	1108.50
LJRandPEX	1.00	1082.30	10606.10	18445.60	15650.75	11289.90
LJRandPropPEX	1.00	1.00	5244.45	10876.00	5193.50	4720.65
SumHeightPEX	1.00	1.65	1949.25	4844.35	1349.90	1360.10
SumHeightPropPEX	1.00	1.00	29.90	73.60	260.65	25.70
VarHeightPEX	1.00	1.20	575.80	2153.30	2143.35	581.30
VarHeightPropPEX	1.00	1.00	17.40	101.75	133.60	1.15

Table D.15. **Experiment 1, 3-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	176.25	256.90	8738.50	48972.60	7671.45	30617.45
CBASlackPropPEX	176.25	187.35	6772.00	46492.55	8184.90	30343.40
LJRandPEX	176.25	11503.90	115480.65	224197.45	133291.05	83683.80
LJRandPropPEX	176.25	187.35	153710.30	336382.90	156560.00	100006.75
SumHeightPEX	176.25	194.30	23680.70	61266.45	18865.80	16763.40
SumHeightPropPEX	176.25	187.35	1632.95	4042.75	11984.65	1457.30
VarHeightPEX	176.25	190.05	8968.85	29644.65	27632.65	5459.25
VarHeightPropPEX	176.25	187.35	1209.25	5909.90	7050.50	672.15

Table D.16. **Experiment 1, 3-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	5.30	675.00	4171.55	743.00	2487.55
CBASlackPropPEX	0.00	0.00	501.30	4081.70	788.40	2466.75
LJRandPEX	0.00	2364.70	23596.50	40250.40	35489.05	25133.15
LJRandPropPEX	0.00	0.00	10499.60	21777.50	10460.95	9518.20
SumHeightPEX	0.00	1.30	3902.70	9725.70	2777.40	2835.65
SumHeightPropPEX	0.00	0.00	64.80	180.75	598.10	158.10
VarHeightPEX	0.00	0.40	1156.05	4345.10	4359.55	1273.25
VarHeightPropPEX	0.00	0.00	39.75	236.65	344.85	108.30

Table D.17. **Experiment 1, 3-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.36	4.12	8.40	17.84	37.01
CBASlackPropPEX	0.00	0.00	3.23	8.02	17.90	36.48
LJRandPEX	0.00	1.09	15.46	23.97	35.30	30.73
LJRandPropPEX	0.00	0.00	4.15	6.70	9.13	10.51
SumHeightPEX	0.00	0.43	8.49	13.17	16.44	21.28
SumHeightPropPEX	0.00	0.00	2.11	5.77	11.91	17.71
VarHeightPEX	0.00	0.15	7.02	12.30	15.92	20.62
VarHeightPropPEX	0.00	0.00	1.77	5.32	11.84	17.62

Table D.18. **Experiment 1, 3-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.1.4 Five-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
LJRandPEX	0.05	0.25	0.75	0.80	0.60	0.50
LJRandPropPEX	0.00	0.00	0.55	0.50	0.10	0.05
SumHeightPEX	0.00	0.00	0.20	0.15	0.00	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.05	0.00	0.10	0.05	0.00	0.00
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.19. **Experiment 1, 5-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.97	0.81	138.78	187.36	56.25	3.00
CBASlackPropPEX	0.61	0.72	108.79	176.37	57.86	3.43
LJRandPEX	60.57	303.67	942.54	1028.47	728.56	625.14
LJRandPropPEX	0.65	0.81	686.98	686.25	231.31	86.32
SumHeightPEX	1.81	1.03	300.96	235.43	48.14	1.36
SumHeightPropPEX	0.61	0.73	10.67	10.22	2.66	2.47
VarHeightPEX	60.59	1.45	179.75	135.05	2.73	1.60
VarHeightPropPEX	0.66	0.78	9.61	8.22	2.73	2.80

Table D.20. **Experiment 1, 5-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	6.70	4.35	2358.40	3305.65	978.20	21.85
CBASlackPropPEX	1.05	1.20	1711.75	3005.80	978.35	21.80
LJRandPEX	1000.70	5148.95	15810.80	16230.70	11011.35	9269.40
LJRandPropPEX	1.30	1.80	7699.85	7856.80	2784.85	1055.35
SumHeightPEX	19.90	7.90	4383.85	3522.60	701.30	0.15
SumHeightPropPEX	1.00	1.05	79.15	74.95	2.25	0.10
VarHeightPEX	817.40	12.80	2392.30	1782.90	16.50	0.20
VarHeightPropPEX	1.05	1.05	62.70	45.80	0.20	0.00

Table D.21. **Experiment 1, 5-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	405.35	369.35	69493.70	75595.60	23567.45	1315.30
CBASlackPropPEX	244.10	283.75	50461.15	68069.45	23585.10	1314.75
LJRandPEX	8728.95	51343.35	155630.20	147687.75	79254.35	50670.50
LJRandPropPEX	252.75	319.70	247499.80	237096.30	80877.10	25213.10
SumHeightPEX	482.45	355.05	55101.05	41659.80	8876.75	588.35
SumHeightPropPEX	243.90	281.70	5150.85	4632.35	804.45	617.40
VarHeightPEX	9674.45	420.10	33116.15	22062.40	833.20	596.50
VarHeightPropPEX	244.80	286.10	4191.20	3351.15	670.80	624.60

Table D.22. **Experiment 1, 5-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	13.00	11.40	4749.85	6711.80	2142.50	322.05
CBASlackPropPEX	1.15	7.55	3454.10	6109.60	2141.40	320.10
LJRandPEX	2156.50	11942.20	36165.85	36080.05	24356.05	20838.45
LJRandPropPEX	3.70	5.70	15414.30	15767.90	5651.45	2196.35
SumHeightPEX	39.75	18.40	8791.20	7111.90	1511.90	129.35
SumHeightPropPEX	1.00	5.70	177.40	211.60	111.30	126.30
VarHeightPEX	1634.70	27.30	4811.80	3637.65	144.50	128.90
VarHeightPropPEX	1.30	2.80	142.80	155.50	111.05	129.05

Table D.23. **Experiment 1, 5-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.32	1.62	7.96	14.33	23.72	40.03
CBASlackPropPEX	0.13	0.95	6.63	14.22	23.59	39.80
LJRandPEX	1.26	9.13	21.87	28.18	30.66	36.09
LJRandPropPEX	0.36	0.57	5.55	7.68	9.93	10.88
SumHeightPEX	0.71	2.95	13.06	16.85	18.58	22.30
SumHeightPropPEX	0.12	0.76	3.23	8.84	16.50	20.91
VarHeightPEX	0.86	3.06	11.29	16.98	18.98	22.00
VarHeightPropPEX	0.15	0.34	3.02	9.38	17.22	21.40

Table D.24. **Experiment 1, 5-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.1.5 Seven-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.05	0.15	0.05
CBASlackPropPEX	0.00	0.00	0.05	0.05	0.10	0.05
LJRandPEX	0.05	0.25	0.70	0.85	0.70	0.75
LJRandPropPEX	0.00	0.10	0.25	0.50	0.45	0.20
SumHeightPEX	0.00	0.00	0.05	0.20	0.25	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.00	0.05	0.00	0.20	0.25	0.05
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.25. **Experiment 1, 7-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.22	46.47	198.49	223.17	336.19	114.82
CBASlackPropPEX	0.15	33.45	194.62	201.12	277.02	114.79
LJRandPEX	60.15	300.46	879.24	1020.17	896.97	905.90
LJRandPropPEX	0.15	142.22	386.00	616.98	601.71	296.23
SumHeightPEX	2.54	73.74	174.41	339.71	340.18	38.46
SumHeightPropPEX	0.15	2.54	3.69	78.94	14.08	2.90
VarHeightPEX	1.14	91.52	68.57	342.59	351.20	81.64
VarHeightPropPEX	0.16	2.39	3.49	53.85	10.61	3.14

Table D.26. **Experiment 1, 7-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	2.15	688.25	3300.20	3655.50	5973.45	1942.70
CBASlackPropPEX	1.00	426.05	3112.95	2863.20	4818.65	1916.90
LJRandPEX	900.15	4483.10	14213.55	15648.00	13562.90	13770.25
LJRandPropPEX	1.00	1453.95	4302.80	7136.55	6582.15	3140.45
SumHeightPEX	35.05	960.65	2463.80	4887.15	4830.15	514.15
SumHeightPropPEX	1.00	17.30	22.20	649.30	105.25	6.80
VarHeightPEX	13.35	1085.35	872.95	4387.15	4461.35	1022.60
VarHeightPropPEX	1.00	14.40	17.65	382.95	61.30	4.80

