# ODO: A CONSTRAINT-BASED SCHEDULER FOUNDED ON A UNIFIED PROBLEM SOLVING MODEL

Eugene Donald Davis

TR-EIL-94-1

Enterprise Integration Laboratory
Dept. of Industrial Engineering
4 Taddle Creek Road
University of Toronto
Toronto, Ontario, CANADA
M5S 1A4

Tel:+1(416)978-6823 Fax:+1(416)978-3453
Internet:eil@ie.utoronto.ca

# ODO: A CONSTRAINT-BASED SCHEDULER FOUNDED ON A UNIFIED PROBLEM SOLVING MODEL

by

Eugene Donald Davis

A thesis submitted in conformity with the requirements

for the degree of Master of Science

Graduate Department of Computer Science

University of Toronto

# Abstract

ODO: a constraint-based scheduler founded on

a unified problem solving model

Eugene Donald Davis

Master of Science

Department of Computer Science

University of Toronto

1994

We propose a unified model for constraint-based scheduling. This model formulates all problem solving as an incremental search process, where each step successively asserts a new commitment, propagates constraints, and releases commitments if the resulting problem state was deemed undesirable. Decisions are based upon constraint graph measurements called textures. Both constructive and repair-based search mechanisms can be represented in this framework.

We have implemented a constraint-based scheduling system, ODO, which performs search using this model. A scheduling heuristic is realized within ODO by specifying the appropriate model parameters at run-time. ODO provides a texture-based language for the declaration of these parameters. Since all problem solving adheres to our model and all decisions are based upon constraint graph measures, ODO serves as a basis for focused empirical analysis of heuristic scheduling methods. Preliminary experiments demonstrate the sensitivity of scheduling performance to model parameter adjustments.

# Acknowledgements

I would first and foremost like to thank my wife, Holly, for her patience and support. I am grateful to my supervisor, Mark Fox, for providing guidance and encouragement over the past sixteen months. Also, many thanks to my second reader, Hector Levesque, for his useful comments.

Monte Zweben, Chris Beck, Karen Reid, and Mark Ringer all provided invaluable comments on earlier drafts, and I am indebted to Monte, Chris, Hector, Steve Minton, Mark Drummond, and David Mitchell for their suggestions regarding this research. Finally, I'd like to express my appreciation for the financial assistance provided by the Department of Computer Science and the University of Toronto.

# Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

The work in this thesis was motivated by observation of the current trends in constraint-based scheduling. Many different constraint-based scheduling systems now exist and have been applied successfully to various application domains. However, most published results have reported only the performance of a particular system's heuristics on a particular domain; we know of little comparative analysis suggesting when or why one approach might be preferred over another. Although the various heuristics differ, they all share a common problem representation (a constraint graph), as well as many problem-solving components.

This combined evidence suggests that there is an unexplored opportunity to isolate the differentiating components of both the problem and the problem solver, and to correlate these components with search performance. With this goal in mind we constructed a unified model of search that captures many known scheduling heuristics. We then implemented ODO, a constraint-based scheduling system founded on our problem solving model.

We have performed initial experiments that have provided insight into some of the relations between problem structure and efficient search methods. This thesis presents our model of problem solving, describes ODO's current implementation of the model, and presents experimental results.

## 1.1 Job Shop Scheduling

The term *scheduling* is used to describe a general class of problems primarily concerned with the allocation of activities to resources over time. One particular scheduling problem class is *job shop scheduling*, which is representative of many scheduling problems encountered in the manufacturing domain. The job shop scheduling problem is described with:

- a set of unit-capacity *resources*
- a set of *jobs*; each job consists of a set of ordered *activities*; each activity has positive *duration* and specifies a particular resource
- a positive *due date*.

where execution times are assigned to activities such that:

- the activities within each job are ordered properly
- no two activities requiring the same resource are assigned overlapping execution times, and
- no activity starts before time 0 or completes after the due date.

For instance, given:

- three resources, R1, R2, and R3

- two jobs, A and B such that each job consists two activities (A1, A2, B1, B2) and all activity durations = 2; resource specifications for each activity are: A1, R1; A2, R2; B1, R3; B2, R2
- a due date of 6

there are several satisfying solutions, one of which we show in Figure 1. (The chart in Figure 1 is referred to as a *Gantt chart* [Baker 92] ; it displays for each resource the execution times of activities requiring that resource.)

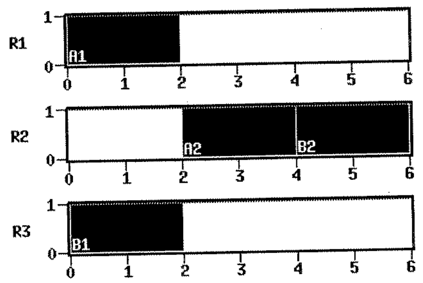

Figure 1. Gantt chart of a satisfying solution to the example problem.

Although there exist polynomial algorithms for solving restricted versions of the job shop problem, the general case is known to be NP-complete [Garey 79] . We might still hope that the practical versions of this problem might yield solutions to some polynomial algorithms, but to date no algorithms are known.

In some scheduling domains it is important not only to find a solution that orders jobs properly and ensures that no resource is overallocated, but also to optimize some other criteria. For example, if inventory is modeled, then schedules which maintain lower inventory levels may be preferred [Fox 87] .

## 1.2 Constraint-based Problem Solving

*Constraint satisfaction* describes a general method for representing and solving a problem. In this approach, the problem is represented by a constraint graph, where the nodes are the variables of the problem and the arcs are the constraints between the variables [Mackworth 86] . Constraints restrict the set of acceptable values for the variables to which they are attached. Solving a *constraint satisfaction problem* (CSP) amounts to assigning values to variables such that all constraints are satisfied.

A constraint-based representation for a problem may be chosen for several reasons. First, a problem may have a natural mapping into CSP's language of variables and constraints. Second, variations of the problem may be easily created by adding or deleting variables and constraints; this may be of particular importance in dynamic domains or when it is necessary to solve several versions of the problem in greater or lesser detail.

Perhaps the most important reason to use a constraint-based representation is to exploit its graph-based structure during problem solving. The constraint graph enables a powerful mechanism called *consistency enforcement*: when a variable's value or domain is changed, that change can *propagate* through the constraint graph and alter actual or possible values on other variables [Mackworth 77] . For CSP's in general, many consistency enforcing techniques and search mechanisms have been developed [Bitner 75] [Haralick 80] [Gaschnig 77] [Davis 87] . A CSP may even exhibit a particular graph structure that guarantees a solution in polynomial time [Freuder 82] . When no known polynomial algorithm exists, the constraint-based problem structure can be utilized by domain-specific heuristics during search or propagation [Davis 87] [Waltz 75] .

When search is required for CSP's, it often involves an incremental process of assigning a tentative value to a variable, followed by some form of consistency enforcement. Many incremental search heuristics focus on general overall strategies for variable and value selection that will efficiently lead to a solution. *Constructive* and *repair-based* search methods are two such examples. In the constructive (also known as *backtracking*) approach, an assignment is made to a variable only if it is consistent with all previous assignments; backtracking occurs when it is determined that no complete solution can be obtained with the current assignments. In the repair-based method, search begins with a problem state where all variables have assignments (even if those assignments are inconsistent with problem constraints); search is performed by repeatedly changing assignments on variables (making "repairs") in an attempt to reduce the total number of inconsistencies. Like constructive methods which can backtrack, repair-based methods may revert to a previous search state.

## 1.3 Constraint-based Scheduling

In recent years the constraint-based approach to scheduling has been gaining attention in both research and application [Sadeh 91] [Minton 92] [Zweben 94] [Keng 89] [Le 91] [Smith 87] [Fox 87] . The previously mentioned reasons for the appeal of the CSP model — the convenience of the representation, the availability of problem-solving tools, and the ability to exploit the problem structure — apply equally well to scheduling problems. Recent real-world successes [Deale 94] [Johnston 89] [Saks 93] further fuel interest in the constraint-based scheduling methodology.

A job shop scheduling problem can be readily converted into a constraint satisfaction problem. The variables are activity execution times and the constraints are restrictions placed on the execution times. Constraint types are typically either temporal (e.g., ensuring that one activity only executes later in time than another activity) or resource-oriented (e.g., not allowing two activities to use the same resource simultaneously). For problems that prefer some solutions over others, additional constraints can be created to reflect the desired criteria. This type of *optimization* constraint may never be completely satisfied in feasible solutions, but the constraint itself can still guide search. CSP's which contain optimization constraints are sometimes referred to as *constrained optimization problems* [Sadeh 91] .

Interestingly, several competing approaches have been developed for solving a scheduling problem represented as a CSP. Distinctions in method range from different variable and value ordering heuristics to completely different search mechanisms. While both constructive and repair-based approaches have been used successfully in certain scheduling domains, little is understood as to when one method might be preferred over another. And while it is clear that consistency enforcement is an important component in both search mechanisms, there exists no consensus on how much enforcement is best for a given problem class. The same is true for the many variable and value selection strategies.

Beyond the fact that we do not fully understand which search parameters are most appropriate for a given problem, there are other aspects of the problem solving process for which all known methods can be improved upon. One of these is the criterion for terminating search. Search typically halts if a solution has not been found within a certain number of pre-specified iterations [Zweben 94] [Minton 92] . We assert that some other measures exist that reflect the current search method's likelihood for finding a solution in a reasonable amount of time. An accurate termination condition is clearly useful for repair-based methods, since those methods are incomplete. But it would also be of value to complete search strategies, since we cannot always afford to wait until either a solution is found or the search space is exhausted.

## 1.4 Unifying Constraint-based Scheduling

We believe that there is an opportunity to discover some of the associations between problem structure and heuristic performance in constraint-based scheduling. If we can find these relationships, we can then postulate when one search heuristic is preferred over another. In addition, we should be able to create new heuristics that refine certain aspects of the search process (such as the search termination criteria). Our approach is to view the representation and solution of a constraint-based scheduling problem from a unified model that combines common components and isolates essential differences.

### 1.4.1 A Common Problem Representation

If we employ a constraint-based representation in our model, then all problems will be formulated as constraint graphs. We propose that graph property measurements can characterize heuristic search performance. It has already been shown that the *width* of a constraint graph determines how much consistency enforcement is required to guarantee backtrack-free search [Freuder 82] . In [Fox 89] the authors introduce *textures* as properties of a constraint graph that guide heuristic decision making. This concept of texture clarifies the separation between a heuristic decision and the constraint graph information (i.e. the texture measurement) that went into making that decision. In [Fox 89] the authors describe some texture measures and demonstrate the relations between those measures to some common heuristics.

By using the constraint representation, we can exploit existing research into correlations between graph properties and heuristic performance. We will use the term texture to refer to constraint graph measurements we perform within our model.

### 1.4.2 A Common Problem Solving Model

We have noted a pattern to the way many constraint-based schedulers perform incremental search. At each step, a modification is chosen — typically the assignment of a value to a variable. Once this modification has been asserted, constraint propagation is performed. Propagation may assign new values to other variables, or it may further restrict the set of possible values for unassigned variables. After propagation, the resulting state can be retracted if an undesirable or dead-end state is encountered. This incremental process repeats until some termination conditions are met.

Although the primary modification operator is the assignment of a value to a variable, other modifications are sometimes used to traverse the search space [Muscettola 93] . The intent of these modifications in general is to assert some tentative *commitment* within the resulting problem-solving state. We propose that the loop presented in Figure 2 can be used as a template to describe many heuristic search mechanisms. Each heuristic is characterized by what type of commitments are allowed, how commitments are chosen, how much constraint propagation is performed, how to release a commitment, and what the acceptance and termination criteria are. We define a *policy* as a specification of the exact manner in which each component of this search loop is to be performed. Decisions made within a policy are based upon texture measures.

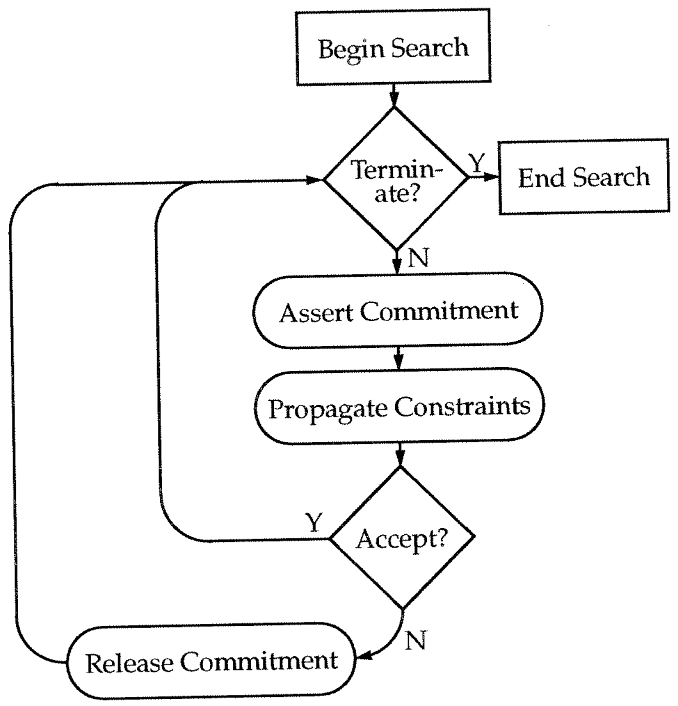

Figure 2. A flow diagram of the general search loop.

Given common models for problem representation and problem solving, we have an opportunity to associate heuristic performance of the problem on a given solver with the properties that make the problem and/or solver unique.

## 1.5 ODO: An Implementation of the Unified Model

ODO is a constraint-based scheduling system that implements our unified model. ODO represents the scheduling problem as a constraint graph, and performs search using the loop shown in Figure 2. ODO accepts as input both the problem to be solved and the parameters for which search is to be performed. In this sense, ODO is a scheduler "interpreter", since heuristics are constructed at run-time with the input language.

Using ODO we have reconstructed several well-known scheduling heuristics of both the constructive and repair-based variety. We have conducted experiments to verify ODO's competence at emulating these and related heuristic variants. Our results demonstrate a heuristic's sensitivity to the exact amount of propagation being performed, and show the value of lookahead to some repair heuristics.

## 1.6 Related Work

ODO is clearly related to many constraint-based schedulers, but represents a generalization of their approach to search with its unified model. ODO's actual constraint representation has been heavily influenced by that found in GERRY [Zweben 92] .

ODO's principles of providing the user with a language for use within a constraint-based problem solver can be found in CHIP [Van 84] and MULTI-TAC [Minton 93] . CHIP extends logic programming to reason explicitly about finite-domain variables and constraint-based consistency techniques. MULTI-TAC accepts as input a description of a combinatorial problem along with a sample set of problem instances, and generates an appropriate problem-solver. Both CHIP and MULTI-TAC currently only attempt to solve problems with constructive approaches, although the designers of MULTI-TAC plan to add an iterative component in the near future.

GERRY has a declarative rule system for constructing schedules [Zweben 89] . Its rules perform variable and value selection, constraint propagation, and backtracking. The rule system has not been generalized to incorporate repair-based search.

## 1.7 Thesis Contributions

The contributions of this thesis are threefold:

1. We describe a unified mechanism for constraint-based problem solving in the context of scheduling. This mechanism provides a model for search in a manner that represents a large set of heuristics of both the constructive and repair-based variety.
2. We have created a constraint-based scheduling system that embodies the above-mentioned problem solving model. With this system, we demonstrate the ability to emulate many state-of-the-art scheduling systems and show how alternative heuristics can be designed.
3. We empirically demonstrate the utility of such a system by presenting experimental results. These results demonstrate the sensitivity of some known scheduling heuristics to adjustments in their search parameters.

## 1.8 Thesis Organization

In Chapter 2 we review background material relevant to the thesis. This covers constraint-based problem solving, the scheduling problem in general, and the constraint perspective as applied to scheduling problems. Chapter 3 introduces our model of constraint-based scheduling as search through the space of commitments. We show how commitment-based search can be further enhanced with the addition of problem reformulation techniques.

The next three chapters of the thesis describe ODO and our use of ODO to explore our problem-solving paradigm. Chapter 4 outlines ODO's problem representation and describes the language for declaring a scheduling problem. In Chapter 5 we describe ODO's problem solving mechanism and how it is used. Chapter 6 presents the results of experiments that demonstrate the competence of ODO's problem solving model.

In Chapter 7 we summarize, provide conclusions, and outline plans for future work.

# Chapter 2 Background

In this chapter we review the basic principles of scheduling and constraint-based reasoning. We then review relevant research in constraint-based scheduling. This background information will set the context for later chapters, where we describe our particular constraint-based scheduling model.

## 2.1 Job Shop Scheduling

### 2.1.1 Problem Description

This thesis focuses on the job shop scheduling problem class as described in Chapter 1. This was presented as a decision problem. Practically speaking however, we not only want to know if a satisfying schedule exists, but also what that schedule is. This requirement does not in general add to the complexity of the problem, since it is often the case that the only way to know that a schedule exists is to actually find one.

Not all scheduling problems fit into the strict job shop description. However, we can extend the basic definition in a number of ways and still preserve much of the character of the job shop problem. Here we outline a few of the more common extensions:

- **Hierarchy of tasks**: The standard job shop scheduling problem described entities referred to as *jobs*, which were composed of *activities*. One generalization of this is to allow for an arbitrary hierarchy of tasks. Jobs therefore may have parent jobs, and so on. So that we do not have to worry about distinguishing between jobs and activities, we often refer to any job or activity simply as a *task*.
- **Non-unit resource capacity**: We remove the restriction that the capacity for resources be = 1. Some resources may be able to process more than one activity at a time. We can now think of a group of human workers as a resource whose capacity is the number of workers in the group. This group of workers is then a *pool* of individual resources that can be drawn upon. All members of such a pool are considered equivalent. We will often refer to resources as the more general notion of resource-pools.
- **Resource class/pool hierarchy**: We define a resource pool as an actual resource entity for use during scheduling. We use resource *classes* to define the functionality of each resource pool. Resource pools have associated with them one or more resource classes. For example, we might define a "Lathe" resource class: any resource pool that belongs to this class can perform lathing operations. When an activity requires a resource during execution, it needs to only specify the class of resource required. Any valid resource pool should be able to service that requirement.
- **Non-unit resource-requests**: Individual activities may require more than one unit of a particular resource class. For example, the task of transporting a palette may require two workers.

- **Multiple resource-requests**: Individual activities may require more than one type of resource class. For example, a lathing operation may require a lathe and a lathe operator. Activities are also allowed to make *no* requests for resources.
- **General precedence relations among tasks within job**: We remove the restriction that all tasks in a job must have a fully ordered sequence. Tasks for a job may have only partial or no sequencing.
- **Precedence relations between jobs**: We can specify that one job must complete before another begins.
- **Unique earliest-start/ latest-finish dates for each job**: Instead of all jobs having earliest starts of 0 and identical due dates, specify an earliest-start and latest-finish for each job (or even each activity).
- **Relaxable due date:** We allow for some activities to be scheduled beyond the due date. Schedules which minimize the tardiness of these activities are generally preferred.

In this thesis our discussion will focus on the basic job shop problem; however, it is useful to think ahead about how our problem solving model will generalize when the problem class grows.

## 2.1.2 Non-constraint-based Approaches to Job Shop Scheduling

Many non-constraint-based problem solving methods have been developed that apply to job shop scheduling problems. These approaches vary from finding solutions to large mathematical models to merely executing some dispatch rules. Here we review some of these methods.

### 2.1.2.1 Linear/Integer Programming

Operations Research approaches to scheduling traditionally involve representing the problem using a series of equations and an objective function. If no variables in the equations are restricted to integer values, then a linear programming method will find the optimal solution (i.e., one that maximizes or minimizes the objective function) in polynomial time [Hillier 90] .

When scheduling problems are represented in this manner, many variables require integer values. Integer programming techniques are then necessary, and in the worst case these require exponential computation. Software packages such as CPLEX [CPLEX 92] perform integer programming search by assigning integer values to one variable and solving the rest of the problem as a linear-programming relaxation. This *branch and bound* [Hillier 90] type of approach can solve problems efficiently; however, there is an art to representing the problem appropriately, which is further complicated by the non-intuitive mapping of activities and resources to variables and equations.

### 2.1.2.2 Bottleneck Analysis

Practitioners have observed that in many scheduling domains only part of the problem may be difficult to solve, while other parts do not require as much attention. Specifically, those resources that are maximally utilized by activities over given time periods typically have the greatest impact on the quality of the overall schedule. These *bottleneck* resources then become a focus for heuristic problem solvers. Non-constraint-based procedures that reason explicitly about bottlenecks include the Shifting Bottleneck Procedure [Adams 88] and OPT [Goldratt 90] . The Shifting Bottleneck Procedure solves one-resource relaxations of the original problem, and uses this information to identify bottlenecks for subsequent sequencing decisions. OPT's approach is to schedule activities so that bottleneck resources are maximally utilized at all times.

#### 2.1.2.3 Priority Dispatch Rules

Another scheduling method often encountered in practice involves the use of simple priority dispatch rules [Baker 92]. These rules order schedulable activities by some simple criteria (such as the duration of the task), then assign each activity in turn to the earliest available time slot. Dispatch rules are not likely to generate schedules of the quality found by more informed techniques, but the results may be acceptable to the application.

## 2.2 Constraint-based Problem Solving

Many problems can be represented and solved in a constraint-based framework. Here we briefly review the general constraint representation and some of the problem solving methods.

### 2.2.1 Problem Representation

A constraint model is represented by a set of variables and a set of constraints. Each variable has a domain of possible values; each constraint is attached to some subset of the variables, restricting the set of mutually compatible values on those variables.

When we discuss the constraint satisfaction problem, we refer to the simplest class of constraint problems: all variables and constraints are predefined, all domains are finite, and only one satisfying solution is required. There are many variants to this original problem specification:

- New variables and constraints can be added during problem-solving [Mittal 88].
- All solutions are to be found instead of just one [Nadel 89].
- A solution is to be found that satisfies as many constraints as possible [Zweben 94] [Johnston 89] [Freuder 92]
- Optimization constraints are added [Sadeh 91] [Zweben 94].

While these variants are of interest, our discussion will mainly focus on the basic CSP, since this characterizes the greatest body of research in constraint-based reasoning and is most applicable to the job shop scheduling domain.

### 2.2.2 Problem Solving Methods

The CSP is *decidable* [Kfoury 82], since we can always determine if a solution exists by enumerating over all possible values for all variables, and testing all constraints for compatibility (sometimes referred to as *generate and test* [Mackworth 86]).[1] However, this simple enumeration approach might require exponential time in the number of variables to find a solution: given $n$ variables each with finite domain $D_i$, the number of unique assignments to all variables is the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$.

Unless P=NP, no polynomial-time algorithm exists for solving the general CSP [Mackworth 86]. Much research has gone into developing various algorithms that can exploit the constraint-based problem construction. We discuss some of these below.

1. Note that in many of the problem variants described above, decidability is not guaranteed.

#### 2.2.2.1 Backtrack Search

The first improvement we can make to the generate-and-test approach is to incrementally assign a value to each variable, checking all constraints attached to variables with values at each step, and abandoning a particular set of assignments as soon as it is clear that no future assignments will lead to a solution. If performed systematically, this is referred to as *backtrack* search [Bitner 75] , since when an inconsistency is detected, the most recent variable assignment causing the inconsistent value is retracted and a new value is tried. If no value is consistent for a given variable, then the algorithm retracts the previous variable, assigning a new value for it, and so on. Though in the worst case backtracking does not reduce search at all, in practice it has been shown to be a major improvement over systematic generate-and-test [Knuth 75] [Golomb 65] .

Backtracking performed in the manner described above is often referred to as *chronological* backtracking, and is analogous to depth-first-search of a tree [Kumar 92] . Several enhancements have been made to the chronological backtracking approach. These enhancements include *backjumping* [Gaschnig 78] and *backmarking* [Gaschnig 77] . Backjumping takes advantage of the principle that the most recent variable assignment is not always responsible for the current dead-end state. If constraint testing is performed in an appropriate manner, the algorithm can backtrack to an earlier variable that was more likely to have caused the inconsistency. In backmarking, a simple data structure cache is maintained that stores which constraints have been tested positively and negatively. As search proceeds, some constraint checks can be reduced to a simple lookup in the data structure when it is known that the stored result is still valid. *Dependency-directed* backtracking is another enhancement [Stallman 77] found in truth maintenance systems [Doyle 79] . This method stores information that "justifies" a particular variable/value assignment, resulting in similar benefits gained from both backjumping and backmarking.

Backjumping is one technique that tries to avoid a general backtracking phenomenon known as *thrashing*. Thrashing refers to the condition where many backtracks are encountered deep in the search tree, even though the ultimate cause of the of the inconsistencies lies undetected in an early assignment. However, backjumping cannot eliminate all thrashing.

#### 2.2.2.2 Repair-based Search

Backtracking search methods incrementally *construct* a solution one variable at a time. A variable's value can only be changed if backtracking occurs back to that variable's assignment. Backtracking search methods therefore enforce a particular structure on how variables and values are selected. A less structured approach to the way in which variables can be assigned values is the so-called *repair-based* search. In this search scenario, an initial assignment is made to all variables, where some (maybe all) constraints remain violated.[1] New values are selected for variables in the hope that fewer constraints will remain violated after the change. Variations on this include hill-climbing and simulated annealing [Kirkpatrick 83] . The GSAT algorithm is one hill-climbing search procedure [Selman 92] . Though not specifically referred to as a constraint satisfaction algorithm, we can easily view it in that context.

1. We can also consider the initial assignment phase as solving a *relaxed* version of the problem.

#### 2.2.2.3 Non-systematic Backtracking

Backtracking is normally applied systematically; sometimes however a nonsystematic backtracking approach may be preferred. It may be desirable, for example, to restart search after some threshold of backtracks has been reached. It has been shown that a search method that restarts after even one backtrack will exhibit superior performance over systematic search methods on some problems [Langley 92] .

### 2.2.3 Consistency Checking

In the process of selecting a new value for the current variable, the backtrack algorithm checks all constraints between that variable and other previously assigned variables. This *consistency checking* over a subset of variables and constraints in the problem allows the search procedure to prune parts of the search space without removing any solutions.

Consistency checking can be much more extensive than that found in the simplest backtrack mechanisms. Researchers have classified the different levels of consistency checking by the resulting consistency in the constraint network. A graph is considered *k-consistent* if for any subset of *k*-1 variables in the network, a value exists in the domain of a *k*th variable that is consistent with respect to the constraints in the network with all possible values in the other *k*-1 variables [Freuder 78] .

At the simplest level, a network is 1-consistent (also called *node-consistent*) if the domains on all variables in the network are compatible with all unary constraints attached to the variables. 2-consistency (*arc*) and 3-consistency (*path*) are the next highest in complexity. Achieving an *n*-consistent network amounts to finding all solutions to the problem. Since the enforcement of *k*-consistency is exponential in *k* in the worst case [Kumar 92] , consistency of higher degrees becomes less feasible.

Experience has shown that arc and path consistency tend to be the most practical from the point of view of pruning the search space efficiently [Dechter 89] [Nadel 89] . If the entire network is arc-consistent, then the network has achieved *full* arc-consistency. *Partial* arc-consistency is achieved if only part of the entire constraint network is arc-consistent [Nadel 89] .

Determining how much consistency-checking is appropriate has been a central focus of research in improving the efficiency of backtracking search. However, as has been noted by [Prosser 93] , extra consistency checking can degrade search. We will see an example of this in Chapter 6.

### 2.2.4 Variable and Value Ordering

Since in the backtrack search model variables are assigned values incrementally, the order in which variables are assigned and values are tried can also have a dramatic impact on search performance. Search would always be $O(n)$ if a solution exists and we select the correct variable and value at each step. No backtracking would occur.

Much effort has gone into finding efficient variable and value orderings [Brelaz 79] [Bitner 75] [Purdom 83] [Dechter 88] . When systematic search is being performed, the most common approach is to select the *most constrained* variable and assign it the *least constraining* value. The most constrained variable is selected first since it is more likely to cause backtracking than other variables, and backtracking earlier in the search tree causes less thrashing. Selecting the least constraining value is an attempt to find an assignment for the current variable that gives the greatest opportunity for finding consistent assignments to later variables (and hence avoid backtracking).

How one decides what it means for a variable to be most constrained and a value to be least constraining remains an issue of research. Constraint-based schedulers, for example, often exploit domain-dependent constraint information when making these decisions. In [Zweben 92a] the authors demonstrate how variable and value orderings can be learned from a statistical analysis of previous search runs.

## 2.3 Research in Constraint-based Scheduling

At a recent knowledge-based scheduling workshop, most papers in the published proceedings used the constraint model [IJCAI 93]. This evidence suggests that the constraint framework is useful in the scheduling domain. In this section we review some relevant research in constraint-based scheduling.

### 2.3.1 Constraint-Guided Scheduling: ISIS/OPIS/MicroBOSS

ISIS [Fox 87] was the first scheduling system that used constraint information for decision making. ISIS represented many real-world factory conditions using a constraint model. Constraints represented not only physical restrictions but also practical (optimization) preferences. ISIS' problem solving mechanism however was not capable of efficiently managing the bottleneck resources. Its "job-centered" approach (namely scheduling all activities within one job before proceeding to the next one) was vulnerable to creating inefficient schedules around resource bottlenecks when a more resource-centered approach would have been more appropriate.

OPIS [Smith 87] followed ISIS, and represented a more "opportunistic" approach to decision making. OPIS was capable of dynamically switching between resource-centered and job-centered scheduling. The resource-centered approach helped resolve the bottleneck activities efficiently. Still, OPIS was considered "macro-opportunistic" due to the limitations placed on the way opportunism could be exploited.

MicroBOSS [Sadeh 91] has a" micro-opportunistic" approach to scheduling. In this approach, activities are scheduled as a function of constant analysis of the current scheduling state. Therefore MicroBOSS is more capable of handling shifting and multiple bottleneck scenarios than its predecessors. In [Sadeh 91] the author describes how MicroBOSS is a natural evolution of ISIS' and OPIS' search mechanisms. We will discuss more about MicroBOSS in Chapter 6.

### 2.3.2 Iterative Repair: GERRY and SPIKE

GERRY [Zweben 94] and SPIKE [Johnston 89] are two repair-based scheduling systems. In GERRY each constraint has a penalty and weight, the product of which determines the degree to which a particular constraint's violation degrades the overall quality of the schedule. Each constraint has a repair function that attempts to lower that constraint's degree of violation. In each iteration of the repair process, repair functions are called on some number of the violated constraints. GERRY typically uses cheap domain-dependent constraint information to make decisions in its repair functions.

SPIKE uses the Min-Conflicts heuristic [Minton 92] , a lookahead repair method. The designers of SPIKE have reported positive results from the use of this heuristic. We will return to Min-Conflicts in the experimental section of the thesis.

Both SPIKE and GERRY have been used successfully in the scheduling domain. The GERRY scheduling system models activities performed in inspecting, repairing and refurbishing space shuttle orbiters for launch. New schedules are constructed constantly to cope with new problems while still trying to preserve the overall completion date. The SPIKE system performs long-term experiment scheduling for the Hubble Space Telescope. Demand always exceeds the experimental capabilities of the telescope's resources, so the task of generating long-term experiment schedules is an overconstrained problem. This type of problem is well-suited to a repair method.

### 2.3.3 Constraint Posting

Both GERRY and SPIKE typically perform repairs by moving activities from one time interval to another. Constructive approaches such as MicroBOSS incrementally assign times to activities, and backtrack over those assignments only when it is determined that the assignment leads to no feasible solution.

An alternative approach used by [Muscettola 93] is referred to as *constraint posting*. Here instead of assigning an absolute time value to an activity (and perhaps changing that value later in the search process as necessary), the search process involves posting additional temporal constraints between tasks that then force particular orderings between tasks. These postings can be retracted by the search process as desired.

The advantage to such an approach is that each step in the search process may require less unnecessary commitment than methods which assign an actual execution time; if unnecessary commitment is avoided, greater degrees of freedom are left available for later search steps. Both Muscettola and Smith [Smith 93] have reported positive results using this procedure.

## 2.4 Summary

In this chapter we presented relevant background in scheduling, constraint-based reasoning, and constraint-based scheduling systems. We have seen that scheduling in general (and job shop scheduling in particular) has been the focus of research from several perspectives. The constraint-based approach, though relatively new, has demonstrated its value in several applications and maintains a strong following among knowledge-based scheduling researchers.

We also have noted that different scheduling methods have been employed successfully within the constraint-based framework. However, the relative strengths and weaknesses of these approaches remains to be assessed. In the next chapter we describe a model of problem solving that encompasses many strategies used by schedulers. This model characterizes different scheduling methods by a small number of decision-making parameters, and thus provides a basis for associating problem solving performance with parameter settings.

# Chapter 3 A Unified Model for Constraint-based Problem Solving

In this chapter we present our unified model of constraint-based problem solving. Problem solving is viewed as transformational search through a set of problem-solving states. Each transformation involves the assertion or release of some tentative commitment. Transformations are caused by an explicit commitment, consistency enforcement, or an explicit release of one or more commitments. These commitment-based transformations can be structured to perform either constructive or repair-based search.

## 3.1 Search as Commitment Transformation

Search is sometimes viewed as the traversal of problem-solving states via state transition operators [Nilsson 71] . A state transition transforms one problem-solving state into another (see Figure 3). Once a state has been reached that is acceptable, search can terminate. A few systems explicitly represent the search space this way [Laird 87] [Fox 87] [Drummond 93] . Every decision made during search results in a "belief" that the decision made transforms the problem state to one "closer" to the desired solution.

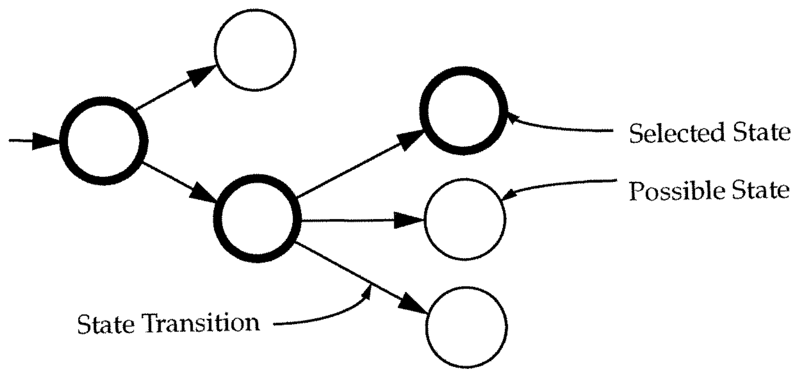

Figure 3. Transformation search through problem solving states.

Let us consider a constraint-based backtracking search procedure from the transformational perspective. The actual transformations being performed typically consist of assigning values to variables as new assertions can be made, and retracting variable assignments during backtracking. As we have discussed previously, consistency enforcement is an important part of the constraint-based search process. When a value is assigned to a variable, the propagation of that assignment may change actual or possible values on other variables. Therefore propagation is a means of making new assertions as well.

As demonstrated by constraint-posting schedulers, the search process may involve the assertion and retraction of temporal constraints instead of variable/value assignments. Both constraint-posting and variable/value assignment create new solution states that (when going forward) restrict actual and/or possible values for variables or (when backtracking) release those restrictions. This suggests that in general search consists of making transformations that create new *commitments*; when backtracking occurs, more transformations *release* those commitments. Once one commitment is made, other commitments may occur as a result of constraint propagation. When an original commitment is released, dependent commitments may also be released.

Repair-based strategies follow commit-release transformations as well. When a variable is assigned a new value, its previous commitment to some value is released before the new value is committed to. If propagation occurs, other new commitments may be created as a result.

Our view of search as the process of asserting and releasing tentative commitments underscores a primary difference between backtracking and repair-based search mechanisms. The backtracking search procedure will only release a commitment when it is determined that the commitment is incompatible with previous commitments. Repair-based techniques can readily release commitments if new commitment opportunities become available.