Table D.27. **Experiment 1, 7-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	263.30	23700.20	84412.50	98016.00	146180.15	48433.80
CBASlackPropPEX	243.40	16379.95	85651.95	85007.50	117754.75	48122.60
LJRandPEX	6169.50	39390.45	140058.40	142322.75	129298.40	72600.50
LJRandPropPEX	243.40	48850.55	125190.40	220814.65	219381.30	96530.95
SumHeightPEX	537.80	12218.05	26554.35	65790.55	51488.15	5982.15
SumHeightPropPEX	243.55	1388.95	1662.45	33731.25	6080.60	949.60
VarHeightPEX	494.30	11020.75	10956.10	63728.60	49520.95	10952.15
VarHeightPropPEX	243.40	1167.55	1414.65	21648.20	4217.05	936.15

Table D.28. **Experiment 1, 7-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	2.35	1378.45	6629.05	7378.85	12111.85	4123.30
CBASlackPropPEX	0.00	852.70	6251.80	5798.90	9812.80	4073.90
LJRandPEX	2146.15	10180.45	31130.45	35335.45	29997.15	30526.75
LJRandPropPEX	0.00	2912.95	8627.85	14317.20	13224.70	6356.40
SumHeightPEX	68.10	1922.55	4951.40	9832.95	9750.65	1150.05
SumHeightPropPEX	0.00	36.40	61.95	1347.00	294.20	135.60
VarHeightPEX	24.70	2172.85	1769.05	8830.45	9006.90	2160.90
VarHeightPropPEX	0.00	29.85	58.25	816.85	204.90	123.55

Table D.29. **Experiment 1, 7-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.30	1.90	6.24	10.34	21.82	32.29
CBASlackPropPEX	0.00	1.27	4.44	10.38	22.01	32.30
LJRandPEX	1.80	8.85	23.60	26.68	25.90	40.26
LJRandPropPEX	0.00	1.59	5.18	7.23	7.87	9.15
SumHeightPEX	1.09	3.53	12.07	15.25	19.55	23.36
SumHeightPropPEX	0.00	0.98	3.14	7.63	10.93	19.05
VarHeightPEX	0.46	4.08	11.88	13.82	17.86	21.80
VarHeightPropPEX	0.00	0.88	4.04	7.90	11.55	18.12

Table D.30. **Experiment 1, 7-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.2 Experiment 2

D.2.1 Overall Results

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	0.00	0.00	0.43	0.60
CBASlackPropPEX	0.00	0.00	0.44	0.59
LJRandPEX	0.01	0.49	0.65	0.74
LJRandPropPEX	0.00	0.20	0.59	0.68
SumHeightPEX	0.00	0.06	0.44	0.67
SumHeightPropPEX	0.00	0.00	0.25	0.52
VarHeightPEX	0.00	0.03	0.43	0.63
VarHeightPropPEX	0.00	0.00	0.23	0.56

Table D.31. **Experiment 2 Overall: Fraction of Problems Timed-out.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	0.12	64.53	538.76	733.88
CBASlackPropPEX	0.08	57.96	547.28	732.95
LJRandPEX	18.31	614.82	789.77	895.76
LJRandPropPEX	0.11	282.05	721.89	828.19
SumHeightPEX	0.13	98.12	569.54	815.28
SumHeightPropPEX	0.10	4.56	329.63	661.54
VarHeightPEX	0.14	63.53	531.39	775.55
VarHeightPropPEX	0.11	4.13	323.84	691.05

Table D.32. **Experiment 2 Overall: Mean CPU Time in Seconds.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	4.34	1112.53	3086.16	1444.59
CBASlackPropPEX	1.06	953.33	2971.62	1257.19
LJRandPEX	1411.28	9745.32	3110.13	1295.90
LJRandPropPEX	1.56	3233.32	2148.22	859.42
SumHeightPEX	4.13	1439.28	2354.68	1322.41
SumHeightPropPEX	0.65	26.42	787.73	540.35
VarHeightPEX	3.60	837.02	2098.49	1170.77
VarHeightPropPEX	0.70	18.47	706.85	523.06

Table D.33. **Experiment 2 Overall: Mean Number of Backtracks.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	147.80	28457.79	114637.06	76523.38
CBASlackPropPEX	131.47	23993.05	109110.17	62690.99
LJRandPEX	7363.59	82219.18	71178.33	45915.08
LJRandPropPEX	146.63	98543.12	88908.11	43496.00
SumHeightPEX	123.17	17843.91	44773.49	42457.52
SumHeightPropPEX	119.41	1955.11	51475.72	42238.53
VarHeightPEX	123.64	11117.13	44331.42	48063.00
VarHeightPropPEX	120.03	1561.44	54455.00	48005.02

Table D.34. **Experiment 2 Overall: Mean Number of Commitments.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	21.18	2325.10	6464.90	3638.53
CBASlackPropPEX	13.65	2005.65	6236.67	3261.46
LJRandPEX	3410.23	21923.18	6636.85	2809.23
LJRandPropPEX	12.67	6506.57	4353.38	1825.48
SumHeightPEX	21.30	2933.75	4795.79	2771.67
SumHeightPropPEX	12.80	105.55	1666.05	1222.83
VarHeightPEX	19.78	1730.81	4284.28	2471.52
VarHeightPropPEX	11.97	90.42	1505.73	1179.98

Table D.35. **Experiment 2 Overall: Mean Number of Heuristic Commitments.**

	Problem Size			
Algorithm	**5**	**10**	**15**	**20**
CBASlackPEX	12.52	14.66	12.46	11.20
CBASlackPropPEX	9.13	14.22	12.32	11.07
LJRandPEX	17.72	21.20	9.41	6.05
LJRandPropPEX	6.58	5.83	3.96	3.67
SumHeightPEX	13.70	12.41	9.38	6.61
SumHeightPropPEX	9.15	8.39	4.93	3.44
VarHeightPEX	13.33	12.20	8.95	5.24
VarHeightPropPEX	8.60	8.58	4.74	3.00

Table D.36. **Experiment 2 Overall: Mean Percentage of Heuristic Commitments.**

D.2.2 5×5 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
LJRandPEX	0.00	0.05	0.00	0.00	0.00	0.00
LJRandPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightPEX	0.00	0.00	0.00	0.00	0.00	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.37. **Experiment 2, 5×5 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.05	0.09	0.12	0.20	0.14	0.11
CBASlackPropPEX	0.04	0.06	0.07	0.11	0.11	0.11
LJRandPEX	0.75	60.68	41.63	3.76	2.64	0.41
LJRandPropPEX	0.04	0.06	0.08	0.17	0.14	0.15
SumHeightPEX	0.04	0.10	0.12	0.22	0.21	0.11
SumHeightPropPEX	0.04	0.06	0.09	0.13	0.15	0.16
VarHeightPEX	0.05	0.09	0.18	0.24	0.16	0.13
VarHeightPropPEX	0.04	0.06	0.09	0.14	0.16	0.16

Table D.38. **Experiment 2, 5X5 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.65	3.85	4.85	9.30	3.85	2.55
CBASlackPropPEX	0.95	1.20	0.95	1.80	0.70	0.75
LJRandPEX	44.30	4376.70	3578.10	256.10	191.10	21.40
LJRandPropPEX	0.95	1.00	1.10	3.65	1.25	1.40
SumHeightPEX	1.10	3.90	3.80	8.55	7.10	0.30
SumHeightPropPEX	0.95	0.85	0.75	0.80	0.25	0.30
VarHeightPEX	1.25	2.90	6.45	9.00	1.85	0.15
VarHeightPropPEX	1.00	0.85	0.85	1.05	0.45	0.00

Table D.39. **Experiment 2, 5X5 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	78.30	108.15	137.90	219.15	176.95	166.35
CBASlackPropPEX	77.70	99.10	122.85	171.75	158.50	158.90
LJRandPEX	332.20	21591.15	19600.20	1389.45	1006.55	262.00
LJRandPropPEX	77.80	102.10	125.75	202.40	185.35	186.40
SumHeightPEX	73.10	96.40	119.40	168.75	156.15	125.20
SumHeightPropPEX	77.05	93.50	116.50	144.70	146.75	137.95
VarHeightPEX	73.10	93.80	139.15	173.10	136.60	126.10
VarHeightPropPEX	77.15	95.15	117.70	147.45	147.45	135.30

Table D.40. **Experiment 2, 5X5 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	2.30	9.75	16.25	32.05	31.70	35.00
CBASlackPropPEX	0.50	3.40	7.40	15.95	24.20	30.45
LJRandPEX	105.00	10347.50	8874.25	598.40	464.25	71.95
LJRandPropPEX	0.65	4.75	8.55	19.70	19.95	22.40
SumHeightPEX	1.45	9.55	14.50	32.40	38.75	31.15
SumHeightPropPEX	0.70	3.05	7.50	15.10	21.90	28.55
VarHeightPEX	1.50	7.70	19.25	32.20	27.55	30.45
VarHeightPropPEX	0.50	2.25	6.70	14.15	22.35	25.85

Table D.41. **Experiment 2, 5×5 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.52	5.64	9.49	14.84	19.37	24.23
CBASlackPropPEX	0.27	1.88	4.27	9.62	16.45	22.27
LJRandPEX	5.61	12.59	17.12	29.33	22.35	19.31
LJRandPropPEX	0.34	2.40	4.76	8.97	10.70	12.32
SumHeightPEX	1.16	5.95	9.69	17.08	22.65	25.69
SumHeightPropPEX	0.40	1.84	4.59	10.10	15.69	22.31
VarHeightPEX	1.31	5.35	10.79	16.87	20.52	25.16
VarHeightPropPEX	0.28	1.27	3.99	9.45	16.08	20.53

Table D.42. **Experiment 2, 5×5 Problems: Mean Percentage of Heuristic Commitments.**

D.2.3 10×10 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.00	0.00	0.00	0.00	0.00
CBASlackPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
LJRandPEX	0.05	0.25	0.75	0.80	0.60	0.50
LJRandPropPEX	0.00	0.00	0.55	0.50	0.10	0.05
SumHeightPEX	0.00	0.00	0.20	0.15	0.00	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.05	0.00	0.10	0.05	0.00	0.00
VarHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00

Table D.43. **Experiment 2, 10×10 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.97	0.81	138.78	187.36	56.25	3.00
CBASlackPropPEX	0.61	0.72	108.79	176.37	57.86	3.43
LJRandPEX	60.57	303.67	942.54	1028.47	728.56	625.14
LJRandPropPEX	0.65	0.81	686.98	686.25	231.31	86.32
SumHeightPEX	1.81	1.03	300.96	235.43	48.14	1.36
SumHeightPropPEX	0.61	0.73	10.67	10.22	2.66	2.47
VarHeightPEX	60.59	1.45	179.75	135.05	2.73	1.60
VarHeightPropPEX	0.66	0.78	9.61	8.22	2.73	2.80

Table D.44. **Experiment 2, 10×10 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	6.70	4.35	2358.40	3305.65	978.20	21.85
CBASlackPropPEX	1.05	1.20	1711.75	3005.80	978.35	21.80
LJRandPEX	1000.70	5148.95	15810.80	16230.70	11011.35	9269.40
LJRandPropPEX	1.30	1.80	7699.85	7856.80	2784.85	1055.35
SumHeightPEX	19.90	7.90	4383.85	3522.60	701.30	0.15
SumHeightPropPEX	1.00	1.05	79.15	74.95	2.25	0.10
VarHeightPEX	817.40	12.80	2392.30	1782.90	16.50	0.20
VarHeightPropPEX	1.05	1.05	62.70	45.80	0.20	0.00

Table D.45. **Experiment 2, 10×10 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	405.35	369.35	69493.70	75595.60	23567.45	1315.30
CBASlackPropPEX	244.10	283.75	50461.15	68069.45	23585.10	1314.75
LJRandPEX	8728.95	51343.35	155630.20	147687.75	79254.35	50670.50
LJRandPropPEX	252.75	319.70	247499.80	237096.30	80877.10	25213.10
SumHeightPEX	482.45	355.05	55101.05	41659.80	8876.75	588.35
SumHeightPropPEX	243.90	281.70	5150.85	4632.35	804.45	617.40
VarHeightPEX	9674.45	420.10	33116.15	22062.40	833.20	596.50
VarHeightPropPEX	244.80	286.10	4191.20	3351.15	670.80	624.60

Table D.46. **Experiment 2, 10×10 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	13.00	11.40	4749.85	6711.80	2142.50	322.05
CBASlackPropPEX	1.15	7.55	3454.10	6109.60	2141.40	320.10
LJRandPEX	2156.50	11942.20	36165.85	36080.05	24356.05	20838.45
LJRandPropPEX	3.70	5.70	15414.30	15767.90	5651.45	2196.35
SumHeightPEX	39.75	18.40	8791.20	7111.90	1511.90	129.35
SumHeightPropPEX	1.00	5.70	177.40	211.60	111.30	126.30
VarHeightPEX	1634.70	27.30	4811.80	3637.65	144.50	128.90
VarHeightPropPEX	1.30	2.80	142.80	155.50	111.05	129.05

Table D.47. **Experiment 2, 10×10 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.32	1.62	7.96	14.33	23.72	40.03
CBASlackPropPEX	0.13	0.95	6.63	14.22	23.59	39.80
LJRandPEX	1.26	9.13	21.87	28.18	30.66	36.09
LJRandPropPEX	0.36	0.57	5.55	7.68	9.93	10.88
SumHeightPEX	0.71	2.95	13.06	16.85	18.58	22.30
SumHeightPropPEX	0.12	0.76	3.23	8.84	16.50	20.91
VarHeightPEX	0.86	3.06	11.29	16.98	18.98	22.00
VarHeightPropPEX	0.15	0.34	3.02	9.38	17.22	21.40

Table D.48. **Experiment 2, 10×10 Problems: Mean Percentage of Heuristic Commitments.**

D.2.4 15×15 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.15	0.80	0.90	0.60	0.15
CBASlackPropPEX	0.00	0.15	0.80	0.90	0.60	0.20
LJRandPEX	0.00	0.30	1.00	1.00	0.90	0.70
LJRandPropPEX	0.00	0.20	0.90	0.95	0.90	0.60
SumHeightPEX	0.00	0.05	0.65	0.95	0.75	0.25
SumHeightPropPEX	0.00	0.05	0.55	0.65	0.20	0.05
VarHeightPEX	0.00	0.05	0.70	0.85	0.75	0.20
VarHeightPropPEX	0.00	0.05	0.50	0.65	0.15	0.00

Table D.49. **Experiment 2, 15×15 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.32	182.07	1007.53	1080.90	765.66	196.06
CBASlackPropPEX	0.33	180.97	998.27	1081.20	767.11	255.80
LJRandPEX	0.31	401.37	1200.21	1200.20	1081.49	855.02
LJRandPropPEX	0.31	240.54	1107.13	1140.81	1086.56	755.97
SumHeightPEX	0.33	153.85	855.21	1140.42	901.31	366.11
SumHeightPropPEX	0.35	76.94	715.64	797.02	311.00	76.85
VarHeightPEX	0.34	77.65	883.33	1020.84	957.98	248.21
VarHeightPropPEX	0.34	65.66	665.92	847.35	344.10	19.66

Table D.50. **Experiment 2, 15X15 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.00	1098.50	6022.95	6545.30	3912.05	937.15
CBASlackPropPEX	1.00	1042.60	5363.50	6389.10	3891.60	1141.90
LJRandPEX	1.00	2050.50	5509.25	4836.00	3731.20	2532.85
LJRandPropPEX	1.00	818.95	3462.65	3370.55	3152.20	2084.00
SumHeightPEX	1.00	712.85	3873.55	4809.25	3433.40	1298.05
SumHeightPropPEX	1.00	236.20	1687.95	1893.95	746.60	160.65
VarHeightPEX	1.00	334.90	3807.05	4126.20	3485.40	836.40
VarHeightPropPEX	1.00	164.35	1397.25	1872.05	801.35	5.10

Table D.51. **Experiment 2, 15X15 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	441.35	40490.20	231427.10	234433.80	146874.05	34155.85
CBASlackPropPEX	441.35	36647.35	202072.25	230467.75	142733.25	42299.05
LJRandPEX	441.30	43435.75	134369.60	132362.10	78608.15	37853.10
LJRandPropPEX	441.30	39106.60	149928.55	156688.15	112701.40	74582.65
SumHeightPEX	441.30	16182.00	97825.65	82072.60	49897.65	22221.75
SumHeightPropPEX	441.30	11829.55	116011.30	127608.50	43035.90	9927.80
VarHeightPEX	441.40	10448.65	109392.30	78767.45	51101.25	15837.45
VarHeightPropPEX	441.40	10472.35	113698.30	146507.15	53385.55	2225.25

Table D.52. **Experiment 2, 15X15 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	2204.50	12120.50	13308.80	8371.10	2784.50
CBASlackPropPEX	0.00	2090.35	10801.00	12998.60	8334.65	3195.40
LJRandPEX	0.00	4442.05	11812.50	10152.95	7876.95	5536.65
LJRandPropPEX	0.00	1641.75	6960.60	6802.95	6401.40	4313.60
SumHeightPEX	0.00	1427.10	7788.50	9712.15	7018.55	2828.45
SumHeightPropPEX	0.00	473.25	3405.15	3893.45	1667.80	556.65
VarHeightPEX	0.00	670.75	7646.65	8351.25	7119.20	1917.80
VarHeightPropPEX	0.00	329.45	2823.60	3847.50	1773.65	260.20