## 3.2 Commitment Policies

When we consider existing constraint-based scheduling methods that use a commitment-oriented search strategy, we observe that they generally conform to the following pattern:

```
while (not search_termination_condition) do
   select_commitment;
   assert_commitment;
   propagate;
   if (not acceptable_resulting_state) then
      release_commitment(s);
```

We define a *policy* as a specification of the exact manner in which each component of this search loop is to be performed. Policies enforce structure on the commit and release transformations. A new policy can begin once a previous policy has ended. Thus we will assert that a set of commit/release transformations can be viewed as sequentially executing policies which adhere to the above decision-making model. Figure 4 shows how two or more policies can work together to perform transformational search.

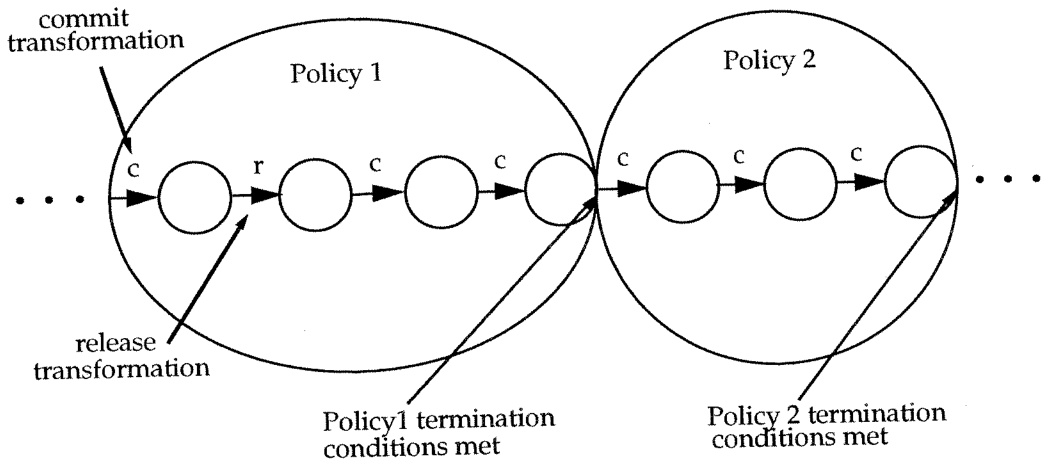
commit transformation
Policy 1
Policy 2
c
r
c
c
c
c
c
release transformation
Policy1 termination conditions met
Policy 2 termination conditions met

Figure 4. Transformations structured by multiple policies.

Each policy then requires the following pieces of information to be completely specified:

- A method for selecting commitments to assert
- A method for propagating after the selected commitment has been asserted
- A function to evaluate the resulting state after propagation
- A release procedure to follow if the resulting state is rejected by the evaluation function
- A condition to determine whether to terminate search using this policy

Both constructive (i.e., backtrack) and repair-based search can be viewed in this way. We note that the policy model makes explicit the repair-based problem solver's requirements for a separate mechanism to generate an initial assignment to all variables.

## 3.2.1 Constructive Commitment Search

In constructive search, extra information is kept so that backtracking can be performed. This information includes a boolean variable (**FORWARD**) which tracks whether search is proceeding forward or backward. Other information is kept to determine when the search space has been exhausted.

When going backward, a different commitment selection method may be preferred than when going forward. We therefore allow for a different selection strategy depending upon the value of **FORWARD**.

Keeping track of the forward/backward and fail condition information, and allowing for different commitment selections according to search direction, results in the following:

```
FORWARD = TRUE
while (not search_termination_condition) do
   if (FORWARD == TRUE) then
      select commitment by forward mechanism;
   else
      select commitment by backward mechanism;
   if (commitment selected) then
      FORWARD = TRUE;
      assert commitment;
      propagate;
      if (not acceptable_resulting_state) then
         release previous commitment;
         FORWARD = FALSE;
   else
      if (search_exhausted) then
         FAIL = TRUE;
      release previous commitment;
      FORWARD = FALSE;
```

This is a nonrecursive version of the basic backtracking search mechanism. For now we restrict ourselves to the simplest release strategy, which only releases the previous commitment when going backwards. Thus our model for backtracking is the standard chronological one.

### 3.2.2 Repair-based Commitment Search

In repair-based search, there generally is no notion of going forward or backward. In addition, there is no notion of exhausting the search space. This leads to a commitment policy structure that is a subset of the one used for constructive search:

```
while (not search_termination_condition) do
   select_commitment;
   assert commitment;
   propagate;
   if (not acceptable_resulting_state) then
      release previous commitment;
```

### 3.2.3 Variable/Value Selection as Commitment

As we have stated, there exists at least two known commitment methods used in constraint-based scheduling: assigning a value to a variable and posting temporal constraints. For now we will focus on the more common case where the main operation for performing commitments involves assigning a value to a variable. Search performed in this framework then becomes the following:

```
while (not search_termination_condition) do
   select var and val;
   assert var and val;
   propagate;
   if (not acceptable_resulting_state) then
      release var and val;
```

The process of selecting a variable and value can be performed many different ways. Often variable and value selection is done separately: first a variable is selected and then a value is selected for that variable. Other methods collect a pool of candidate variable/value pairs and then select from among the members of the pool. Since the former is actually a special case of the latter, we can use the following four steps to do both:

```
select a set of variables
generate a set of candidate values for each of the variables
evaluate all generated variable/value pairs
select one variable/value pair as a result of the evaluation
```

Putting it all together for the paradigm of variable/value selection as commitment, we have the following search template (lines with asterisks do not apply to the repair paradigm):

```
*  FORWARD = TRUE;
   while (not search_termination_condition) do
*     if (FORWARD == TRUE) then
         select vars via forward strategy;
*     else
*        select vars via backward strategy;
      generate candidate vals for vars;
      evaluate all var/val pairs;
      select var/val pair;
*     if (var/val pair selected) then
*        FORWARD = TRUE;
         assert var/val;
         propagate;
         if (not acceptable_resulting_state) then
            release var/val;
*           FORWARD = FALSE;
*     else
*        if (search_exhausted) then
*           FAIL = TRUE;
*        release var/val;
*        FORWARD = FALSE;
```

We now have a general search loop that applies to many constructive and repair-based search methods. This loop operates in the space of commitment transformations when each commitment consists of assigning a value to a variable.

## 3.3 Other Transformation Types

Our perspective of search through transformational commitment can be used to describe many search heuristics. However it does not capture some problem-solving mechanisms that involve *problem reformulation*. If we reformulate a problem during search, we anticipate that this alteration to the problem will help find a solution. We postulate that reformulations can be considered in the transformational model along with our commitment strategies.

Let us consider some possible ways to reformulate constraint-based problems. One way is to reduce the number of variables. Another way is to remove or relax some of the constraints. A third way is to decompose one problem into several smaller problems. Any problem solving performed on reformulated versions of the problem may guide search for the original problem or possibly serve as a solution in its own right.

We propose that problem reformulation techniques can be merged with our general commitment strategy to form an enhanced model of transformational search. We look forward to expanding our commitment search model to incorporate problem reformulation methods. In Figure 5 we show a hypothetical example of transformational search incorporating both commitment and problem reformulation strategies.

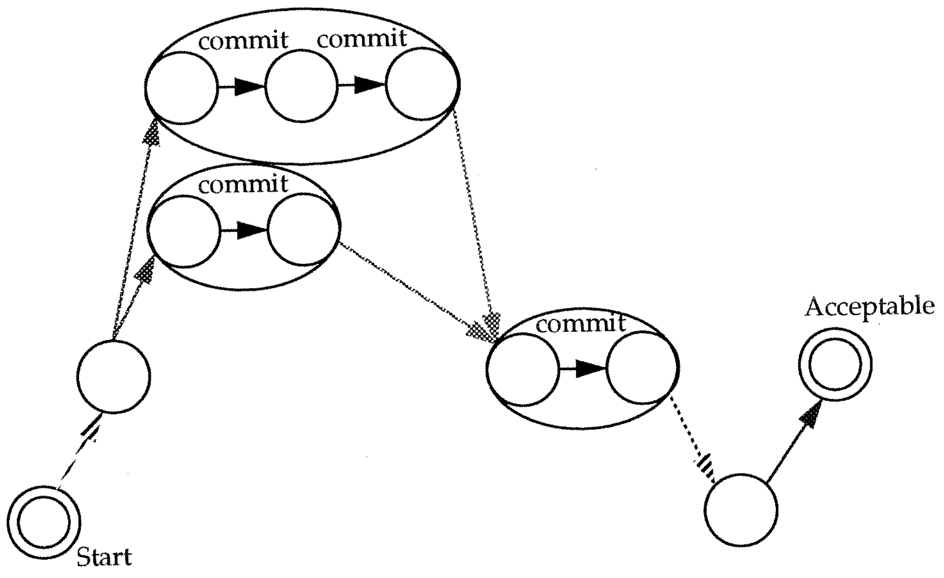

Figure 5. A hypothetical use of commitment with problem reformulation strategies in transformational search. The ovals around the commitment transitions represent individual policies. All other transitions involve some type of reformulation.

# Chapter 4 Representing Scheduling Problems within ODO

In this chapter we describe ODO's representation of a scheduling problem. We also describe the set of input specifications for declaring a problem in ODO.

## 4.1 ODO's Constraint Model

### 4.1.1 Time

Not surprisingly, time is an essential component in a scheduling problem. A *timeline* represents the temporal dimension for scheduling events. Time *points* are the instances at which events occur in a scheduling setting (for example, a certain activity begins execution at time point 12).

For scheduling problems, there often is a notion of a time *horizon* as well. The horizon defines the lower and upper bounds on the timeline within which we restrict the occurrence of events.

In general, we can allow events to occur anywhere on the timeline. We may instead restrict all events to only discrete time points on the timeline. Thus time can be represented either continuously or discretely. If represented discretely, then typically a time *granularity* is chosen that is appropriate to the domain. The discrete time representation is applied in most job-shop-type settings. When representing scheduling problems as constraint satisfaction problems, some variables will represent the time points at which an event is expected to occur. We refer to these as *temporal variables*. When time is represented discretely and a time horizon is defined, all temporal variables will have finite domains of possible values.

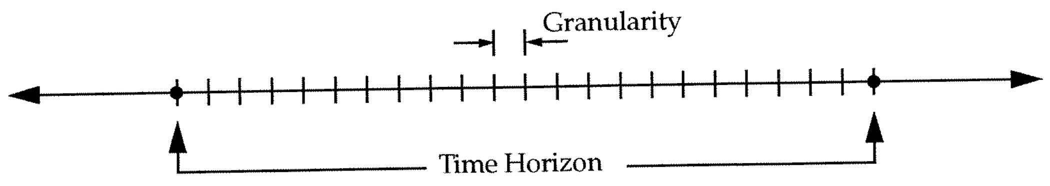

Figure 6. Timeline.

In our current implementation, the granularity is set to one time unit, and the time horizon is defined by the minimum and maximum declared times for activities or jobs (see below).

### 4.1.2 Jobs/Activities/Tasks

In our description of the job-shop problem, jobs are composed of activities, and activities are the entities which must be assigned valid execution times in a satisfying schedule. It is often more convenient to refer to jobs and activities simply as *tasks*. When a hierarchical relation exists among tasks (as is the case between jobs and activities), we simply refer to the tasks as *parent* tasks or *subtasks* as appropriate.

All tasks have *durations* associated with them; a duration specifies the relation between the task's start and end event time points. If a task has subtasks, its duration is only defined by that of its subtasks. This is true in the job shop domain of the relationship between jobs and activities: only when we know the execution times of all activities of a given job will we know when that job itself begins and ends.

In ODO, we represent the task's start-time and end-time with temporal variables. We also make the duration a variable. Expressing the relation between a task's start-time, end-time, and duration is done in the form of a temporal constraint between the three variables:

$st(T) + dur(T) = et(T)$

Figure 7 illustrates the basic task object composed of three variables and a duration constraint.

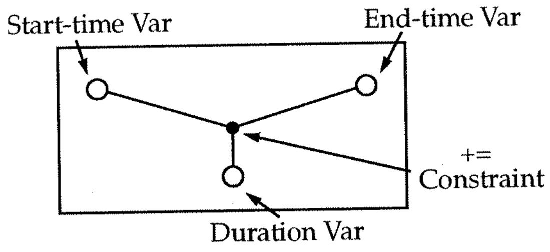

Figure 7. Basic Task.

### 4.1.3 Resources

Resources include concepts such as machines, raw materials, and personnel. In Chapter 2 we discussed how individual resources can be generalized into *resource pools* of a particular *resource class*.

A resource pool object represents a particular resource that is allocated to activities in a schedule. The resource pool points to its resource class and to its timeline representation [Zweben 92] [Williams 86] which is called a *history*. A history tracks the status of a resource pool's available capacity over time. This is represented using a list of connected intervals, where each interval specifies a value. For example, given:

$[-\infty\ .\ 100 — 1][100\ .\ 200 — 0][200\ .\ +\infty — 1]$

This history specifies that over the time interval $-\infty$ to 100, the resource pool has one unit of available capacity; from 100 to 200, zero units of capacity; from 200 to $+\infty$, one unit of capacity.

For search efficiency the history intervals are implemented as a red-black tree [Cormen 91]. This structure provides *O(log(n))* performance in finding, inserting, and deleting elements of the history. In addition, since updating often occurs to successive tree elements, the tree is *threaded* so that access to a particular history element's successor and predecessor is *O(1)*. For small problems, such a complex structure is not necessary. However, our use of the red-black tree implementation will ensure that history access and update performance is affected minimally as the scheduling problem grows.

### 4.1.4 Temporal Constraints

Temporal constraints are those that connect temporal variables. The duration constraint is one type of temporal constraint. Others include the constraints between variables of two different tasks or other temporal variables relevant to the scheduling domain.

Tasks may have absolute time restrictions on the values for the start and end-time variables. These usually come in the form of *earliest start* and *latest finish* (ES/LF) times for the task. In the constraint framework, we can create temporal variables representing the ES and LF times. Then we can form >= constraints between the ES/LF variables and the task's start-time and end-time variables.

When it is specified that one task can begin only after another task has ended, we attach a constraint between the start-time of one task and the end-time of the other. For example, if T2 cannot start until T1 has finished, we have the following constraint:

*st(T2) >= et(T1)*

Allen [Allen 84] has enumerated 13 possible temporal relations between tasks. Each of these relations can be readily duplicated in the variable/constraint framework.[1] As an example, we have the Allen relation "*x* DURING *y*", which implies that task *x*'s execution occurs completely within the time span of task *y*'s execution. For this relation we need 2 constraints between the start and end time variables of *x* and *y*:

$st(x) > st(y)$

$et(x) < et(y)$

If these constraints are both satisfied, then Allen's DURING relation holds.

### 4.1.5 Resource Constraints

Tasks often require the use of some quantity of some resource pool (see Section 4.1.3). The requirement is specified at the resource class level. If there are multiple pools, any pool can address the request for the resource. For example, if task T2 needs a mill operator, then it can look to either Employee Pool1 or Employee Pool2 for an actual resource pool.

In ODO's constraint framework a task is given a *resource request variable* for each of its resource requirements. The resource request variable caches the necessary resource quantity; this variable will be assigned a resource pool as its value when a pool is allocated to the task. A *resource constraint* ties the resource request variable to the start and end time variables of the task. The constraint is satisfied only

1. It should be noted that our temporal constraints alone cannot duplicate logical quantification, conjunction, and disjunction over Allen relations, as the temporal logic described in [Allen 84] performs.

if the pool assigned to the resource request variable has the requested amount available from the start time to the end time of the task. This is determined through the use of the history information of the assigned resource pool. Once a resource pool has been allocated to a task over a specified period of time, that resource pool's history is updated to reflect the allocation.

At the completion of a task, the requested resource may be returned as available (the resource was only *used*), or it may have been *consumed* by the task itself. Personnel are used, whereas raw materials are typically consumed. Tasks may also *produce* a particular resource as a result of execution. For example, a task may produce a subassembly that will be consumed by other tasks in the assembly of a finished product. In these cases the resource pool's histories must be updated to represent all consumptions and productions appropriately.

## 4.2 Problem Declaration in ODO

Problems are constructed within ODO by establishing the above objects, variables, and constraints as appropriate. It is often tedious to describe a problem using variables and constraints alone. ODO's input language accepts a more natural description of the problem, from which ODO can construct the constraint graph. Many known scheduling input languages describe problems using tables of numbers, or cannot easily extend an existing definition of a previously declared object. ODO's language for describing problems was designed to be incremental, extensible, and intuitive.

ODO represents a useful superset of the strict job shop scheduling problem class. The following subsections enumerate the problem declarations accepted by ODO as input. Keywords and literal text are written in **`courier bold`** font, identifiers and integers are written in normal font, nonterminals are written in *italic*, and optional arguments are placed within square brackets ([ ] - one optional argument; [ ]* - zero or more optional arguments; [ ]+ - one or more optional arguments). The complete grammar is summarized in the appendix.

### 4.2.1 Declaring Resources

Resources are declared by specifying resource classes and the resource pools that belong to those classes.

- **`resource_class`** resource_class_string [parent_resource_class_string] **`;`**

  Description:

  A resource class is declared with name resource_class_string. If parent_resource_class_string is specified, resource_class_string is defined as a child of parent_resource_class_string.

  Restrictions:

  The parent resource class must have been previously declared.

  Example:

  **`resource_class WorkCenter1A WorkCenter1;`**

- **`resource_pool`** resource_pool_string initial_amount_integer [resource_class_string]+ **`;`**

  Description:

A resource pool is declared with name resource_pool_string. It belongs to all classes specified in [resource_class_string]+. The pool's history is initialized with initial_amount_integer from time negative-infinity to time positive-infinity.

Restrictions:

The resource classes must have been previously declared.

Example:

```
resource_pool Machine1 1 WorkCenter1;
```

### 4.2.2 Declaring Tasks

- **`task`** task_string [parent_string];

  Description:

  A task is declared with name task_string. If parent_string is specified, then task_string is defined as a child of parent_string.

  Restrictions:

  The parent task must have been previously defined.