Table D.53. **Experiment 2, 15×15 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	1.63	5.43	9.66	17.60	40.44
CBASlackPropPEX	0.00	1.16	5.12	9.58	17.87	40.18
LJRandPEX	0.00	4.40	10.54	8.30	13.01	20.22
LJRandPropPEX	0.00	0.94	4.56	4.73	6.39	7.14
SumHeightPEX	0.00	2.29	8.28	14.81	15.19	15.72
SumHeightPropPEX	0.00	0.64	3.00	4.80	8.51	12.61
VarHeightPEX	0.00	1.69	7.44	11.42	17.25	15.90
VarHeightPropPEX	0.00	0.45	2.36	4.55	7.58	13.49

Table D.54. **Experiment 2, 15×15 Problems: Mean Percentage of Heuristic Commitments.**

D.2.5 20×20 Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	0.45	0.95	0.95	0.85	0.40
CBASlackPropPEX	0.00	0.45	0.90	0.95	0.85	0.40
LJRandPEX	0.00	0.60	1.00	1.00	1.00	0.85
LJRandPropPEX	0.00	0.55	1.00	1.00	0.95	0.60
SumHeightPEX	0.00	0.40	0.90	1.00	0.95	0.80
SumHeightPropPEX	0.00	0.35	0.90	0.85	0.65	0.35
VarHeightPEX	0.00	0.35	0.90	1.00	0.90	0.65
VarHeightPropPEX	0.00	0.35	0.90	0.85	0.80	0.45

Table D.55. **Experiment 2, 20×20 Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.60	543.62	1143.61	1142.43	1040.73	532.27
CBASlackPropPEX	0.59	545.35	1127.01	1143.25	1041.33	540.15
LJRandPEX	0.61	745.84	1200.28	1200.67	1200.63	1026.55
LJRandPropPEX	0.61	676.87	1200.53	1200.61	1144.93	745.58
SumHeightPEX	0.60	489.23	1097.13	1200.36	1140.96	963.38
SumHeightPropPEX	0.61	453.85	1124.90	1044.49	814.83	530.57
VarHeightPEX	0.61	480.27	1102.54	1200.32	1082.03	787.52
VarHeightPropPEX	0.61	435.52	1098.31	1040.89	981.26	589.72

Table D.56. **Experiment 2, 20×20 Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	1.00	1227.55	2468.80	2431.75	1813.60	724.85
CBASlackPropPEX	1.00	918.35	1891.20	2322.40	1680.35	729.85
LJRandPEX	1.00	1468.45	1995.45	1626.35	1521.95	1162.20
LJRandPropPEX	1.00	821.20	1261.80	1163.65	1195.80	713.05
SumHeightPEX	1.00	1043.00	2087.55	1975.60	1555.45	1271.85
SumHeightPropPEX	1.00	415.25	959.55	870.85	640.40	355.05
VarHeightPEX	1.00	984.15	1963.15	1733.80	1444.00	898.50
VarHeightPropPEX	1.00	309.80	852.15	786.20	747.85	441.35

Table D.57. **Experiment 2, 20×20 Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	751.95	69423.25	156506.85	110694.30	86278.65	35485.30
CBASlackPropPEX	751.95	50891.05	107434.60	100777.05	81221.65	35069.65
LJRandPEX	751.85	49652.80	77090.30	58287.85	53803.40	35904.25
LJRandPropPEX	751.85	44110.65	63823.85	58345.95	57777.55	36166.15
SumHeightPEX	751.85	38346.05	84049.90	61836.55	38919.85	30840.90
SumHeightPropPEX	751.85	32340.45	76645.30	64654.95	53626.65	25412.00
VarHeightPEX	751.80	40047.70	95248.95	66817.25	54982.30	30530.00
VarHeightPropPEX	751.80	34196.50	77494.00	72304.35	63906.70	39376.75

Table D.58. **Experiment 2, 20×20 Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	2476.20	5152.55	5479.75	4977.25	3745.45
CBASlackPropPEX	0.00	1862.35	3997.20	5262.05	4716.30	3730.85
LJRandPEX	0.00	3129.50	4203.05	3515.65	3321.15	2686.05
LJRandPropPEX	0.00	1663.90	2597.10	2444.70	2561.40	1685.80
SumHeightPEX	0.00	2102.75	4260.20	4107.95	3331.90	2827.25
SumHeightPropPEX	0.00	849.00	2007.55	1909.85	1532.50	1038.10
VarHeightPEX	0.00	1984.10	4012.90	3616.45	3106.00	2109.65
VarHeightPropPEX	0.00	635.40	1790.60	1733.65	1700.20	1220.05

Table D.59. **Experiment 2, 20×20 Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.00	2.34	3.73	7.37	13.20	40.53
CBASlackPropPEX	0.00	1.94	3.63	7.51	13.37	39.98
LJRandPEX	0.00	4.35	6.40	7.51	8.39	9.63
LJRandPropPEX	0.00	2.58	4.20	4.35	4.79	6.11
SumHeightPEX	0.00	3.14	6.05	7.44	10.74	12.31
SumHeightPropPEX	0.00	1.52	2.74	3.78	5.11	7.52
VarHeightPEX	0.00	2.78	4.51	6.55	7.95	9.63
VarHeightPropPEX	0.00	1.06	2.39	3.04	3.70	7.80

Table D.60. **Experiment 2, 20×20 Problems: Mean Percentage of Heuristic Commitments.**

D.3 Experiment 3

D.3.1 Overall Results

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	0.92	0.91	0.90	0.92
CBASlackPropPEX	0.82	0.84	0.85	0.88
LJRandPEX	0.57	0.67	0.56	0.70
LJRandPropPEX	0.25	0.33	0.29	0.33
SumHeightPEX	0.27	0.30	0.26	0.33
SumHeightPropPEX	0.00	0.02	0.02	0.03
VarHeightPEX	0.35	0.43	0.42	0.37
VarHeightPropPEX	0.08	0.16	0.14	0.18

Table D.61. **Experiment 3 Overall: Fraction of Problems Timed-out.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	1102.65	1093.63	1086.51	1101.00
CBASlackPropPEX	985.59	1022.66	1028.03	1060.77
LJRandPEX	691.72	816.29	697.83	864.81
LJRandPropPEX	313.35	415.83	360.23	402.61
SumHeightPEX	334.19	370.67	329.34	423.62
SumHeightPropPEX	16.13	55.01	36.16	54.66
VarHeightPEX	434.55	515.69	509.80	446.23
VarHeightPropPEX	126.22	198.55	188.68	230.48

Table D.62. **Experiment 3 Overall: Mean CPU Time in Seconds.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	10063.45	7217.11	5816.05	5769.42
CBASlackPropPEX	7103.89	5482.73	4509.05	4527.57
LJRandPEX	4878.19	5225.35	4017.67	4606.58
LJRandPropPEX	1721.65	2121.72	1626.24	1729.18
SumHeightPEX	2678.84	2117.53	1610.11	1949.32
SumHeightPropPEX	55.55	174.58	89.35	128.93
VarHeightPEX	3589.64	3114.68	2549.19	2151.77
VarHeightPropPEX	698.05	879.28	720.72	824.03

Table D.63. **Experiment 3 Overall: Mean Number of Backtracks.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	116728.23	72898.35	62314.90	51164.23
CBASlackPropPEX	261910.83	222099.97	180997.09	188905.32
LJRandPEX	41478.40	45947.70	36662.81	41333.24
LJRandPropPEX	62688.65	87457.83	61562.55	68487.63
SumHeightPEX	15573.78	13255.09	9780.87	11306.79
SumHeightPropPEX	5895.03	19412.31	9807.72	15306.43
VarHeightPEX	23245.18	19283.14	15983.41	13459.43
VarHeightPropPEX	36628.22	44978.25	37939.99	41634.23

Table D.64. **Experiment 3 Overall: Mean Number of Commitments.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	20531.94	14748.95	12013.14	11871.96
CBASlackPropPEX	14323.97	11106.92	9208.02	9245.17
LJRandPEX	14479.85	15561.61	12160.99	13783.11
LJRandPropPEX	3513.08	4308.37	3324.34	3528.42
SumHeightPEX	5558.11	4445.97	3470.99	4142.54
SumHeightPropPEX	302.53	548.43	406.83	486.97
VarHeightPEX	7384.27	6440.63	5348.87	4559.77
VarHeightPropPEX	1586.91	1957.53	1676.12	1875.88

Table D.65. **Experiment 3 Overall: Mean Number of Heuristic Commitments.**

	Maximum Number of Alternative Process Plans			
Algorithm	**1**	**3**	**5**	**7**
CBASlackPEX	19.45	22.12	22.39	25.22
CBASlackPropPEX	5.83	5.13	6.78	5.75
LJRandPEX	30.22	29.47	27.80	29.37
LJRandPropPEX	7.47	6.40	6.61	6.17
SumHeightPEX	30.67	28.35	29.56	30.21
SumHeightPropPEX	19.50	16.88	17.20	16.36
VarHeightPEX	31.30	30.31	30.78	29.62
VarHeightPropPEX	19.61	16.73	17.73	16.21

Table D.66. **Experiment 3 Overall: Mean Percentage of Heuristic Commitments.**

D.3.2 One-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.65	1.00	1.00	1.00	1.00	0.85
CBASlackPropPEX	0.15	1.00	1.00	0.95	0.95	0.85
LJRandPEX	0.60	1.00	0.90	0.60	0.30	0.00
LJRandPropPEX	0.15	0.85	0.45	0.05	0.00	0.00
SumHeightPEX	0.25	0.85	0.45	0.05	0.00	0.00
SumHeightPropPEX	0.00	0.00	0.00	0.00	0.00	0.00
VarHeightPEX	0.60	0.90	0.60	0.00	0.00	0.00
VarHeightPropPEX	0.15	0.35	0.00	0.00	0.00	0.00

Table D.67. **Experiment 3, 1-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	780.19	1200.07	1200.06	1200.05	1200.06	1035.46
CBASlackPropPEX	180.48	1200.10	1200.09	1171.57	1140.25	1021.04
LJRandPEX	737.66	1200.18	1080.78	752.40	369.86	9.43
LJRandPropPEX	180.61	1062.95	563.44	64.66	4.02	4.44
SumHeightPEX	304.92	1057.60	574.71	62.64	2.62	2.64
SumHeightPropPEX	1.26	75.07	4.82	5.14	5.24	5.23
VarHeightPEX	773.31	1102.24	721.29	3.92	3.21	3.30
VarHeightPropPEX	191.87	538.91	9.75	5.50	5.66	5.61

Table D.68. **Experiment 3, 1-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	7108.05	10906.05	10843.25	10766.55	11172.85	9583.95
CBASlackPropPEX	1092.65	7516.70	8073.00	8540.60	8677.05	8723.35
LJRandPEX	6214.90	9335.85	7246.25	4032.40	2426.85	12.90
LJRandPropPEX	991.30	5746.80	3209.15	382.45	0.15	0.05
SumHeightPEX	2416.00	8523.80	4653.20	480.00	0.00	0.05
SumHeightPropPEX	2.80	330.30	0.15	0.05	0.00	0.00
VarHeightPEX	6718.30	9003.55	5808.65	7.35	0.00	0.00
VarHeightPropPEX	1188.85	2972.95	26.45	0.00	0.05	0.00

Table D.69. **Experiment 3, 1-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	53473.50	97805.70	115656.35	144438.60	147275.70	141719.55
CBASlackPropPEX	50373.20	400954.40	344725.75	274468.35	272631.15	228312.15
LJRandPEX	57109.05	83208.70	56110.70	31536.75	20040.25	864.95
LJRandPropPEX	35381.70	222738.70	103762.00	12331.60	973.55	944.35
SumHeightPEX	13664.30	50351.95	25316.55	2450.85	838.30	820.70
SumHeightPropPEX	812.20	30542.75	1223.00	1029.45	914.45	848.35
VarHeightPEX	49135.65	54356.35	33375.85	932.05	849.30	821.90
VarHeightPropPEX	53080.25	160965.30	2957.75	1025.55	898.50	842.00

Table D.70. **Experiment 3, 1-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	14500.05	22554.75	22302.20	21817.95	22607.15	19409.55
CBASlackPropPEX	2194.50	15110.50	16258.45	17225.50	17515.30	17639.55
LJRandPEX	18841.45	27597.35	21028.25	11973.15	7226.30	212.60
LJRandPropPEX	1987.85	11541.85	6493.45	859.25	97.95	98.10
SumHeightPEX	4848.45	17182.35	9509.40	1212.20	292.35	303.90
SumHeightPropPEX	13.95	760.35	200.50	250.95	286.60	302.85
VarHeightPEX	13479.90	18136.35	11819.45	271.60	291.60	306.70
VarHeightPropPEX	2386.50	6035.75	242.55	255.80	290.75	310.10

Table D.71. **Experiment 3, 1-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	18.01	25.57	21.92	19.07	17.76	14.35
CBASlackPropPEX	0.71	4.00	5.46	6.91	6.97	10.92
LJRandPEX	23.98	34.25	36.29	33.86	29.45	23.47
LJRandPropPEX	1.11	6.26	7.66	9.26	10.09	10.43
SumHeightPEX	12.64	34.24	33.69	31.29	34.99	37.14
SumHeightPropPEX	0.79	6.25	17.02	25.08	31.84	35.99
VarHeightPEX	19.26	34.51	32.77	29.45	34.42	37.42
VarHeightPropPEX	0.90	5.74	15.86	25.54	32.66	36.99

Table D.72. **Experiment 3, 1-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.3.3 Three-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.50	1.00	1.00	1.00	1.00	0.95
CBASlackPropPEX	0.30	1.00	0.95	0.95	1.00	0.85
LJRandPEX	0.60	1.00	0.90	0.75	0.55	0.20
LJRandPropPEX	0.30	0.80	0.55	0.30	0.05	0.00
SumHeightPEX	0.30	0.95	0.45	0.10	0.00	0.00
SumHeightPropPEX	0.00	0.15	0.00	0.00	0.00	0.00
VarHeightPEX	0.45	1.00	0.80	0.20	0.10	0.00
VarHeightPropPEX	0.30	0.60	0.05	0.00	0.00	0.00

Table D.73. **Experiment 3, 3-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	602.40	1200.06	1200.06	1200.06	1200.09	1159.11
CBASlackPropPEX	360.47	1200.09	1140.34	1141.57	1200.11	1093.36
LJRandPEX	720.29	1200.10	1113.07	951.61	663.11	249.58
LJRandPropPEX	360.45	963.76	684.03	409.54	72.64	4.53
SumHeightPEX	372.40	1154.57	559.88	129.05	4.12	4.00
SumHeightPropPEX	74.19	227.18	7.13	6.88	7.53	7.14
VarHeightPEX	560.44	1200.07	961.09	243.56	124.18	4.80
VarHeightPropPEX	360.44	738.01	70.12	7.22	7.62	7.88

Table D.74. **Experiment 3, 3-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	3712.95	7784.70	7853.25	7901.25	8029.40	8021.10
CBASlackPropPEX	1677.60	5591.25	5778.45	6262.00	6843.90	6743.20
LJRandPEX	4005.50	7381.00	7414.60	6477.60	4535.95	1537.45
LJRandPropPEX	1684.65	4805.20	3668.60	2131.75	439.65	0.45
SumHeightPEX	2153.75	6599.40	3279.30	672.05	0.50	0.15
SumHeightPropPEX	290.45	753.95	3.10	0.00	0.00	0.00
VarHeightPEX	3464.30	7064.75	5815.30	1512.90	830.85	0.00
VarHeightPropPEX	1688.60	3323.45	263.40	0.20	0.05	0.00

Table D.75. **Experiment 3, 3-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	30564.45	60238.70	76287.75	82579.85	82259.60	105459.75
CBASlackPropPEX	96186.95	349959.55	276655.70	215815.40	216777.30	177204.95
LJRandPEX	42842.15	70102.95	64242.60	53762.35	32725.60	12010.55
LJRandPropPEX	83037.25	233479.15	134139.45	66108.45	6865.60	1117.05
SumHeightPEX	14812.10	38261.65	19561.45	4926.50	992.70	976.15
SumHeightPropPEX	25499.35	85791.80	1806.05	1274.90	1084.70	1017.05
VarHeightPEX	24320.25	45347.70	31814.70	7897.60	5324.15	994.45
VarHeightPropPEX	84431.70	168999.15	13063.05	1255.95	1096.60	1023.05

Table D.76. **Experiment 3, 3-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	7604.60	15966.80	16036.25	16104.55	16361.25	16420.25
CBASlackPropPEX	3368.30	11281.20	11701.60	12697.90	13884.00	13708.50
LJRandPEX	12025.25	21649.50	22057.35	19740.35	12982.55	4914.65
LJRandPropPEX	3374.55	9655.45	7400.80	4346.65	974.00	98.80
SumHeightPEX	4323.95	13331.30	6771.55	1609.80	309.05	330.15
SumHeightPropPEX	586.75	1608.25	198.45	261.15	306.75	329.25
VarHeightPEX	6974.40	14267.25	11818.95	3290.65	1961.25	331.30
VarHeightPropPEX	3392.40	6750.20	713.05	254.10	303.50	331.95

Table D.77. **Experiment 3, 3-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	14.45	27.55	24.03	22.44	25.91	18.36
CBASlackPropPEX	1.11	3.45	4.76	6.21	6.91	8.34
LJRandPEX	16.95	31.25	34.53	34.50	32.26	27.35
LJRandPropPEX	1.23	4.82	6.57	7.94	8.96	8.89
SumHeightPEX	10.92	35.10	30.69	28.26	31.21	33.92
SumHeightPropPEX	0.59	5.55	12.60	21.12	28.77	32.67
VarHeightPEX	16.63	32.85	35.21	31.35	32.38	33.42
VarHeightPropPEX	1.23	5.77	11.88	20.82	28.07	32.59