  Example:

  ```
  task Activity1 Job1;
  ```

### 4.2.3 Declaring Temporal Constraints

- **`duration`** task_string duration_integer;

  Description:

  The duration for task_string is set to duration_integer. No units are associated with time.

  Restrictions:

  The task must have been previously declared. Only non-negative durations are allowed. Durations for parent tasks are ignored during problem solving.

  Example:

  ```
  duration Activity1 10;
  ```

- **`before`** task_string task_string;

  Description:

  This establishes a temporal constraint between the first task and the second task. The constraint is satisfied only if the end-time of the first task is less than or equal to the start-time of the second task.

  Restrictions:

  Both tasks must have been previously declared.

  Example:

  ```
  before Activity1 Activity2;
  ```

- **`earliest_start`** task_string earliest_start_integer;

  Description:

  This establishes a constraint on the earliest value for the start-time of the task.

  Restrictions:

The task must have been previously declared.

Example:

```
earliest_start Activity1 20;
```

- **`latest_end`** task_string latest_end_integer;

  Description:

  This establishes a constraint on the latest value for the end-time of the task.

  Restrictions:

  The task must have been previously declared.

  Example:

  ```
  latest_end Activity1 50;
  ```

### 4.2.4 Declaring Resource Constraints

- **`use_resource`** task_string resource_class_string amount_integer;

  Description:

  This specifies that the declared task requires (from its start-time to its end-time) an amount_integer quantity of some resource pool that belongs to resource_class_string.

  Restrictions:

  The task and resource class must have been previously declared.

  Example:

  ```
  use_resource JobA1 WorkCenter1 1;
  ```

- **`consume_resource`** task_string resource_class_string amount_integer *change_start;*

  Description:

  This specifies that the declared task requires and consumes an amount_integer quantity of some resource pool that belongs to resource_class_string. The consumption is performed either at the start-time (`st`) or end-time (`et`) of the task (indicated by *change_start*).

  Restrictions:

  The task and resource class must have been previously declared.

  Example:

  ```
  consume_resource JobA1 RawMaterial1Pool 10 st;
  ```

- **`produce_resource`** task_string resource_class_string amount_integer *change_start;*

  Description:

  This specifies that the declared task produces an amount_integer amount of the specified resource pool. The production is performed either at the start-time (`st`) or end-time (`et`) of the task (indicated by *change_start*).

  Restrictions:

  The task and resource_pool must have been previously declared.

  Example:

  ```
  produce_resource JobA1 RawMaterial2Pool 10 et;
  ```

## 4.3 Problem Representation Example

In this section we will demonstrate how to declare a simple job shop problem within ODO. This problem is identical to an example problem found in [Sadeh 91] . The problem consists of four resources and four jobs. Each job consists of either two or three activities. All activity durations are three time units. The earliest-start and latest-end for all jobs is 0 and 15, respectively. Figure 8 presents the job/activity structure of the problem; Figure 9 depicts the resulting constraint graph created within ODO.

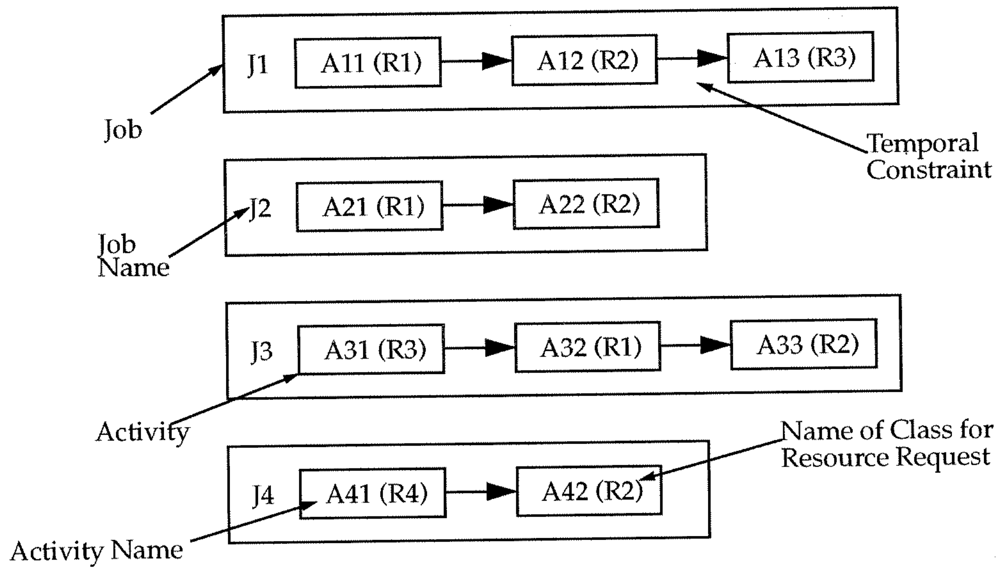

Figure 8. Example problem. All activities have duration = 3. All jobs have an earliest start time of 0 and a latest end time of 15. Temporal constraints define what task must finish (the tail of the arrow) before what task can start (the head of the arrow).

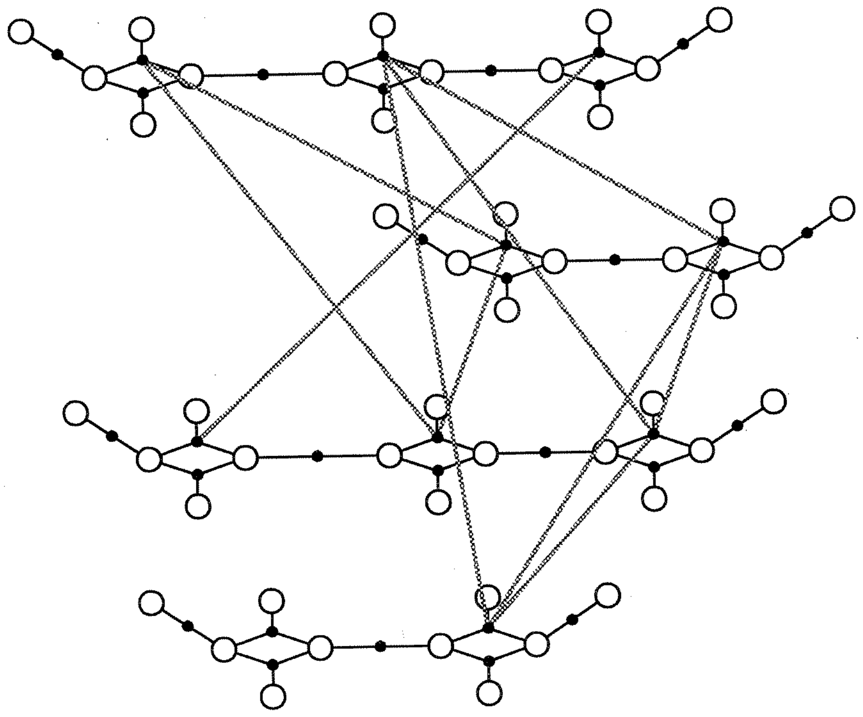

Figure 9. Constraint graph of example problem. Empty circles are variables; filled circles with lines are constraints (solid: temporal; gray: resource).

If the following statements are read as input (in this order or something similar but respecting the restrictions outlined earlier), then this problem will be instantiated within ODO's structures (ODO allows lines that begin with '#' to indicate comments):

```
# declare resource classes and pools (one pool per class,
# capacity of 1 in each pool)
resource_class RC1;
resource_pool RP1 1 RC1;
resource_class RC2;
resource_pool RP2 1 RC2;
resource_class RC3;
resource_pool RP3 1 RC3;
resource_class RC4;
resource_pool RP4 1 RC4;

# 4 jobs
task J1;
task J2;
task J3;
task J4;

# individual activities for each job
task A11 J1;
task A12 J1;
task A13 J1;
task A21 J2;
task A22 J2;
task A31 J3;
task A32 J3;
task A33 J3;
task A41 J4;
task A42 J4;

# all activity durations are 3
duration A11 3;
duration A12 3;
duration A13 3;
duration A21 3;
duration A22 3;
duration A31 3;
duration A32 3;
duration A33 3;
duration A41 3;
duration A42 3;

# temporal relations between activities in the jobs
before A11 A12;
before A12 A13;
before A21 A22;
before A31 A32;
before A32 A33;
before A41 A42;

# resource requests for each activity
use_resource A11 RC1 1;
use_resource A12 RC2 1;
use_resource A13 RC3 1;
use_resource A21 RC1 1;
use_resource A22 RC2 1;
use_resource A31 RC3 1;
use_resource A32 RC1 1;
use_resource A33 RC2 1;
use_resource A41 RC4 1;
use_resource A42 RC2 1;

# time bounds on each job
earliest_start J1 0;
latest_end J1 15;
earliest_start J2 0;
latest_end J2 15;
earliest_start J3 0;
latest_end J3 15;
earliest_start J4 0;
latest_end J4 15;
```

# Chapter 5 Solving Scheduling Problems in ODO

Problem solving within ODO involves the declaration of all policy parameters followed by the initiation of ODO's search mechanism. In this chapter we describe how to declare the search parameters and begin problem solving.

## 5.1 Performing Search

ODO uses the problem solving procedure outlined in Chapter 3 for search using variable and value selection. We reprint this procedure for convenience (recall that lines with asterisks are only performed when constructing as opposed to repairing):

```
*  FORWARD = TRUE;
   while (not search_termination_condition) do
*     if (FORWARD == TRUE) then
         select vars via forward strategy;
*     else
*        select vars via backward strategy;
      generate candidate vals for vars;
      evaluate all var/val pairs;
      select var/val pair;
*     if (var/val pair selected) then
*        FORWARD = TRUE;
         assert var/val;
         propagate;
         if (not acceptable_resulting_state) then
            release var/val;
*           FORWARD = FALSE;
*     else
*        if (search_exhausted) then
*           FAIL = TRUE;
*        release var/val;
*        FORWARD = FALSE;
```

ODO performs this search according to the declared policy parameters. Once a policy has been declared, search is begun by calling `construct` or `repair`.

ODO maintains extra problem-solving information during search. For variables, a set of *possible values* is kept. This set maintains those values that may be asserted for the variable in this search iteration. Each search state also maintains a list of those variables that have been assigned values and the remaining variables that have not. During constructive search, these lists change as new variables are assigned values or as backtracking occurs. Finally, some number of previous states are kept for the purposes of backtracking or reverting. In constructive search, all previous states that may be backtracked to are stored; in repair-based search, only the previous state is kept.

In the current version of ODO, the decision variables are the start-times of all activities. Once a start-time has been determined for a task, the end-time is derived by propagating through the duration constraint. We assume that there exists only one resource pool for each resource class, so that decisions do not involve the selection of a resource pool.

## 5.2 Declaring Policies

In order to begin search, a policy must be completely specified. Namely:

- How to select a set of candidate variables
- How to generate a set of candidate values for those variables (i.e., make candidate variable/value "pairs")
- How to evaluate the candidate variable/value pairs
- How to select from among the evaluated variable/value pairs
- How to perform propagation
- How to decide whether or not to accept the resulting variable/value assertion
- When to terminate search using the above settings

ODO provides a library of texture measures for controlling how each policy parameter is performed. In the following subsections we describe ODO's policy options. Keywords and literal text are written in **`courier bold`** font, identifiers and integers are written in normal font, nonterminals are written in *italic*, and optional arguments are placed within square brackets ([ ] - one optional argument; [ ]* - zero or more optional arguments; [ ]+ - one or more optional arguments). The complete grammar is summarized in the appendix.

### 5.2.1 Variable and Value Selection

We saw in our description of our unified model (Chapter 3) the following pseudocode to select a variable/value pair for assertion:

```
select a set of variables
generate a set of candidate values for each of the variables
evaluate all generated variable/value pairs
select one variable/value pair as a result of the evaluation
```

ODO executes four procedures sequentially to perform this process — **`SelectVariables`**, **`GenerateVarValPairs`**, **`ScoreVarValPairs`**, and **`SelectVarValPair`**. We describe these procedures in the next four subsections and show how the policy declarations are made for each.

### 5.2.1.1 The SelectVariables Procedure

The `SelectVariables` procedure begins the variable/value selection process. Its input is a list of filter functions. Each filter function takes a list of variables as input and generates as output a subset of that variable list. `SelectVariables` calls each filter function in turn, passing as input to a filter function the output from the previous filter function. The first filter function is passed a list of all activity start-time variables. The output of the last filter function is passed on to the `GenerateVarValPairs` procedure.

Each filter function performs a texture measure of the graph. The filter functions are specified using ODO's declarative language in the following manner:

> **`var_selection`** [*var_selection_filter*]+**;**

Since we may choose to use a different variable selection procedure when going backward, we also have:

> **`backward_var_selection`** [*var_selection_filter*]+**;**

Each *var_selection_filter* uniquely specifies a filter function. Filters are applied in the order they are declared. The following is the current catalog of the variable-selection filter functions in ODO:

> **`violated`** - filters the input list to the start-time variables that belong to tasks that have a violated resource constraint.
>
> **`most_recent_failure`** - filters list to the variable that most recently failed in backtrack search.
>
> **`smallest_pv_cardinality`** - filters list to those variables that have the smallest set of possible values.
>
> **`orr`** - filters list to the variables with the highest *operations resource reliance* value [Sadeh 91] (described in Chapter 6).
>
> **`all_predecessors_assigned`** - filters list to those variables whose tasks have no unassigned temporal predecessors. This is useful for simple constructive variable selection in a forward-dispatching manner.
>
> **`all_successors_assigned`** - filters list to those variables whose tasks have no unassigned temporal successors. This is useful for simple constructive variable selection in a backward-dispatching manner.
>
> **`unassigned`** - filters list to those variables that are currently unassigned.
>
> **`random`** - nondeterministically filters the input list down to one element. Uses the system's random function to select the element to keep in the list.
>
> **`arbitrary`** - deterministically filters the input list down to one element (i.e. the first element of the list is always kept).
>
> **`all`** - redundant (since it performs no filtering), but useful to put down as the only element in case all variables are to be considered.
>
> **`none`** - returns an empty list. This is useful for when a policy should force backtracking.

The following is an example declaration using multiple filters. This declaration will find all unassigned start-time variables, filter those down to ones whose predecessors are assigned, and then select one randomly:

```
var_selection unassigned all_predecessors_assigned random;
```

### 5.2.1.2 The GenerateVarValPairs Procedure

`GenerateVarValPairs` accepts as input the filtered variable list (output from `SelectVariables`) and a value generation function. It will output a new list of variable/value pairs. The generation function will create some number of unique variable/value-pair entries for each variable in the input list. Each resulting variable/value pair represents a candidate assertion for the current search process. ODO passes the list of variable/value pairs to `ScoreVarValPairs` for evaluation.

ODO's value generation function is declared using the following:

> `val_generation` *val_generation_function* `;`

The following is the current catalog of texture measures used as generation functions in ODO:

> `all_pv` - generates variable/value pairs for all values in the variable's set of possible values.
>
> `all_but_current_pv` - generates variable/value pairs for all values in the variable's set of possible values except for the currently assigned value (assuming the variable has a value).
>
> `arbitrary_pv` - deterministically generates a single variable/value pair from the variable's set of possible values.
>
> `random_pv` - nondeterministically generates a single variable/value pair from the variable's set of possible values.

### 5.2.1.3 The ScoreVarValPairsProcedure

`ScoreVarValPairs` accepts as input a list of variable/value pairs (output from `GenerateVarValPairs`) and a scoring function. The scoring function is called on each variable/value pair. The output of `ScoreVarValPairs` is a list of variable/value/score triples; this list is passed to the `SelectVarValPair` Procedure.

The scoring function is declared using the following:

> `scoring_function` *scoring_function* `;`

The following is the current catalog of texture measures used as scoring functions in ODO:

> `cost_lookahead` - The specified variable/value pair is asserted, propagation is performed according to the current propagation method (see below), and the resulting state is scored using the cost function (see below). After scoring, the original state is restored.
>
> `fss` - The specified variable/value pair is scored via the *filtered survivable schedules* mechanism [Sadeh 91] (described in Chapter 6).
>
> `none` - No scoring is performed (for when there is no need to perform an evaluation of the variable/value pairs).

#### 5.2.1.4 The SelectVarValPair Procedure

`SelectVarValPair` accepts as input a list of variable/value/score triples (from ScoreVarValPairs) and a list of selection filter functions. `SelectVarValPair` calls each filter function in turn, passing as input to a filter function the output from the previous filter function. The first filter function is passed the original variable/value/score triple list. The output of the function is the variable/value pair from the first element of the surviving variable/value/score triple list.

The scoring function is declared using the following:

> `selection_function` [*selection_function*]+ ;

The following is the current catalog of texture measures used as selection functions in ODO:

> `min_score` - selects the variable/value/score triples with the lowest score
>
> `max_score` - selects variable/value/score triples with the highest score
>
> `earliest_value` - selects variable/value/score triples with the earliest value
>
> `latest_value` - selects variable/value/score triples with the latest value
>
> `earliest_latest_value` - selects variable/value/score triples with either the earliest or latest value
>
> `random` - selects one among all variable/value/score triples randomly
>
> `arbitrary` - selects one among all variable/value/score triples arbitrarily

The following example filters all variable/value/score triples down to those that have the lowest score, then filters that list down to one randomly:

```
selection_function min_score random;
```

### 5.2.2 Constraint Propagation

Once a variable and value are selected, the value is asserted for that variable. Since variable and value selection occur on start-time variables only, the assertion is propagated through the duration constraint to the task's end time. In addition, since we assume only one resource pool per resource class, we can assign the appropriate resource pool's to this task's resource request variables.

At this point, any desired additional constraint propagation will occur. ODO performs this propagation by calling the `Propagate` procedure. `Propagate` accepts as input the variable/value pair that has just been asserted and a list of propagation functions to perform. ODO performs the propagation functions in sequence, passing to each function the variable/value pair.

The propagation functions are declared using the following:

> `propagation_method` [*propagation_function*]+ ;

A propagation function may enforce consistency on either actual or possible values for a variable. The following is the current catalog of propagation functions in ODO:

> `none` - no propagation is performed.