Table D.78. **Experiment 3, 3-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.3.4 Five-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.60	1.00	1.00	1.00	0.95	0.85
CBASlackPropPEX	0.50	1.00	1.00	1.00	0.85	0.75
LJRandPEX	0.60	0.95	0.90	0.50	0.35	0.05
LJRandPropPEX	0.40	0.70	0.45	0.15	0.05	0.00
SumHeightPEX	0.45	0.75	0.35	0.00	0.00	0.00
SumHeightPropPEX	0.05	0.05	0.00	0.00	0.00	0.00
VarHeightPEX	0.65	0.95	0.75	0.15	0.00	0.00
VarHeightPropPEX	0.45	0.35	0.05	0.00	0.00	0.00

Table D.79. **Experiment 3, 5-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	756.09	1200.09	1200.09	1200.07	1140.80	1021.92
CBASlackPropPEX	600.42	1200.11	1200.13	1200.12	1063.50	903.91
LJRandPEX	766.92	1140.22	1080.55	661.63	439.63	98.05
LJRandPropPEX	491.14	849.79	560.99	189.75	64.73	4.98
SumHeightPEX	542.78	907.40	460.00	56.14	4.94	4.81
SumHeightPropPEX	99.90	84.08	7.60	8.26	8.66	8.43
VarHeightPEX	780.32	1140.38	902.51	209.80	19.78	6.00
VarHeightPropPEX	557.75	477.34	68.34	9.31	9.49	9.87

Table D.80. **Experiment 3, 5-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	4034.90	6045.20	6340.35	6459.35	6122.40	5894.10
CBASlackPropPEX	2336.35	4835.20	5045.65	5231.65	4948.95	4656.50
LJRandPEX	4550.90	6455.15	6140.40	3814.10	2500.55	644.95
LJRandPropPEX	2098.95	3778.00	2682.15	900.35	297.85	0.15
SumHeightPEX	2688.85	4289.30	2419.05	262.15	1.30	0.00
SumHeightPropPEX	292.05	243.90	0.15	0.00	0.00	0.00
VarHeightPEX	3950.85	5815.70	4495.35	968.80	64.45	0.00
VarHeightPropPEX	2282.15	1806.50	235.50	0.15	0.00	0.05

Table D.81. **Experiment 3, 5-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	31952.60	50329.00	62676.70	72047.15	81443.70	75440.25
CBASlackPropPEX	134723.40	260129.70	217460.10	188007.00	157854.95	127807.40
LJRandPEX	45014.60	63249.60	53790.45	32909.60	20438.40	4574.20
LJRandPropPEX	90614.70	168304.85	74303.85	25841.40	9098.85	1211.65
SumHeightPEX	15831.05	24405.35	13996.10	2262.95	1105.60	1084.15
SumHeightPropPEX	27907.10	25468.00	1706.10	1416.30	1220.45	1128.35
VarHeightPEX	28593.60	33323.05	25829.00	5729.90	1343.05	1081.85
VarHeightPropPEX	114129.75	101236.30	8559.95	1362.80	1211.60	1139.55

Table D.82. **Experiment 3, 5-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	8404.10	12545.45	13018.20	13372.95	12650.10	12088.05
CBASlackPropPEX	4710.70	9802.35	10281.55	10681.75	10162.30	9609.45
LJRandPEX	13850.45	19456.90	18374.50	11800.15	7548.80	1935.15
LJRandPropPEX	4216.00	7611.60	5440.95	1889.70	691.35	96.45
SumHeightPEX	5440.25	8761.60	5090.90	834.85	334.90	363.45
SumHeightPropPEX	616.65	623.85	226.65	288.15	332.85	352.85
VarHeightPEX	7991.55	11812.85	9224.60	2238.65	465.30	360.25
VarHeightPropPEX	4608.60	3745.25	716.15	296.90	331.55	358.30

Table D.83. **Experiment 3, 5-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	17.90	26.61	24.98	21.69	19.91	23.28
CBASlackPropPEX	1.95	4.18	5.17	6.16	9.53	13.70
LJRandPEX	20.39	31.19	33.31	31.71	27.88	22.31
LJRandPropPEX	2.71	5.70	7.89	7.45	7.88	8.02
SumHeightPEX	17.97	34.23	32.11	29.03	30.40	33.61
SumHeightPropPEX	2.16	6.75	13.77	21.31	27.75	31.49
VarHeightPEX	18.66	35.88	34.35	30.59	31.80	33.39
VarHeightPropPEX	2.41	6.41	15.98	22.39	27.64	31.58

Table D.84. **Experiment 3, 5-Alternative Problems: Mean Percentage of Heuristic Commitments.**

D.3.5 Seven-Alternative Results

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	0.65	1.00	1.00	1.00	1.00	0.85
CBASlackPropPEX	0.45	1.00	1.00	1.00	0.85	1.00
LJRandPEX	0.65	1.00	0.95	0.70	0.55	0.35
LJRandPropPEX	0.25	0.65	0.70	0.25	0.10	0.00
SumHeightPEX	0.40	0.80	0.45	0.35	0.00	0.00
SumHeightPropPEX	0.05	0.10	0.00	0.00	0.00	0.00
VarHeightPEX	0.65	0.85	0.50	0.15	0.05	0.00
VarHeightPropPEX	0.45	0.55	0.00	0.05	0.00	0.00

Table D.85. **Experiment 3, 7-Alternative Problems: Fraction of Problems Timed-out.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	780.35	1200.08	1200.11	1200.10	1200.10	1025.29
CBASlackPropPEX	540.55	1200.15	1200.11	1200.09	1023.60	1200.11
LJRandPEX	780.41	1200.18	1189.86	882.86	662.88	472.69
LJRandPropPEX	304.76	799.35	853.25	318.09	129.60	10.59
SumHeightPEX	522.36	998.61	585.32	424.11	5.31	6.02
SumHeightPropPEX	68.66	221.28	9.43	9.89	9.42	9.31
VarHeightPEX	780.35	1021.40	603.43	199.55	66.08	6.59
VarHeightPropPEX	540.54	687.50	36.12	97.45	10.53	10.73

Table D.86. **Experiment 3, 7-Alternative Problems: Mean CPU Time in Seconds.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	4003.20	6043.55	6131.85	6239.65	6600.15	5598.10
CBASlackPropPEX	2016.70	4726.60	4956.70	5017.85	4578.75	5868.80
LJRandPEX	4097.00	6328.70	6363.40	4629.55	3667.30	2553.55
LJRandPropPEX	1256.55	3526.55	3697.75	1294.75	568.20	31.25
SumHeightPEX	2280.45	4747.05	2687.75	1976.95	0.25	3.45
SumHeightPropPEX	197.10	570.45	3.45	2.55	0.05	0.00
VarHeightPEX	3841.25	4872.50	2931.85	962.60	302.40	0.00
VarHeightPropPEX	2111.65	2385.90	94.55	351.70	0.30	0.05

Table D.87. **Experiment 3, 7-Alternative Problems: Mean Number of Backtracks.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	29405.90	51237.00	52411.35	57251.40	58655.05	58024.70
CBASlackPropPEX	112137.05	260504.85	216587.20	201418.10	169214.15	173570.55
LJRandPEX	41865.40	60669.65	55974.85	39370.60	29581.90	20537.05
LJRandPropPEX	49740.45	160775.20	137559.50	47509.65	13842.40	1498.55
SumHeightPEX	13154.15	27757.40	13650.95	10956.55	1159.50	1162.20
SumHeightPropPEX	19718.55	65805.95	2020.80	1760.85	1310.60	1221.80
VarHeightPEX	24644.60	29211.70	15549.55	7008.50	3188.25	1153.95
VarHeightPropPEX	89923.55	140953.20	6392.95	10007.45	1292.15	1236.10

Table D.88. **Experiment 3, 7-Alternative Problems: Mean Number of Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	8285.70	12364.25	12719.90	12877.60	13476.20	11508.10
CBASlackPropPEX	4075.10	9591.20	10111.00	10267.85	9416.50	12009.35
LJRandPEX	12643.90	18770.45	18845.60	13661.65	10906.90	7870.15
LJRandPropPEX	2531.10	7107.95	7460.05	2683.00	1227.60	160.80
SumHeightPEX	4614.85	9673.70	5617.50	4243.45	337.35	368.40
SumHeightPropPEX	432.05	1270.55	228.75	297.30	335.75	357.40
VarHeightPEX	7773.65	9928.75	6119.80	2224.30	947.40	364.70
VarHeightPropPEX	4269.30	4890.10	410.05	985.30	339.50	361.00

Table D.89. **Experiment 3, 7-Alternative Problems: Mean Number of Heuristic Commitments.**

	Makespan Factor					
Algorithm	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
CBASlackPEX	19.16	26.64	28.11	26.10	28.52	22.80
CBASlackPropPEX	1.81	4.65	5.50	5.51	7.97	9.03
LJRandPEX	20.47	31.67	33.95	33.47	29.17	27.48
LJRandPropPEX	2.19	5.49	6.61	6.47	7.77	8.50
SumHeightPEX	19.81	33.80	34.22	32.43	29.19	31.81
SumHeightPropPEX	2.17	6.52	13.31	19.99	26.38	29.80
VarHeightPEX	21.22	33.51	33.07	28.55	29.65	31.75
VarHeightPropPEX	2.30	5.84	12.64	20.09	26.91	29.46

Table D.90. **Experiment 3, 7-Alternative Problems: Mean Percentage of Heuristic Commitments.**

Chapter 9 References

Adams, J., Balas, E., and Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156.

Baker, A. (1994). The hazards of fancy backtracking. In *Proceedings of AAAI-94*, pages 288–293.

Baker, K. (1974). *Introduction to sequencing and scheduling*. John Wiley and Sons.

Baptiste, P. and Le Pape, C. (1995). Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*.

Baptiste, P. and Le Pape, C. (1996). Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group*. Available from http://www.hds.utc.fr/ baptiste/.

Baptiste, P., Le Pape, C., and Nuijten, W. (1995a). Constraint-based optimization and approximation for job-shop scheduling. In *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95*.

Baptiste, P., Le Pape, C., and Nuijten, W. (1995b). Incorporating efficient operations research algorithms in constraint-based scheduling. In *Proceedings of the First Joint Workshop on Artificial Intelligence and Operations Research*. Workshop proceedings available on world wide web from http://www.cirl.uoregon.edu/aior/.

Beasley, J. E. (1990). OR-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072. Also available by ftp from ftp://graph.ms.ic.ac.uk/pub/paper.txt.

Beck, J. C., Davenport, A. J., Davis, E. D., and Fox, M. S. (1998). The ODO project: Toward a unified basis for constraint-directed scheduling. *Journal of Scheduling*, 1(2):89–125.

Beck, J. C., Davenport, A. J., and Fox, M. S. (1997a). Five pitfalls of empirical scheduling research. In Smolka, G., editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP97)*, pages 390–404. Springer-Verlag.

Beck, J. C., Davenport, A. J., Sitarski, E. M., and Fox, M. S. (1997b). Texture-based heuristics for scheduling revisited. In *Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. AAAI Press, Menlo Park, California.

Beck, J. C. and Jackson, K. (1997). Constrainedness and the phase transition in job shop scheduling. Technical report, School of Computing Science, Simon Fraser University.

Blazewicz, J., Domschke, W., and Pesch, E. (1996). The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1–33.

Bogart, K. P. (1988). *Discrete Mathematics*. D.C. Heath and Company, Lexington, Mass.

Brasel, H., Harborth, M., Tautenhahn, T., and Willenius, P. (1997). On the hardness ofthe classical job shop problem. Technical Report 23, Fakultat fur Mathematik, Otto-von-Guericke-Universitat Magdeburg, Postfach 4120, 39016 Magdeburg, Germany.

Brucker, P. and Thiele, O. (1996). A branch & bound method for the general-shop problems with sequence dependent set-up times. *OR Spektrum*, 18:145–161.

Burke, P. and Prosser, P. (1994). The distributed asynchronous scheduler. In Zweben, M. and Fox, M., editors, *Intelligent Scheduling*, chapter 11, pages 309–339. Morgan Kaufmann Publishers, San Francisco.

Carlier, J. and Pinson, E. (1989). An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176.

Carlier, J. and Pinson, E. (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161.

Caseau, Y. and Laburthe, F. (1994). Improved CLP scheduling with task intervals. In *Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press.

Caseau, Y. and Laburthe, F. (1995). Improving branch and bound for jobshop scheduling with constraint propagation. In *Proceedings of the Eighth Franco-Japanese Conference CCS'95*.

Caseau, Y. and Laburthe, F. (1996). Cumulative scheduling with task intervals. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press.

Cesta, A. and Oddi, A. (1996). Gaining efficiency and flexibility in the simple temporal problem. In Chittaro, L., Goodwin, S., Hamilton, H., and Montanari, A., editors, *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*, Los Alamitos, CA. IEEE Computer Society Press.

Cesta, A., Oddi, A., and Smith, S. F. (1998). Profile-based algorithms to solve multiple capacitated metric scheduling problems. In Simmons, R., Veloso, M., and Smith, S. F., editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 214–223, Menlo Park, CA. AAAI Press.

Cheng, C. C. and Smith, S. F. (1997). Applying constraint satisfaction techniques to job shop scheduling. *Annals of Operations Research, Special Volume on Scheduling: Theory and Practice*, 70:327–378.

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1994). Shortest paths algorithms: theory and experimental evaluation. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525.

Clements, D. P., Crawford, J. M., Joslin, D. E., Nemhauser, G. L., Puttlitz, M. E., and Savelsbergh, M. W. P. (1997). Heuristic optimization: A hybrid AI/OR approach. In *Proceedings of CP97 Workshop on Constraint-Directed Scheduling*.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. The MIT Press, Cambridge, Mass.

Collinot, A. and Le Pape, C. (1987). Controlling constraint propagation. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*.

Crawford, J. M. and Baker, A. B. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1092–1097.

Davenport, A. J., Beck, J. C., and Fox, M. S. (1999). An investigation into two approaches for resource allocation and scheduling. Technical report, Enterprise Integration Laboratory, Department of Mechanical and Industrial Engineering, University of Toronto.

Davis, E. D. (1994). ODO: A constraint-based scheduler founded on a unified problem solving model. Master's thesis, Enterprise Integration Laboratory, Department of Industrial Engineering, University of Toronto, 4 Taddle Creek Road, Toronto, Ontario M5S 3G9, Canada.

Davis, E. D. and Fox, M. S. (1993). Odo: A constraint-based scheduling shell. In *Proceedings of the IJCAI-93 Workshop on Production Planning, Scheduling and Control*.

DeBacker, B., Furnon, V., Kilby, P., Prosser, P., and Shaw, P. (1997). Local search in constraint programming: Application to the vehicle routing problem. In *CP97 Workshop on Industrial Constraint-Directed Scheduling*.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312.

Dechter, R., Dechter, A., and Pearl, J. (1990). Optimization in constraint networks. In Oliver, R. and Smith, J., editors, *Influence Diagrams, Belief Nets, and Decision Analysis*. John Wiley and Sons, Ltd, Chicester, England.

Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, 49:61–95.

Dorndorf, U. and Pesch, E. (1995). Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22(1):25–40.

Erschler, J., Roubellat, F., and Vernhes, J. P. (1976). Finding some essential characteristics of the feasible solutions for a scheduling problem. *Operations Research*, 24:772–782.

Erschler, J., Roubellat, F., and Vernhes, J. P. (1980). Characterising the set of feasible sequences for n jobs to be carried out on a single machine. *European Journal of Operational Research*, 4:189–194.

Fisher, H. and Thompson, G. L. (1963). Probabilistic learning combinations of local job-shop scheduling rules. In Muth, J. F. and Thompson, G. L., editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, New Jersey.

Fox, M. (1986). Observations on the role of constraints in problem solving. In *Proceedings of the Sixth Canadian Conference on Artificial Intelligence*.

Fox, M. S. (1983). *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Carnegie Mellon University, Intelligent Systems Laboratory, The Robotics Institute, Pittsburgh, PA. CMU-RI-TR-85-7.

Fox, M. S. (1990). Constraint-guided scheduling - a short history of research at CMU. *Computers in Industry*, 14:79–88.

Fox, M. S. (1999). personal communication. 1999.

Fox, M. S., Sadeh, N., and Baykan, C. (1989). Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 309–316.

Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the Association for Computing Machinery*, 21(11):958–966.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery*, 29(1):24–32.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Gaschnig, J. (1978). Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*.

Gent, I., MacIntyre, E., Prosser, P., Smith, B., and Walsh, T. (1996a). An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In Freuder, E., editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, pages 179–193. Springer-Verlag.

Gent, I. P., MacIntyre, E., Prosser, P., and Walsh, T. (1996b). The constrainedness of search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, volume 1, pages 246–252.

Gent, I. P. and Walsh, T. (1994). The hardest random SAT problems. In Nebel, B. and Dreschler-Fischer, L., editors, *Proceedings of KI-94: Advances in Artificial Intelligence. 18th German Annual Conference on Artificial Intelligence*, pages 355–366. Springer-Verlag.

Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46.

Glover, F. (1989). Tabu search part I. *Operations Research Society of America (ORSA) Journal on Computing*, 1(3):109–206.

Glover, F. (1990). Tabu search part II. *Operations Research Society of America (ORSA) Journal on Computing*, 2(1):4–32.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, Mass.