**temporal_pv_unassigned** - propagates the possible values of all variables connected via temporal constraints to the input variable/value pair. If full temporal consistency held before the new variable/value pair assertion, temporal consistency is guaranteed after this propagation has been performed. For job shop problems, this is performed in *O(#activities per job).*

**full_temporal_pv_unassigned** - propagates the possible values of variables via temporal constraints through the complete temporal constraint network. Regardless of temporal consistency before the variable/value assertion, full temporal consistency is guaranteed after this propagation. For job shop problems, this is performed in *O(#activities).*

**temporal_v_assigned** - propagates the actual values of variables connected via temporal constraints. If temporal consistency held before the new variable/value pair assertion, full temporal consistency is guaranteed after this propagation has been performed. For job shop problems, this is performed in *O(#activities per job).*

**unit_resource_pv_unassigned** - rejects possible values for unassigned variables if there does not exist available capacity in an eligible resource pool for the duration of the variable's task starting at that possible value. This achieves the equivalent of "forward checking" [Haralick 80] with respect to resource constraints. For job shop problems, this is performed in *O(#activities * p)*, where *p* is the maximum number of possible values for the start-time of the activity.

**binary_resource_pv_unassigned** - eliminates possible values for the variables of two unassigned tasks that will require the same unit-capacity resource pool and which must overlap [Sadeh 91]. For job shop problems, this is performed in *O(#activities * p)*, where *p* is the maximum number of possible values for the start-time of the activity.

We note that temporal and resource propagation are performed independently. This is due to common practice in the scheduling domain, since full temporal consistency can be achieved more efficiently than resource consistency. The **unit_resource_pv_unassigned** and **binary_resource_pv_unassigned** resource propagation functions achieve partial resource arc-consistency.

The following is an example of a propagation declaration:

```
propagation_method temporal_pv_unassigned
   unit_resource_pv_unassigned;
```

## 5.2.3 Accept Criteria

After assertion and propagation, the resulting state is checked to see if it is to be kept or if the previous state should be restored. This acceptance function is declared as follows:

**accept_criteria** *accept_criteria_function* ;

The following is the current catalog of texture measures used as acceptance functions:

**always** - always accept the resulting state.

**cost_leq** - accept the resulting state when its cost (see below) was kept the same or reduced from the previous state.

**no_empty_pv** - accept when the resulting state did not result in emptying any variable's set of possible values.

### 5.2.4 Backtrack Mechanism

At this point, only chronological backtracking is supported. Therefore no backtrack mechanism needs to be declared.

### 5.2.5 Search Termination Criteria

At the beginning of each search iteration, ODO checks to see if the search termination condition has been met. The search termination condition is an expression that allows for all logical relations and predicates, arithmetical negation, and a small number of *problem-state measurement* functions. The semantics of logical operators follows that of C and C++: for example, a logical negation of a non-zero number returns zero. The symbols, precedence, and associativity are identical to that found in C and C++ [Stroustrup 91] . Precedence can be overridden with the use of parentheses. We describe the expressions in greater detail in Appendix A.

The search termination expression is declared using the following:

> **`search_termination_criteria`** *expression*;

The following is the current catalog of problem-state measurement functions in ODO:

> **`cost`** - returns the value of the declared cost function (see below).
>
> **`search_time`** - returns the time (in seconds) passed since the beginning of search.
>
> **`search_iterations`** - returns the number of iterations since the beginning of search.
>
> **`num_backwards`** - returns the number of iterations for which the **`FORWARD`** search flag was = **`FALSE`**.
>
> **`exhaust_search`** - returns 1 (i.e. **`TRUE`**) if the search space has been exhausted.

The following is an example termination declaration:

```
search_termination_criteria (cost == 0) || exhaust_search;
```

### 5.2.6 Cost Function

In various components of the search process it has been convenient to base decisions upon the returned value of a single 0-ary *cost* function. This function evaluates the current state and returns an integer value. The cost function is declared using the following:

> **`cost_function`** *cost_function* ;

There are currently two texture-based cost functions in ODO:

> **`num_violated_constraints`** - returns the number of violated constraints.
>
> **`num_unassigned`** - returns the number of unassigned (start-time) variables.

## 5.3 Problem Solving Examples

In this section we will demonstrate how to use ODO to solve a particular scheduling problem. We use the same example problem as that shown in Figure 8 (Chapter 4). We will assume that the ODO code presented there has been previously input into ODO. All that remains to begin search is to declare policy parameters and run ODO's problem-solver.

### 5.3.1 Constructive Search

One way to solve the problem is through a simple constructive scheme. For this policy, we perform the following at each step:

- one unassigned variable is selected
- the earliest possible value for that variable is asserted
- temporal and unit resource propagation occurs on the possible values of the remaining unassigned variables
- backtracking occurs if some unassigned variable has an empty set of possible values
- search terminates if the search space is exhausted or if the number of unassigned variables = 0

```
var_selection
      unassigned
      arbitrary;
backward_var_selection
      most_recent_failure;
val_generation
      all_pv;
scoring_function
      none;
selection_function
      earliest;
accept_criteria
      no_empty_pv;
propagation_method
      temporal_pv_unassigned
      unit_resource_pv_unassigned;
cost_function
      num_vars_unassigned;
search_termination_criteria
      (cost == 0) || exhaust_search;
```

Figure 10 shows ODO's resulting schedule (the 'A' is omitted from the activity labels on the Gantt chart).

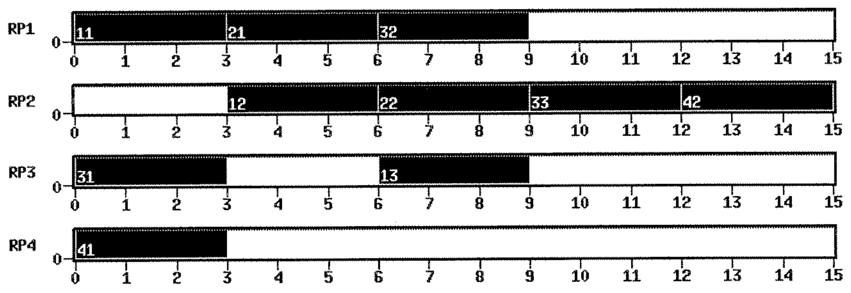

Figure 10. Gantt chart of resulting schedule after constructive search.

## 5.3.2 Repair-based search

Another way to solve the problem is to search using a repair method. The repair method must be initialized by executing a policy that assigns values to all variables. We can do that in the following constructive manner, where tasks are placed consistently with respect to temporal constraints but without checking any resource constraints:

```
var_selection
      unassigned
      random;
backward_var_selection
      none;
val_generation
      all_pv;
scoring_function
      none;
selection_function
      earliest_value;
propagation_method
      temporal_pv_unassigned;
accept_criteria
      no_empty_pv;
cost_function
      num_vars_unassigned;
search_termination_criteria
      (cost == 0);
```

Figure 11 shows the schedule resulting from the initialization policy. Note here that we can use a Gantt chart to display a schedule containing violated constraints. If at any time two tasks are placed on the chart above the resource pool's capacity line (dotted), then there exist resource violations.

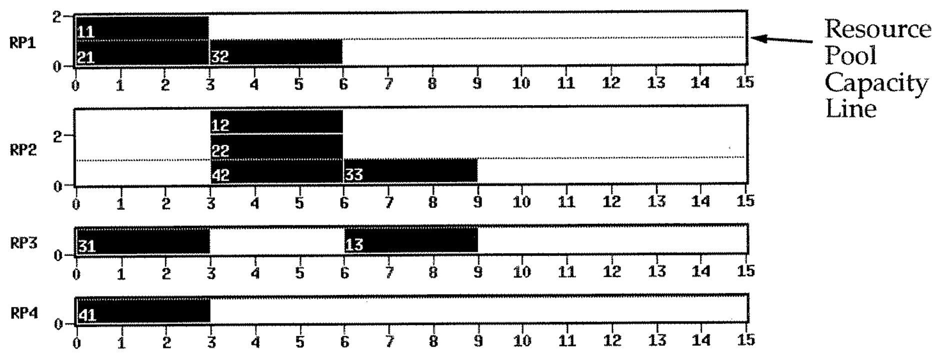

Figure 11. Temporally consistent schedule after initial assignments (in preparation for repair-based search).

Now we can call a repair policy to fix the schedule. At each step:

- a random variable is selected
- a random value is asserted for that variable
- propagation enforces temporal constraints; therefore other variables that are linked via temporal constraints to the newly asserted variable may have their values changed
- the resulting state accepted only if the resulting number of constraint violations was less than or the same as that found in the previous state

```
var_selection
      random;
val_generation
      random_pv;
scoring_function
      none;
selection_function
      random;
propagation_method
      temporal_v_assigned;
accept_criteria
      cost_leq;
cost_function
      num_violated_constraints;
search_termination_criteria
      (cost == 0);
```

This policy is essentially performing random repairs at each step, accepting only those that lower the number of violated constraints. Figure 12 shows the resulting schedule.

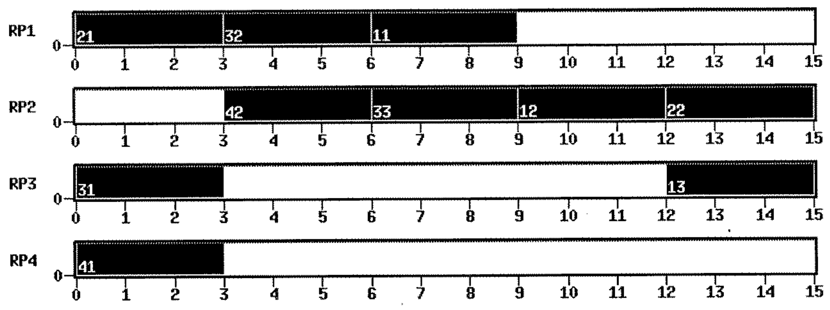

Figure 12. Resulting schedule after repair policy performed.

# Chapter 6 Experiments Using ODO

ODO's capabilities as an interpreter of scheduling policies allow us to test many heuristics from within our problem solving model. This gives us the opportunity to isolate the differences between problem solving methods and correlate search performance with problem properties. In this chapter we describe some preliminary experimental results. We first demonstrate how ODO emulates two successful scheduling mechanisms — one constructive, the other repair-based. We also show how slight variations to these algorithms alters performance on a known set of scheduling benchmark problems. Lastly, we show that the Min-Conflicts heuristic may not solve a problem in thousands of iterations even if search begins one step away from a solution.

## 6.1 Reconstructing MicroBOSS within ODO

Our reconstruction of MicroBOSS is based upon the version described in [Sadeh 91] . Micro-BOSS' decision-making heuristics rely upon the constructive approach to generating a satisfactory schedule. The variable selection method is a heuristic aimed at finding the most constrained variable and assigning it the least constraining value. Therefore Micro-BOSS attempts to make its most critical decisions as early as possible; should backtracking occur, it occurs relatively early and hence avoids much thrashing.

### 6.1.1 Variable and Value Selection in MicroBOSS

Micro-BOSS performs separate phases of variable and value selection. Variables are selected by analyzing the remaining unassigned activities (variables) and finding the activity that most heavily depends upon the most contended-for resource/time interval. Once this activity has been identified, an actual temporal assignment (value) is found for that activity that will most likely survive through later assignments to other activities.

Micro-BOSS performs a complex process to make these decisions. In variable selection, Micro-BOSS generates (for each unscheduled activity and each resource that activity may use) a demand profile, which measures the probability that the activity will require a particular resource at a particular time. Aggregate demand profiles are generated for each resource by summing the individual demand profiles associated with a given resource. The resource experiencing the highest aggregate demand is considered to be the most contended-for resource. The activity contributing the greatest individual demand to the aggregate demand peak of this resource is the activity considered most reliant on that most contended-for resource. Micro-BOSS selects that activity for scheduling. Sadeh refers to this method of variable selection as the ORR heuristic (for Operations Resource Reliance).

Value selection is then performed for the selected activity. This is done by determining which of the variable's possible values appears most promising according to *survivability* and *compatibility* measures. The survivability measure reflects the likelihood that other tasks will not require that resource-time reservation, and the compatibility measure estimates how many schedules would be compatible with this assignment. This value-selection method is called the Filtered Survivable Schedules heuristic, or FSS.

Micro-BOSS systematically assigns time-resource reservations to activities using the above methods for variable and value selection. Propagation is performed after each new assignment that results in full arc-consistency with respect to temporal constraints but only partial arc-consistency with respect to resource constraints. Backtracking proceeds in the standard chronological manner.

## 6.1.2 ODO's Implementation of Micro-BOSS

In order to emulate Micro-BOSS within ODO, we needed to reconstruct the calculation metrics described above. We therefore added a new variable selection filter, `orr`, and a new scoring function, `fss`. MicroBOSS' problem-solving procedure is then declared within ODO as follows:

```
cost_function
      num_unassigned_vars;
var_selection
      unassigned
      orr
      arbitrary;
backward_var_selection
      most_recent_failure;
val_generation
      all_pv;
propagation_method
      full_temporal_pv_unassigned
      unit_resource_unassigned
      binary_resource_unassigned;
scoring_function
      fss;
selection_function
      max_score
      arbitrary;
accept_criteria
      no_empty_pv;
search_termination_criteria
      (cost == 0) || (search_iterations == 1000);
```

Note that variable selection and the evaluation method ultimately break ties arbitrarily. MicroBOSS is described as a deterministic algorithm; therefore tie-breaking should not depend upon the state of the random number seed.

The heuristic described above (including the termination of search after 1000 steps) was used in experiments presented in [Sadeh 91] on a suite of 60 test problems. These problems consisted of six different problem "classes", based upon the number of a-priori bottlenecks (one or two) and how loose the due-date was set ("wide"," narrow", and "0" looseness). Ten problems of each class were created for the test suite. Each problem comprised ten jobs of five activities.

Figure 13 shows a solution to one of the 1-bottleneck/0-looseness problems. The numbers labeling each box in the Gantt chart uniquely specify each activity, and are of the form *XY*, where *X* is the job number (0-9) and *Y* is the activity number within job *X* (0-4). For any two activities within a job, $XY_1$ and $XY_2$, if $Y_1 < Y_2$, then activity $XY_1$ must finish before $XY_2$ can begin.

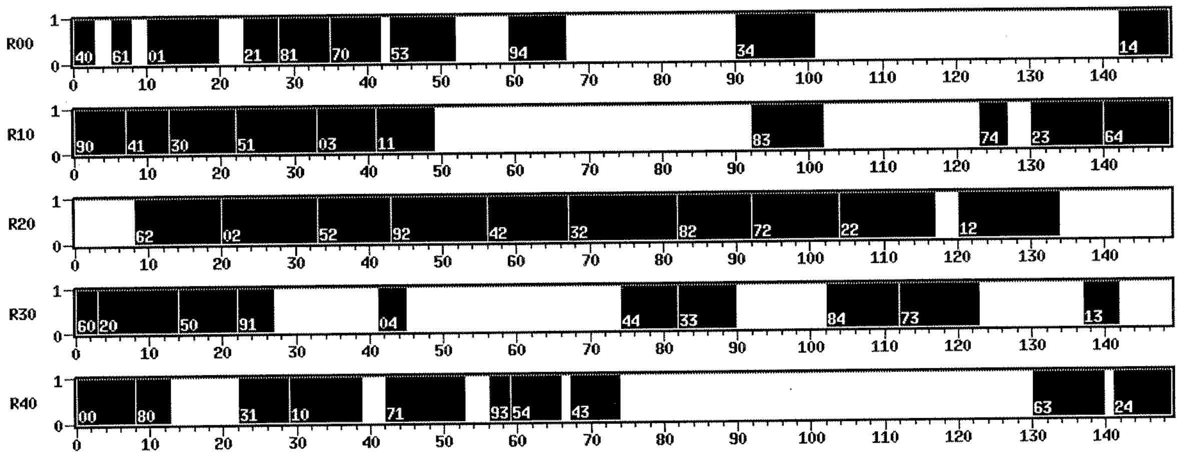

Figure 13. One solution to a problem (1-bottleneck, 0-looseness) from Sadeh's experimental suite.

For this problem, Resource R20 is the bottleneck: each job's third activity requires Resource R20, and each of these activities is relatively long in duration. This bottleneck will likely dictate the difficulty in finding a solution to the problem.

We have translated these 60 problems into ODO's problem description language and duplicated the experiments. Table 1 summarizes our results as compared to those reported in [Sadeh 91].[1] It should be noted that while our results are similar, they do not agree completely. These results however are not incompatible with other known efforts to duplicate the MicroBOSS heuristic [Sadeh 93].

Table 1. Number of Problems Solved (out of 10) for the six problem classes. W: wide looseness; N: narrow looseness; 0: zero looseness.

| | 1 Bottleneck | | | 2 Bottlenecks | | |
|---|---|---|---|---|---|---|
| | W | N | 0 | W | N | 0 |
| Sadeh MicroBOSS | 10 | 8 | 7 | 10 | 9 | 8 |
| ODO MicroBOSS | 9 | 7 | 7 | 8 | 7 | 6 |

1. We note that Sadeh has reported solving all problems using more sophisticated forms of backtracking [Xiong 92], which we have not implemented yet within ODO.

One point we can readily check is the impact of the arbitrary decisions made to keep the algorithm deterministic. We can simply substitute the word **`random`** for each occurrence of the word **`arbitrary`** in ODO's description of Micro-BOSS and observe the net effect over multiple runs. Table 2 compares the number of problems solved at least once in 20 runs when breaking ties randomly against the number of problems solved breaking ties arbitrarily. This outcome indicates the potential reliance of Sadeh's reported results on the arbitrariness of tie-breaking decisions.