Gomes, C. P., Selman, B., and Kautz, H. (1998). Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437.

Haralick, R. M. and Elliot, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence Journal*, 14:263–314.

Harvey, W. D. (1995). *Nonsystematic backtracking search*. PhD thesis, Department of Computer Science, Stanford University.

Harvey, W. D. and Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference onf Artificial Intelligence (IJCAI-95)*, pages 607–613.

Havens, W. S. (1997). Nogood caching for multiagent backtrack search. In *Proceedings of the AAAI-97 Constraints and Agents Workshop*, Providence, Rhode Island.

Herroelen, W. S. and Demeulemeester, E. L. (1995). Recent advances in branch-and-bound procedures for resource-constrained project scheduling problems. In Chretienne, P., Coffman Jr., E., Lenstra, J., and Liu, Z., editors, *Scheduling theory and it applications*, chapter 12. John Wiley & Sons, Ltd.

Hildum, D. W. (1994). *Flexibility in a knowledge-based system for solving dynamic resource-constrained scheduling problems*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, MA. 01003-4610. UMass CMPSCI TR 94-77.

Hogg, T. and Williams, C. P. (1994). The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377.

Holland, J. (1975). *Adaptation in natural systems*. University of Michigan Press, Ann Arbor, Mich.

Hooker, J. N. (1996). Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42.

Hooker, J. N. and Vinay, V. (1995). Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383.

Johnston, M. D. (1990). Spike: AI scheduling for NASA's hubble space telescope. In *Proceedings of the 6th IEEE Conference on AI Applications*, pages 184–190. IEEE Computer Society Press, Los Alamitos, California.

Johnston, M. D. and Minton, S. (1994). Analysing a heuristic strategy for constraint satisfaction scheduling. In Zweben, M. and Fox, M., editors, *Intelligent Scheduling*, chapter 9, pages 257–289. Morgan Kaufmann Publishers, San Francisco.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

Korf, R. (1996). Improved limited discrepancy search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*.

Kott, A. and Saks, V. (1998). A multi-decompositional approach to integration of planning and scheduling – an applied perspective. In *Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*, Pittsburgh, USA.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, pages 32–44.

Laarhoven, P. J. M., Aarts, E. H. L., and Lenstra, J. K. (1992). Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125.

Lawrence, S. (1984). *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement)*. PhD thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Le Pape, C. (1994a). Constraint-based programming for scheduling: An historical perspective. In *Working Papers of the Operations Research Seminar on Constraint Handling Techniques*.

Le Pape, C. (1994b). Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66.

Le Pape, C. (1994c). Using a constraint-based scheduling library to solve a specific scheduling problem. In *Proceedings of the AAAI-SIGMAN Workshop on Artificial Intelligence Approaches to Modelling and Scheduling Manufacturing Processes*.

Le Pape, C. and Baptiste, P. (1996). Constraint propagation techniques for disjunctive scheduling: the preemptive case. In *Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96)*.

Le Pape, C. and Baptiste, P. (1997). An experimental comparison of constraint-based algorithms for the preemptive job shop scheduling problem. In *CP97 Workshop on Industrial Constraint-Directed Scheduling*.

Le Pape, C., Couronné, P., Vergamini, D., and Gosselin, V. (1994). Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the UK Planning Special Interest Group*.

LePape, C. and Smith, S. F. (1987). Management of temporal constraints for factory scheduling. In Rolland, C., Leonard, M., and Bodart, F., editors, *Proceedings IFIP TC 8/WG 8.1 Conference on Temporal Aspects in Information Systems (TAIS 87)*. Elsevier Science Publishers.

Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 1, pages 232–238.

Little, J., Murty, K., Sweeney, D., and Karel, C. (1963). An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989.

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

Minton, S., Johnston, M., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205.

Muscettola, N. (1992). Scheduling by iterative partition of bottleneck conflicts. Technical Report CMU-RI-TR-92-05, The Robotics Institute, Carnegie Mellon University.

Muscettola, N. (1994). On the utility of bottleneck reasoning for scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1105–1110.

Navinchandra, D. (1991). *Exploration and innovation in design*. Springer-Verlag, New York.

Navinchandra, D. and Marks, D. H. (1987). Design exploration through constraint relaxation. In *Expert Systems in Computer-Aided Design*. Elsevier Science Publishers B. V.

Neiman, D., Hildum, D., Lesser, V., and Sandholm, T. (1994). Exploiting meta-level information in a distributed scheduling system. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 394–400,, Seattle, WA.

Nowicki, E. and Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813.

Nuijten, W. (1999). personal communication. 1999.

Nuijten, W. P. M. (1994). *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.

Nuijten, W. P. M. and Aarts, E. H. L. (1997). A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*. To appear.

Nuijten, W. P. M., Aarts, E. H. L., van Arp Taalman Kip, D. A. A., and van Hee, K. M. (1993). Randomized constraint satisfaction for job shop scheduling. In *Proceedings of the IJCAI'93 Workshop on Knowledge-Based Production, Scheduling and Control*, pages 251–262.

Oddi, A. and Smith, S. F. (1997). Stochastic procedures for generating feasible schedules. In *Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. AAAI Press, Menlo Park, California.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.

Roberts, F. (1984). *Applied combinatorics*. Prentice Hall, Engelwood Cliffs, New Jersey.

Sadeh, N. (1991). *Lookahead techniques for micro-opportunistic job-shop scheduling*. PhD thesis, Carnegie-Mellon University. CMU-CS-91-102.

Sadeh, N. (1994). Micro-opportunistic scheduling. In Zweben, M. and Fox, M. S., editors, *Intelligent Scheduling*, chapter 4, pages 99–138. Morgan Kaufmann Publishers, San Francisco.

Sadeh, N. and Fox, M. S. (1989). Focus of attention in an activity-based scheduler. In *Proceedings of the NASA Conference on Space Telerobotics*.

Sadeh, N. and Fox, M. S. (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence Journal*, 86(1).

Sadeh, N., Sycara, K., and Xiong, Y. (1995). Backtracking techniques for the job shop scheduling constraint satisfaction. *Artificial Intelligence*, 76:455–480.

Sadeh, N. M., Hildum, D. W., Laliberty, T. J., McA'Nulty, J., Kjenstad, D., and Tseng, A. (1998). A blackboard architecture for integrating process planning and production scheduling. *Concurrent Engineering: Research and Applications*, 6(2).

Saks, V. (1992). Distribution planner overview. Technical report, Carnegie Group, Inc., Pittsburgh, PA, 15222.

Saks, V., Johnson, I., and Fox, M. S. (1993). Distribution planning: A constrained heuristic search approach. In *Proceedings of the Knowledge-based System and Robotics Workshop*, pages 13–19. Industry Canada.

Selman, B., Levesque, H., and Mitchell, D. G. (1992). A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446. AAAI Press/MIT Press.

Simon, H. A. (1973). The structure of ill-structured problems. *Artificial Intelligence*, 4:181–200.

Simonis, H. and Cornelissens, T. (1995). Modelling producer/consumer constraints. In Montanari, U. and Rossi, F., editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP95)*, pages 449–462. Springer-Verlag.

Smith, B. M. and Grant, S. A. (1997). Trying harder to fail first. Technical Report 97.45, School of Computer Science, University of Leeds.

Smith, S. F.and Ow, P. S., Matthys, D. C., and Potvin, J. Y. (1989). OPIS: An opportunistic factory scheduling system. In *Proceedings of International Symposium for Computer Scientists*.

Smith, S. F. and Cheng, C. C. (1993). Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 139–144.

Stallman, R. and Sussman, G. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196.

Sycara, K., Roth, S., Sadeh, N., and Fox, M. S. (1991). Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21(6):1446–1461.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285.

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

Vaessens, R. J. M., Aarts, E. H. L., and Lenstra, J. K. (1994). Job shop scheduling by local search. Technical Report COSOR Memorandum 94-05, Eindhoven University of Technology. Submitted for publication in INFORMS Journal on Computing.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. MIT Press.

Wagner, T., Garvey, A., and Lesser, V. (1997). Design-to-criteria scheduling: Managing complexity through goal-directed satisficing. In *Proceedings of the AAAI-97 Workshop on Building Resource-Bounded Reasoning Systems*.

Wagner, T., Garvey, A., and Lesser, V. (1998). Criteria directed task scheduling. *International Journal of Approximate Reasoning*, 19:91–118.

Zweben, M., Daun, B., Davis, E., and Deale, M. (1994). Scheduling and rescheduling with iterative repair. In Zweben, M. and Fox, M. S., editors, *Intelligent Scheduling*, chapter 8, pages 241–256. Morgan Kaufmann Publishers, San Francisco.

Zweben, M., Davis, E., Daun, B., and Deale, M. (1993). Informedness vs. computational cost of heuristics in iterative repair scheduling. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1416–1422.