Table 2. Number of Problems Solved (out of 10).W: wide looseness; N: narrow looseness; 0: zero looseness. When random tie-breaking is used, a problem is considered "solved" if a solution was found at least once in 20 runs.

| | **1 Bottleneck** | | | **2 Bottlenecks** | | |
|---|---|---|---|---|---|---|
| | W | N | 0 | W | N | 0 |
| Sadeh MicroBOSS (arbitrary tie breaking) | 10 | 8 | 7 | 10 | 9 | 8 |
| ODO MicroBOSS (arbitrary tie breaking) | 9 | 7 | 7 | 8 | 7 | 6 |
| ODO MicroBOSS (random tie breaking) | 9 | 7 | 7 | 9 | 8 | 7 |

### 6.1.3 Varying Consistency Enforcement

Our first (incorrect) policy implementation of the Micro-BOSS heuristic used a slightly different propagation scheme which yielded surprisingly different results. In it we used **`temporal_pv_unassigned`** instead of **`full_temporal_pv_unassigned`** for the temporal component of the propagation (see Section 5.2.2 for a description of these propagation methods). We reasoned that temporal propagation only needed to occur on those tasks that are temporally connected to the newly assigned task (i.e., those activities within the same job). This is what **`temporal_pv_unassigned`** does. In comparison, **`full_temporal_pv_unassigned`** performs a full temporal propagation across all activities of all jobs. We assumed that propagating via **`full_temporal_pv_unassigned`** instead of **`temporal_pv_unassigned`** would require extra effort without performing any additional propagation.[1] However, because MicroBOSS does *not* propagate resource constraints until full resource arc-consistency is achieved, there always exists the possibility that inconsistent possible values remain on some variables after propagation. Therefore, in the next iteration, **`full_temporal_pv_unassigned`** can prune more possible values left inconsistent from the previous iteration than **`temporal_pv_unassigned`** can.

1. It turns out that the extra computation required by **`full_temporal_pv_unassigned`** is minimal compared to the computation involved in the **`orr/fss`** variable/value selection process.

For 1-bottleneck problems, no real difference resulted. For the 2-bottleneck problems however, the different amount of temporal consistency enforcement made a great difference. Table 3 compares the results of using ODO's Micro-BOSS policy with the two temporal propagation methods.

Table 3. Number of Problems Solved (out of 10), with 20 tries.

| Temporal Consistency Method | 1 Bottleneck | | | 2 Bottlenecks | | |
|---|---|---|---|---|---|---|
| | W | N | 0 | W | N | 0 |
| full_temporal_pv_unassigned | 9 | 7 | 7 | 9 | 8 | 7 |
| temporal_pv_unassigned | 9 | 7 | 7 | 3 | 2 | 0 |

The differing sensitivity of the 1- and 2-bottleneck problems to this minor change in consistency checking suggests that extra consistency checking is useful in the more tightly constrained 2-bottleneck problems. We tested the possibility of improving Micro-BOSS's performance with slightly increased consistency checking beyond that used by Sadeh in his thesis. Here we altered temporal checking to the following:

```
propagation_method
         full_temporal_pv_unassigned
         unit_resource_unassigned
         full_temporal_pv_unassigned
         unit_resource_unassigned
         binary_resource_unassigned;
```

Table 4 compares the results of the normal propagation method against the results using this extra propagation. No change occurred with respect to the 1-bottleneck problems. The results of the 2-bottleneck problem however came as a bit of a surprise: we see a slight improvement for finding solutions to the 0-looseness problems, and a *degradation* of performance for the narrow-looseness problems.

Table 4. Number of Problems Solved (out of 10), with 20 tries, comparing ODO-MicroBOSS using the normal propagation method and ODO-MicroBOSS with extra propagation.

| ODO-MicroBOSS Propagation | 1 Bottleneck | | | 2 Bottlenecks | | |
|---|---|---|---|---|---|---|
| | W | N | 0 | W | N | 0 |
| Normal Propagation | 9 | 7 | 7 | 9 | 8 | 7 |
| Extra Propagation | 9 | 7 | 7 | 9 | 6 | 8 |

Ordinarily we do not anticipate that extra consistency enforcement should degrade search performance. We therefore sought to determine the cause of this result. The first step in doing so was to isolate precisely where the change was occurring. Table 5 enumerates for all 2-bottleneck problems the number of times a solution was found (out of 20) within 1000 iterations.

Table 5. Number Solved Using ODO's MicroBOSS, with 20 tries. "=" : Normal Propagation; ">" : Extra Propagation.

| | **2 Bottlenecks** | | | | | |
|---|---|---|---|---|---|---|
| | **W** | | **N** | | **0** | |
| **Problem #** | = | > | = | > | = | > |
| 1 | 18 | 20 | 0 | 0 | 20 | 19 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 20 | 20 | 20 | 20 | 9 | 20 |
| 4 | 20 | 20 | 20 | 20 | 20 | 20 |
| 5 | 20 | 20 | 20 | 20 | 20 | 20 |
| 6 | 14 | 9 | 20 | 20 | 0 | 20 |
| 7 | 13 | 19 | 20 | 20 | 20 | 20 |
| 8 | 20 | 20 | 12 | 0 | 20 | 20 |
| 9 | 20 | 20 | 20 | 0 | 0 | 0 |
| 10 | 20 | 20 | 20 | 20 | 20 | 20 |

Problems 8 and 9 with 2-bottleneck and narrow-looseness settings demonstrated the degradation most clearly. Problem 8's positive results for the normal setting were subject to arbitrary tie breaking, so we focused our analysis on Problem 9.

We traced the execution of MicroBOSS using both propagation methods on this problem. At the beginning of the second iteration, Activity 82 has 113 possible values with normal propagation and only 92 possible values with extra propagation. Consequently, different demand profiles are generated by the `orr` variable-selection texture measure. When the extra propagation is performed, `orr` determines that Activity 82 is the most-constrained task. Unfortunately, the value selected as best by the `fss` evaluation criteria for Activity 82 is not a good one; as a result, search thrashes until the iteration bound is reached. Using the normal propagation settings, `orr` selects Activity 14 (instead of Activity 82); when Activity 14's value is propagated, it prunes the "bad" Activity 82 value. When Activity 82 is later selected, it is ultimately assigned a good value, and a solution is found with minimal backtracking.

This example suggests that it is not always the most-constrained variable that should be selected for assignment. Rather, variable selection should perhaps be biased by the heuristic's ability to select a good value for that variable. Separate variable and value selection, as performed in MicroBOSS, misses this opportunity. We plan to look further into this issue in the future.

## 6.2 ODO as Min-Conflicts (and Variants)

As described previously, Min-Conflicts is a repair-based heuristic that essentially performs greedy search. Its variable selection consists of randomly selecting a task with a violation and assigning it a value that minimizes the overall number of violations. Min-Conflicts' nondeterministic character helps it avoid cycling during search.

The designers of the Min-Conflicts heuristic have stressed the importance of a good initial starting state from which repairs are to begin [Minton 92]. What defines a good starting state remains an issue. For the moment we will bypass trying to find a good initial start, since we can see some interesting experimental results from using a random initial start (i.e., a starting state where all variables are given random values). We can also relate this to GSAT's repair-based approach to solving satisfiability problems, which does begin with a random assignment to all variables [Selman 92].

### 6.2.1 Generating a Random Initial Assignment

The generation of a random initial assignment is performed using the constructive component of ODO. Here unassigned variables are selected randomly, and they are assigned random values:

```
cost_function
         num_unassigned_vars;
var_selection
         unassigned
         random;
val_generation
         all_pv;
scoring_function
         none;
selection_function
         random;
propagation_method
         temporal_pv_unassigned;
accept_criteria
         always;
search_termination_criteria
         cost == 0;
```

Note that temporal consistency is enforced in this initial setup. In our current implementation, we restrict ourselves only to repairing resource violations. We will allow repairs of temporal constraints in future experiments.

When the constructive setup phase terminates, all variables have values that are at least temporally consistent. In any reasonably difficult scheduling problem, we still expect for many resource constraints to remain violated. Thus some repair mechanism can be used to resolve the remaining violations.

### 6.2.2 Min-Conflicts Repair

We declare the following to emulate Min-Conflicts within ODO. At each repair step, a variable participating in a violated constraint is selected (breaking ties randomly), and a new value is assigned to that variable that minimizes the resulting number of conflicts (breaking ties randomly):

```
cost_function
      num_violated_constraints;
var_selection
      violated
      random;
val_generation
      all_pv;
scoring_function
      cost_lookahead;
selection_function
      min_score
      random;
propagation_method
      temporal_v_assigned;
accept_criteria
      always;
search_termination_criteria
      (cost == 0) || (search_iterations == 1000);
```

The value for the search iteration bound entered here has no particular significance.

### 6.2.3 GSAT-like Min-Conflicts Repair

GSAT is a hill-climbing repair heuristic that has performed well on random satisfiability problems [Selman 92] . A satisfiability problem is a propositional formula in conjunctive normal form (i.e. a conjunction of clauses, where each clause is a disjunction of literals, and a literal is a propositional variable or its negation); a solution to the problem is a truth-value assignment to all variables such that the formula is satisfied. GSAT's hill-climbing search strategy can be paraphrased as follows:

- Begin search with a random truth assignment to all variables.
- At each iteration, reverse the value for the variable that results in the greatest number of satisfied clauses (breaking ties randomly).
- If an iteration bound has been reached without finding a complete satisfying assignment, generate a new random initial state and restart search.

We can consider the satisfiability problem as a CSP by mapping each propositional variable into a CSP variable with domain T or F, and then attaching a constraint between the variables in an individual clause. Each constraint is satisfied only if one of the literals in the clause evaluates to T.

When we view GSAT's hill-climbing mechanism in the context of the general CSP, we see that it is a form of lookahead similar to Min-Conflicts. GSAT performs a greater amount of lookahead than Min-Conflicts does, but in all other respects the heuristics are identical. It is worthwhile to try a GSAT-like variation of Min-Conflicts and see if (in the context of constraint-based scheduling) the extra lookahead is worth its computational expense. In ODO's policy representation we only have to change two of Min-Conflicts' declarations to create a GSAT-like variant:

```
var_selection
      all;
val_generation
      all_but_current_pv;
```

We can see that GSAT performs a full one-step lookahead. The `all_but_current_pv` value generation is a subtle variation on Min-Conflicts' `all_pv` value generation. GSAT can ignore the current assignment since it evaluates all possible problem states one step away. If the current state is in fact a true local minima, then GSAT will perform a repair that will create a condition with *more* violations than the current state. How much these violation-increasing repairs contribute to the success of the GSAT heuristic on satisfiability problems is not well understood [Mitchell 93]. Since Min-Conflicts always evaluates the selected variable's current value, there does not exist a possibility of selecting a repair that will create more violations. Without any better choices, the current value will be selected. Even when the current value is chosen and no change has been performed, the next iteration has the opportunity to select a different variable to repair.

### 6.2.4 Other Variants on Min-Conflicts

We can easily consider some other minor variants to the default settings for Min-Conflicts in ODO.

- **Random variable selection** - Instead of selecting a violated variable, we can simply select a random variable. In some cases, a variable that is not violated may be located at a time that prevents violated variables from moving in a direction that leads to a solution. It may benefit us to allow the selection of variables that are not violated as well.[1] Our only change to Min-Conflicts is:

  ```
  var_selection
        random;
  ```

- **Favor earliest/latest times when breaking ties** - when considering how to break ties during the evaluation phase, break ties using the earliest and latest actual values. This has the net effect of pushing tasks up against each other, which may be necessary to organize a bottleneck resource efficiently. We enable this within ODO by changing the Min-Conflicts evaluation criteria to the following:

  ```
  scoring_function
        min_score
        earliest_latest_value
        random;
  ```

Figure 14 provides an example of the precise behavior of `earliest_latest_value` when used as a scoring filter. In this example, three activities (01, 02, and 03) each require the use of Resource R1. There are no temporal constraints between these three activities. Note that in the starting state, Activity 01 and Activity 02 are violated (since they overlap each other). The starting cost = 2.

1. It has been shown in [Papadimitriou 91] that 2-satisfiability problems (i.e. satisfiability problems where all clauses have two literals) can be solved using random variable and value selection in $O(n^2)$ expected time.

Suppose we use the standard Min-Conflicts settings to repair the violation, except we use the scoring function mentioned above. Now suppose that in the first iteration, Activity 02 is chosen during variable selection. Activity 02's start-time variable has 7 possible values (0-6). Each of these candidate start-times is evaluated; we note that start-times 0 and 1 result in a cost of 3, and start-times 2 through 6 result in a cost of 2. The **`min_score`** scoring function therefore returns start-times 2 through 6. The **`earliest_latest_value`** scoring function chooses the earliest and latest of these actual values: namely 2 and 6. One of these is chosen randomly as the final selected value for repair.

By closely examining Figure 14, we see that when a start-time of 2 or 6 is selected for Activity 02, there is an opportunity to resolve all violations by moving Activity 01 in the next iteration. The same could not be said if a start-time of 3, 4, or 5 was selected instead. The default Min-Conflicts value-selection phase does not distinguish between start-times 2 through 6; however by adding the **`earliest_latest_value`** tie breaking mechanism, we focus repairs to those opportunities that may better organize activities using bottleneck resources for later iterations.

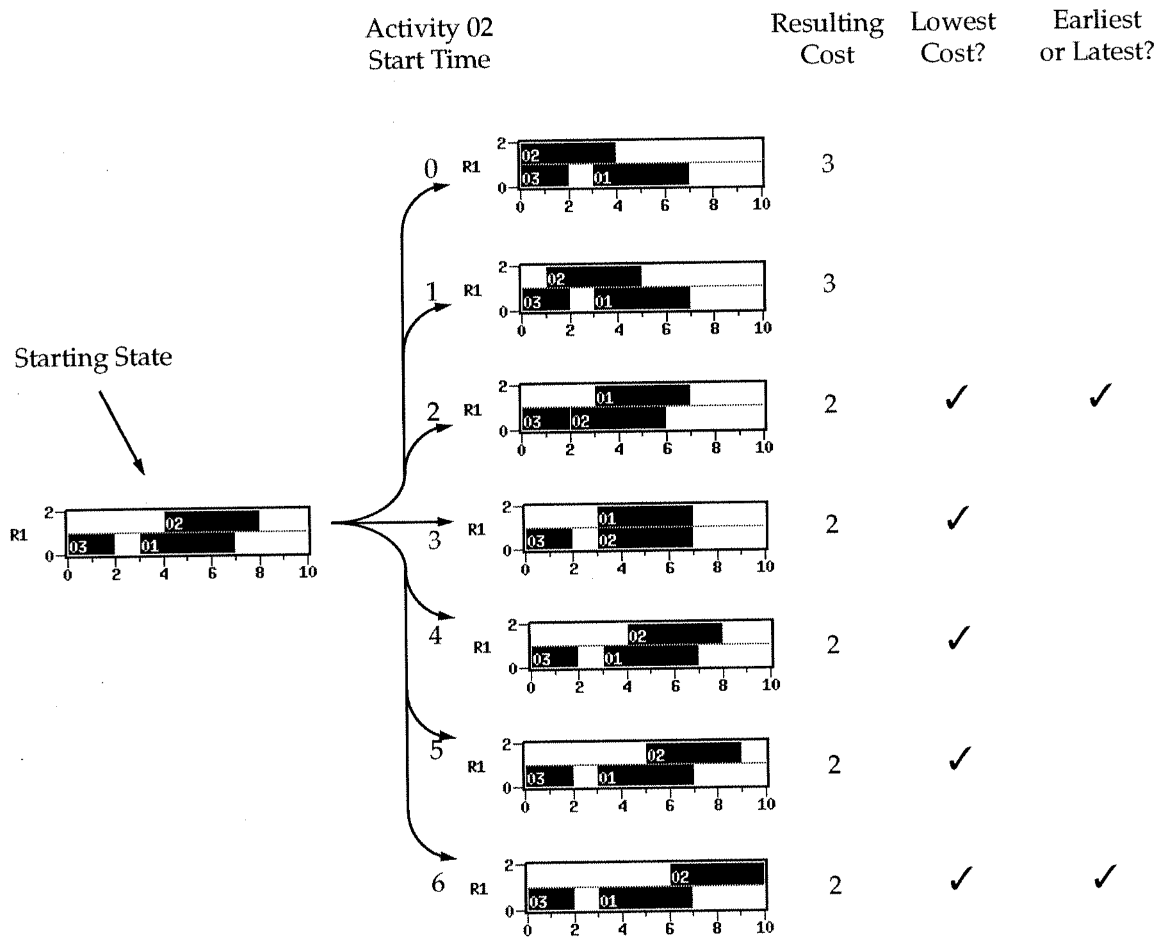

Figure 14. Demonstration of **`earliest_latest_value`** scoring function for tie breaking.

These above 2 variations can be performed independently or together. In addition, we can use the **`earliest_latest_value`** variant with our GSAT-like variable/value generation settings.

### 6.2.5 Min-Conflicts and Variants with Random Starts

Table 6 shows the results for Min-Conflicts and these variants on the benchmark problem shown in Figure 13 (1 bottleneck, 0 looseness) from a random start. The default Min-Conflicts heuristic is reflected in the first row of the table. The GSAT-like heuristic is found in the fifth-row. Each experiment was run 10 times for each of 10 random initial setups. Search terminated after 1000 iterations. The mean number of iterations is reported only for those problems solved by the iteration bound.

Table 6. Results from performing Min-Conflicts and variants from a random start for one 1-bottleneck/0-looseness problem. Number solved is reported out of 100 runs (10 different random setups, 10 runs each, 1000 iterations per run).

| Var Selection | Val Generation | Early/Late tie breaking? | Num Solved | Mean Iterations | Avg.Time/ Iteration (Seconds)[a] |
|---|---|---|---|---|---|
| violated random | all_pv | N | 7 | 393 | 0.7 |
| | | Y | 9 | 107 | 0.7 |
| random | all_pv | N | 8 | 644 | 0.7 |
| | | Y | 53 | 572 | 0.7 |
| all | all_but_ current_pv | N | 15 | 426 | 32 |
| | | Y | 77 | 357 | 32 |

a. See the appendix for a description of ODO's hardware and software environment.

We first observe that none of these repair heuristics could reliably find a solution from a random start in 1000 iterations. Second, the domain-specific bias to break ties using the earliest and latest values seems to improve all heuristics, not only in the number of solutions found but also in the average number of iterations required to find solutions. Further, we see that `random` variable selection generally improves the likelihood for finding a solution over `violated random` variable selection; however, the number of iterations required by `random` variable selection is noticeably higher. Hence there exists a trade-off between selecting the more focused `violated random` variable selection, which, if it finds a solution will do so relatively quickly, against the more reliable but slower `random` variable selection.

Finally, it appears that the full 1-step lookahead variable selection (`all/all_but_current`) is too computationally expensive relative to the added power it provides. We have found that the lookahead process dominates computation time for all of these heuristics. When a problem consists of 50 activities, we can expect the full lookahead to require approximately 50 times more computation per iteration over a method that performs lookahead on one activity only. Since we do not observe a 50-fold improvement in search efficiency using full lookahead, we consider the other variable selection methods to be more economic choices.

Though our GSAT-like version of Min-Conflicts does not appear to be any more promising than Min-Conflicts itself, we should not discount the possibility that the original GSAT heuristic may perform well on a version of the scheduling problem represented as a satisfiability problem. GSAT has been shown to efficiently solve n-queens and graph-colorability problems converted into satisfiability problems [Selman 92]. However, GSAT required modification before it could efficiently solve a planning problem [Kautz 92] [Selman 93]. GSAT's performance (with or without heuristic modifications) on satisfiability-translated scheduling problems remains to be assessed.

Table 7 reports results using the `violated random` and `random` variable selection for representative problems of the six problem classes in the test suite. Not all problems are equally hard for the Min-Conflicts-like heuristics. In general, we observe that for more constrained problems (more bottlenecks and/or less looseness), the repair heuristics are less likely to find a solution. Again, we note that the bias to favor earliest and latest values for tie-breaking purposes almost always improves search performance. For the "easiest" problem class (wide looseness and one bottleneck), the `random` setting with early/late tie-breaking found a solution 98 times out of 100. Still, we might prefer to use the `violated random` setting with early-late tie-breaking instead since a solution is found half as often but (on average) five times faster.

Table 7. Results using the `violated random` and `random` variable selection for representative problems of each class (10 different random setups, 10 runs each, 1000 iterations per run).

| Problem Type | | Policy settings | | Results | |
|---|---|---|---|---|---|
| Looseness | # Btlnecks | Var Selection | Early/Late tie breaking? | Num Solved | Mean Iter. |
| 0 | 1 | violated random | N | 7 | 393 |
| | | | Y | 9 | 107 |
| | | random | N | 8 | 644 |
| | | | Y | 53 | 572 |
| | 2 | violated random | N | 0 | - |
| | | | Y | 1 | 128 |
| | | random | N | 0 | - |
| | | | Y | 4 | 608 |
| N | 1 | violated random | N | 53 | 193 |
| | | | Y | 49 | 79 |
| | | random | N | 49 | 479 |
| | | | Y | 91 | 322 |
| | 2 | violated random | N | 6 | 233 |
| | | | Y | 4 | 171 |
| | | random | N | 3 | 657 |
| | | | Y | 9 | 690 |
| W | 1 | violated random | N | 66 | 155 |
| | | | Y | 53 | 44 |
| | | random | N | 59 | 428 |
| | | | Y | 98 | 270 |
| | 2 | violated random | N | 22 | 253 |
| | | | Y | 28 | 78 |
| | | random | N | 23 | 655 |
| | | | Y | 57 | 516 |

## 6.3 Min-Conflicts and the Horizon Effect

The good start that the Min-Conflicts heuristic informally specifies appears necessary for solving the more difficult of Sadeh's test problems. Though a good start is not well defined, it essentially implies that, with respect to the heuristic, we are near a solution. The concept of "nearness to solution", if it could be captured, could also be usefully applied within search termination conditions. Repair-based search methods such as Min-Conflicts and GSAT typically use a prespecified iteration bound to determine whether to terminate the existing search procedure. We assert that other efficient measures exist that can indicate whether the current search method is near or far from a solution. This is motivated by the example problem states shown in Figures 15 and 16.

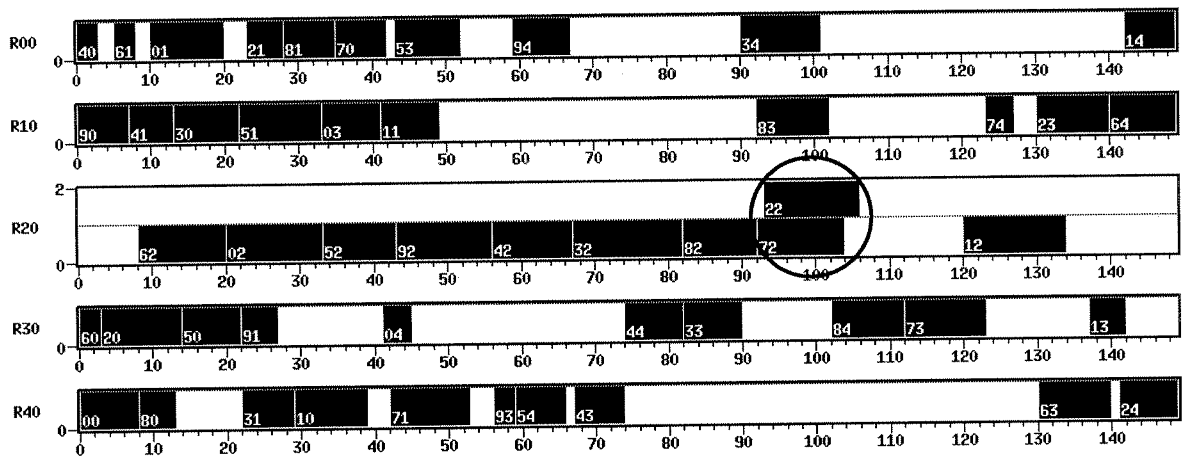

Figure 15. Very near a solution for Min-Conflicts.

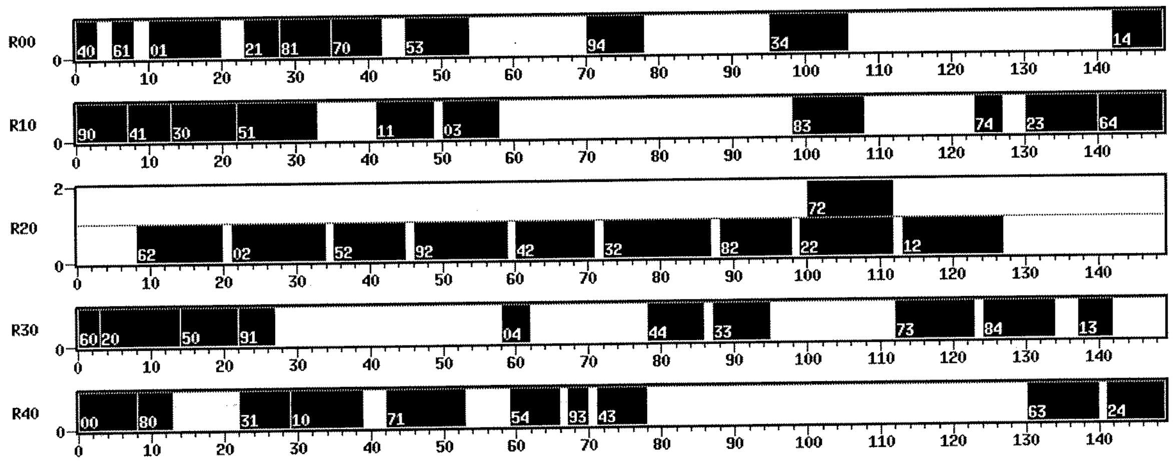

Figure 16. Very far from a solution for Min-Conflicts.

In the first case (Figure 15), we appear to be very close to a solution if we are using the Min-Conflicts heuristic. If Activity 22 (circled) is selected during variable selection, then it will be placed in the available spot beginning at time 104 and search will terminate.

In the second case (Figure 16) we seem to be very far from a solution if we use Min-Conflicts. This is because there is no single available spot for either violated task, or for any other task on this resource for that matter. Somehow the gaps between tasks must be consolidated during Min-Conflicts' repair process before the violation can be resolved.

The differences between the schedules shown in Figures 15 and 16 capture intuitively what we might consider one difference between a good and a bad initial start. In addition, we might wish to terminate search and start over if we found ourselves in Figure 16's schedule state, whereas we probably want to continue search from Figure 15's schedule state. The concepts of initial start and termination conditions are further connected when we consider that we would like to terminate search if we have an expectation that starting over (with some "good" start state) places us in a problem state that is nearer to a solution.

We considered the possibility that ODO's MicroBOSS policy might be useful in creating a good initial starting state. The overall search strategy would then be to run MicroBOSS until it either solves the problem or reaches an iteration bound; if the bound is reached, then the existing partial solution could be used as a basis for creating an initial start for a repair policy. We quickly discovered however that whenever MicroBOSS did not find a solution by the iteration bound, its problem state was relatively high in the search tree; consequently, perhaps only ten of the fifty activities remained assigned at the iteration bound. Before any repairs could begin, the remaining activities would have to be placed using some alternate policy. Our results using various alternate policies were inconclusive; when we attempted to generate a good initial start we found an actual solution to the given problem more often than we established a repair situation that Min-Conflicts could resolve.

We then constructed general texture measures that would possibly indicate when Min-Conflicts was either near or far from a solution. These measures ranged from the fragmentation of resource utilization (where a state such as that shown in Figure 16 would register poorly for Resource R20) to the number of unique moves available to a violated activity that would not increase the number of violated constraints. To date we have found no clear correlation between our texture measures and a probability of finding a solution.

The examples shown in Figures 17 and 18 underscore the subtleties that can dictate whether or not a solution can be found within a few iterations. Figure 17 shows a partially violated schedule that admits a solution to Min-Conflicts readily. Figure 18, however, shows a schedule which does not yield a solution to Min-Conflicts for many iterations, even with multiple restarts. The only differences between the two problem states are the relative positions of Activity 60 and Activity 20 on Resource R30 (circled).

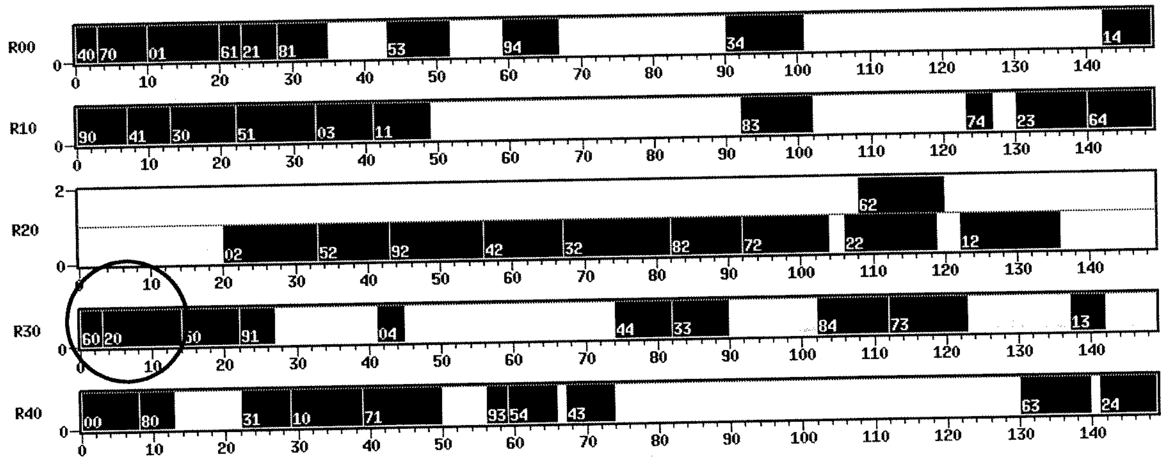

Figure 17. A few steps from solution using Min-Conflicts.

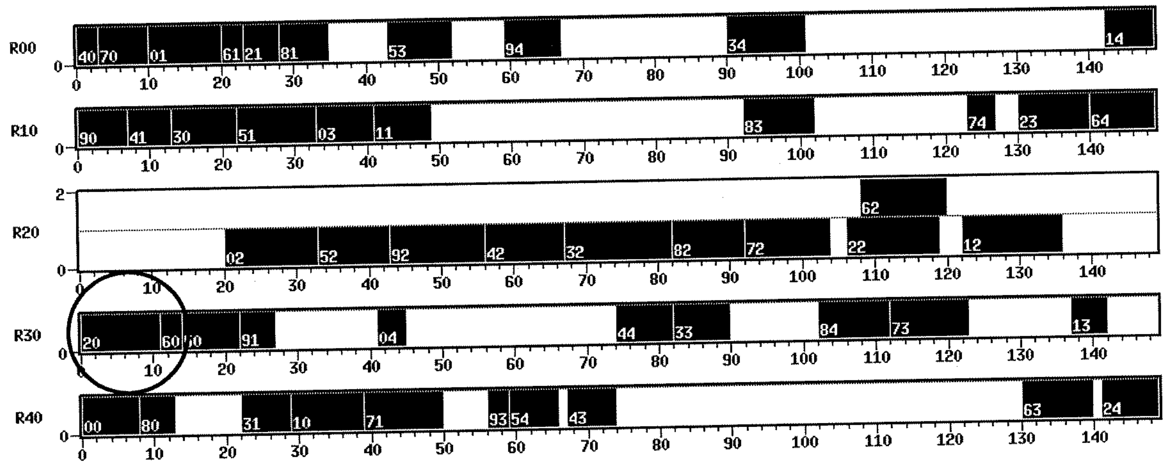

Figure 18. Far from a solution using Min-Conflicts.

When we analyze these two problem states with respect to Min-Conflicts' search heuristic, we see why the small change makes a big difference. Min-Conflicts is able to quickly find a solution if it can place Activity 62 at time 6. Since we enforce temporal constraints, Activity 60 and Activity 61 are subject to value propagation when Activity 62 is moved to that time. However, Min-Conflicts will not allow a move that will cause more violations than it repaired. Therefore, a move that repairs Activity 62's violation (and simultaneously Activity 22's violation) can only do so if it creates less than 2 violations as a result. For the schedule in Figure 17, moving Activity 61 to time 6 will cause the violations of Activity 61 and Activity 70. Since the number of violations remains the same, Min-Conflicts will allow this move. However, for the schedule in Figure 18, moving Activity 62 will cause two new violations with each of Activity 61 and Activity 60. Min-Conflicts will never directly perform this move; consequently the second state is much further from a solution.

We found this interaction by looking several steps ahead into the search process Min-Conflicts was likely to execute. Since Min-Conflicts only evaluates states that are one repair away, these kinds of interactions are beyond its detection. This is an example of the *horizon effect* [Berliner 73], a phenomenon which occurs to heuristic methods with limited information.

### 6.3.1 One-Resource Experiments

As we have noted before, Min-Conflicts does not perform a full one-step lookahead. Consequently, the horizon effect can become apparent within one search step: Min-Conflicts may overlook an available repair that would solve the problem. The missed opportunity may not heavily impact search convergence if there is a reasonable probability that a solution will be found within a small number of iterations anyway.

We performed experiments to determine Min-Conflicts' likelihood for finding a solution when search began with a solution one repair away. To simplify analysis, we focused on one-resource problems of the kind that would represent a subset of one of Sadeh's test problems. As we shall see for even these problems, it is possible for Min-Conflicts to begin search one repair away from a solution but still not find any solution for 1000 iterations or more. This is because the solution may only be found if a *particular* violated variable is selected for repair, and not just any of the violated variables. Figure 19 shows a graph of state-space transformations which illustrates this possibility.

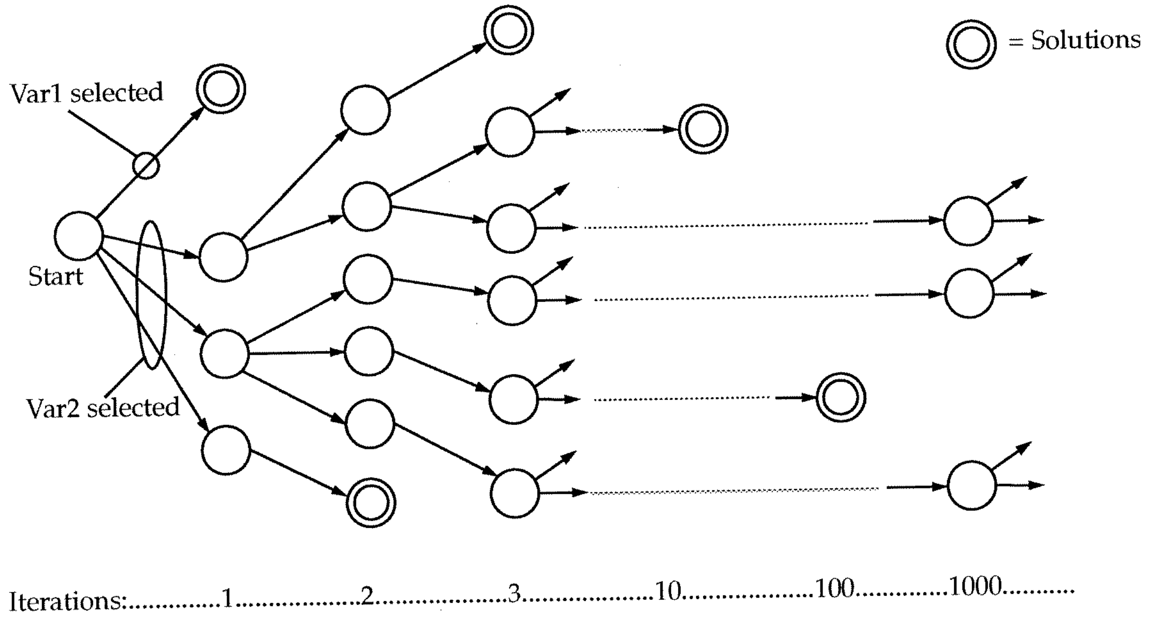
= Solutions
Var1 selected
Start
Var2 selected
Iterations:..............1..................2...................3...................10...................100.............1000...........

Figure 19. Transition graph demonstrating that even though a solution can be found within one step from the starting state, a solution may not be found after many iterations.

We generated six 11-activity/1-resource problems. The distribution of activity durations varied, but all durations in each problem summed to 110. By performing the following steps, the problems were randomly initialized to a state that was one repair away from solution:

- Ten of the eleven activities were placed in a manner consistent with respect to all constraints. In addition, the tasks were placed as compactly as possible so that a gap remained in the resource large enough to accommodate the eleventh task.
- The eleventh activity was placed randomly instead of into the available resource gap.

Figure 20 shows one initial setup. Activity 07 is the randomly placed task; both Activity 07 and Activity 10 are violated as a result. If we use the Min-Conflicts heuristic to find a solution, then the variable selection phase will select either Activity 07 or Activity 10. If Activity 07 is selected, a solution is found this iteration. If Activity 10 is selected instead, however, no solution can be found this iteration. Even so, we would still hope to find a solution within a small number of steps.

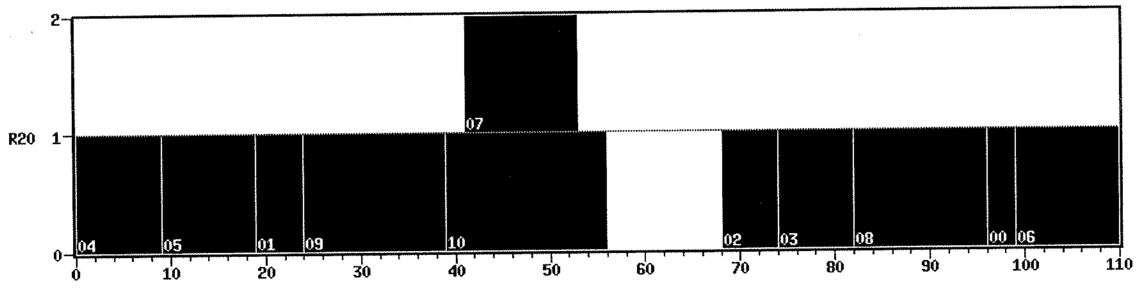

Figure 20. Sample initial setup for 1-resource problems.

We ran the Min-Conflicts heuristic on each of the six problems 100 times, using a different random initial setup each time, and terminating search at 1000 iterations if a solution had not been found. We set each problem's due date to 120, 115, and 110. Figure 21 shows for each of these due date settings the average percent of all problems having found a solution as a function of iteration count.

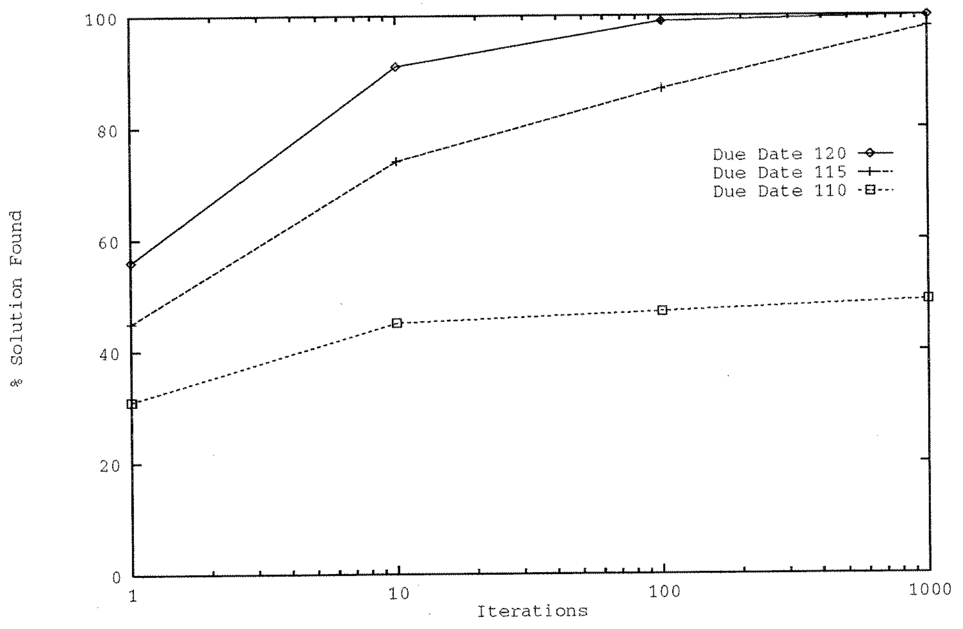

Figure 21. Average % of problems with solution as a function of iteration count.

For the loosest due date setting tested (120), solutions are almost always found within a few iterations. When the due date is shortened to 115, fewer problems are solved within a reasonable number of iterations. Finally, with the tightest due date, we see that despite finding solutions over 40% of the time in the first ten iterations, only about 5% more can be solved in the next 990 iterations.

The data for the 110-due-date problems underscores (at least when a problem contains a tight bottleneck) the importance of a good start for a heuristic such as Min-Conflicts. Min-Conflicts' cost measure — the number of violated constraints — cannot seem to keep the search process within short reach of a solution. Even though we know that the number of violations never increases, we seem to be (probabilistically speaking) wandering further from finding a solution. For such a problem the successful overall search strategy may depend upon multiple restarts from good starting states.

This data also suggests that, if we know we can generate good starting states, perhaps a broader lookahead method should be preferred. For example, in the above 11-task problems, if the variable selection process chose all violated tasks and not just one random violated task, then a solution would have been found in one iteration every time.

These observations emphasize the value that cost-effective texture measures can provide to a search heuristic. Appropriate measures can potentially prescribe necessary and sufficient bounds on decision informedness. The determination of these measures remains a focus of our research.

## 6.4 Summary

This chapter has demonstrated ODO's capabilities in emulating various scheduling heuristics within its policy model. When we perform experiments on these policies, we start to see some of the relationships between problem structure, heuristic method, and search performance.

Our experiments have shown MicroBOSS' sensitivity to the amount of constraint propagation performed. Slight alterations in the degree of consistency enforcement caused noticeable changes in the number of problems that could be solved from the Sadeh experimental suite.

We have also shown that Min-Conflicts (and related repair-based heuristics) do not quickly solve solutions to Sadeh's problems when given a random initial start. Reasons for why a schedule may or may not be easy to repair can be subtle; consequently it remains a difficult issue to efficiently determine a given problem-state's nearness to solution with respect to the Min-Conflicts heuristic. Further, we have seen that if Min-Conflicts is given a problem state one repair away from solution, an actual solution may not be found in many iterations.

# Chapter 7 Concluding Remarks

## 7.1 Summary

In this thesis we have presented a model for constraint-based scheduling that is capable of capturing many known scheduling heuristics. This model views problem solving as transformational search through a set of problem-solving states, where transformations are structured by either constructive or repair-based policies. Policy decisions are based upon constraint graph measures called textures. The policy structure isolates components of a heuristic search loop that includes variable and value selection, constraint propagation, intermediate acceptance, and search termination.

We also described ODO, a constraint-based scheduling system that implements a portion of our model. ODO represents the scheduling problem as a constraint graph, and performs structured search within our policy. ODO accepts as input both the problem to be solved and the parameters for which search is to be performed.

Using ODO we reconstructed two well-known scheduling heuristics — one constructive, the other repair-based. We conducted experiments to verify ODO's competence at emulating these heuristics and to test the sensitivity of these heuristics to small changes in some facet of the heuristic. These results demonstrate a scheduler's sensitivity to the exact amount of propagation being performed, and show how some repair techniques can experience poor performance due to the limited information available for decision-making.

## 7.2 Conclusions and Future Research

This thesis represents a first step into unifying the many aspects of constraint-based scheduling. Our model captures many known heuristics, and isolates some of their essential differences. The next step is to perform a thorough analysis relating graph-based textures of scheduling problems to the performance of heuristic search. We are especially interested in creating a more symbiotic relationship between constructive and repair-based search strategies. For example, a constructive search process could be interrupted for intermediate repairs before resuming. Other promising avenues include associating the proper amount of consistency enforcement for a given problem class, and determining efficient measures of approximate distance to solution.

The ODO scheduler is employed in research other than that described in this thesis. In addition, it is being used for research in constraint relaxation [Beck 94], and in agent-oriented decision-making in manufacturing [Fox 92].

We continue to enhance ODO's problem representation and problem solving capabilities. In the near term, we anticipate adding the following functionality:

- **More consistency checking** - To date we have only inserted propagation methods as they were needed. We plan to incorporate a more complete consistency-checking library within ODO. Recent results presented in [Nuijten 93] indicate that extensive consistency checking can be beneficial to efficient search. The time/value trade-off of this extra propagation is worth further investigation.
- **Optimization constraints** - Optimization constraints can add realism to the job shop model. Of particular interest are activity tardiness and inventory constraints.
- **Other commitment methods** - Beyond variable and value selection, we wish to implement constraint posting as an additional commitment method [Muscettola 93] .
- **Different backtrack schemes** - We plan to incorporate a library of backtracking mechanisms, including backjumping and perhaps dynamic backtracking [Ginsberg 93] .
- **Iterative side effects** - In repair-based heuristics such as those described in [Morris 93] [Selman 93] , constraint "weights" can be altered at each iteration. We plan to add a component to the declaration of search that allows for these types of side effects.

We see many possible avenues for more long-term research:

- **Building robust schedules** - Many scheduling domains operate under uncertain conditions. Machines may break down, unanticipated work may be required, etc. We prefer schedules that can absorb these uncertainties with minimal cost. We believe that the constraint model is adequate to represent these types of uncertainties, and that heuristics can be developed to build more robust schedules.
- **Problem reformulation** - Much research into has gone into problem reformulation techniques such as automated abstraction and relaxation. We assert that the scheduling domain can benefit from such strategies. We anticipate that problem reformulation methods are compatible with our current commitment-based problem solving model.
- **Machine learning** - We might be able to create conditions for which a scheduler itself may be able to determine an efficient heuristic for a problem solving domain. The potential for such heuristics has been shown in [Zweben 92a] [Minton 93] .

# Bibliography

[Adams 88] Adams, J., Balas, E. and Zawack, D., The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science* **33** (3):391-401, 1988.

[Allen 84] Allen, J., Towards a General Theory of Action and Time. *Artificial Intelligence* **23** 123-154, 1984.

[Baker 92] Baker, K., Elements of Sequencing and Scheduling. 1992.

[Beck 94] Beck, C., *Efficient Constraint Relaxation with Applications to Supply Chain Management.* Master's thesis, University of Toronto, 1994. To Appear.

[Berliner 73] Berliner, H., Some Necessary Conditions for a Master Chess Program. *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 77-84, 1973.

[Bitner 75] Bitner, J. and Reingold, E., Backtrack Programming Techniques. *Communications of the ACM* **18** (11):651-656, November, 1975.

[Brelaz 79] Brelaz, D., New Methods to Color the Vertices of a Graph. *Communications of the ACM* **22** (4):251-256, April, 1979.

[Cormen 91] Cormen, T., Leiserson, C., Rivest, R., *Introduction to Algorithms.* MIT Press, 1991.

[CPLEX 92] *Using the CPLEX Callable Library and CPLEX Mixed Integer Library.* CPLEX Optimization, Inc., 1992.

[Davis 87] Davis, E., Constraint Propagation with Interval Labels. *Artificial Intelligence* **32** 281-331, 1987.

[Deale 94] Deale, M., Yvanovich, M., Schnitzius, S., Kautz, D., Carpenter, M., Zweben, M., Davis, G., Daun, B. and Drascher, E., The Space Shuttle Ground Processing Scheduling System. *Intelligent Scheduling.* Morgan Kaufman, 1994. (to appear).

[Dechter 88] Dechter, R. and Pearl, J., Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* **34** 1-38, 1988.

[Dechter 89] Dechter, R. and Meiri, I., Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* 1989.

[Doyle 79] Doyle, J., A Truth Maintenance System. *Artificial Intelligence* **12** (3):231-272, 1979.

[Drummond 93] Drummond, M., Swanson, K., Bresina, J. and Levinson, R., Reaction

First Search. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. 1993.

[Fox 87] Fox, M., *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufman Publishers, Inc., 1987.

[Fox 89] Fox, M., Sadeh, N. and Baykan, C., Constrained Heuristic Search. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* 1989.

[Fox 92] Fox, M., *The TOVE Project: Towards a Common Sense Model of the Enterprise.* Technical Report, Department of Industrial Engineering, University of Toronto, April, 1992.

[Freuder 78] Freuder, E., Synthesizing Constraint Expressions. *Communications of the ACM* **21** 958-966, 1978.

[Freuder 82] Freuder, E., A Sufficient Condition for Backtrack Free Search. *Journal of the ACM* **29** (1):24-32, 1982.

[Freuder 92] Freuder, E. and Wallace, R., Partial constraint satisfaction. *Artificial Intelligence* **58** 21-70, 1992.

[Garey 79] Garey, M. and Johnson, D. *Computers and Intractability.* W. H. Freeman and Co., 1979.

[Gaschnig 77] Gaschnig, J., A General Backtrack Algorithm that Eliminates Most Redundant Tests. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.* 1977.

[Gaschnig 78] Gaschnig, J., Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisficing Assignment Problems. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence.* 1978.

[Ginsberg 93] Ginsberg, M., Dynamic Backtracking. *Journal of Artificial Intelligence Research* **1** 25-46, 1993.

[Goldratt 90] Goldratt, E., *The Haystack Syndrome.* North River Press, 1990.

[Golomb 65] Golomb, S. and Baumert, L., Backtrack Programming. *Journal of the ACM* **12** (4):516-524, 1965.

[Haralick 80] Haralick, R. and Elliott, G., Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* **14** 263-313, 1980.

[Hillier 90] Hillier, F.S. and Lieberman, G.J., *Introduction to Operations Research.* Holden-Day, Inc., San Francisco, CA, 1990.

[IJCAI 93] Sadeh, N. (editor), *IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling, and Control.* 1993.

[Johnston 89] Johnston, M. Knowledge-Based Telescope Scheduling. *Knowledge-Based Systems in Astronomy.* Springer-Verlag, 1989.

[Kautz 92] Kautz, H. and Selman, B., Planning as Satisfiability. *Proceedings of the Tenth European Conference on Artificial Intelligence.* 1992.

[Keng 89] Keng, N. and Yun, D., A Planning/Scheduling Methodology for the Constrained Resource Problem. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* 1989.

| | |
|---|---|
| [Kfoury 82] | Kfoury, A., Moll, R. and Arbib, M., *A Programming Approach to Computability.* Springer-Verlag, 1982. |
| [Kirkpatrick 83] | Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., Optimization by Simulated Annealing. *Science* **220** (4598), 1983. |
| [Knuth 75] | Knuth, D., Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation* **29** 121-136, January, 1975. |
| [Kumar 92] | Kumar, V., Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine* **13** (1):32-44, 1992. |
| [Laird 87] | Laird, J., Newell, A. and Rosenbloom, P., SOAR: An Architecture for General Intelligence. *Artificial Intelligence* **33** 1-64, 1987. |
| [Langley 92] | Langley, P., Systematic and Nonsystematic Search Strategies. *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*. 1992. |
| [Le 91] | Le Pape, C., *Constraint Propagation in Planning and Scheduling.* Technical Report, Robotics Laboratory, Stanford Universtity, January, 1991. |
| [Mackworth 77] | Mackworth, A., Consistency in Networks of Relations. *Artificial Intelligence* **8** 99-118, 1977. |
| [Mackworth 86] | Mackworth, A., Constraint Satisfaction. *Encyclopedia of Artificial Intelligence.* Addison-Wesley, 1986, pages 205-211. |
| [Minton 92] | Minton, S., Johnston, M., Philips, A. and Laird, P., Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* **58** 161-205, 1992. |
| [Minton 93] | Minton, S., Integrating Heuristics for Constraint Satisfaction Problems: A Case Study. *Proceedings of the Eleventh National Conference on Artificial Intelligence*. 1993. |
| [Mitchell 93] | Mitchell, D., Personal communication, 1993. |
| [Mittal 88] | Mittal, S. and Falkenhainer, B., Dynamic Constraint Satisfaction Problems. *Proceedings of the Eighth National Conference on Artificial Intelligence.* 1988. |
| [Morris 93] | Morris, P., The Breakout Method for Escaping from Local Minima. *Proceedings of the Eleventh National Conference on Artificial Intelligence.* 1993. |
| [Muscettola 93] | Muscettola, N., Scheduling by Iterative Partition of Bottleneck Conflicts. *Proceedings of the Ninth Conference on Artificial Intelligence for Applications.* 1993. |
| [Nadel 89] | Nadel, B., Constraint satisfaction algorithms. *Computational Intelligence* **5** 188-224, 1989. |
| [Nilsson 71] | Nilsson, N., *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971. |
| [Nuijten 93] | Nuijten, W.P.M., Aarts, E.H.L., van Erp Taalmen Kip, K.M. and van Hee, K.M., Randomized Constraint Satisfaction for Job Shop Scheduling. *IJCAI-93 Workshop on Knowledge-based Production Planning, Scheduling, and Control*. 1993. |
| [Papadimitriou 91] | Papadimitriou, C., On selecting a satisfying truth assignment. *Proceed-* |

*ings of the 32nd Symposium on the Foundations of Computer Science*. 1991.

[Prosser 93] Prosser, P., Domain filtering can degrade intelligent backtracking search. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. 1993.

[Purdom 83] Purdom, P., Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* **21** 117-133, 1983.

[Sadeh 91] Sadeh, N., *Lookahead Techniques for Micro-Opportunistic Job Shop Scheduling*. PhD thesis, Carnegie Mellon University, 1991. CMU-CS-91-102.

[Sadeh 93] Sadeh, N., Personal communication, 1993.

[Saks 93] Saks, V., Johnson, I. and Fox, M., Distribution Planning: A Constrained Heuristic Search Approach. *Proceedings of the DND Workshop on Knowledge-based Systems and Robotics*. 1993.

[Selman 92] Selman, B., Levesque, H. and Mitchell, D., A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*. 1992.

[Selman 93] Selman, B. and Kautz, H., Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. 1993.

[Smith 87] Smith, S., A Constraint-Based Framework for Reactive Management of Factory Schedules. *Proceedings of the International Conference on the Expert Systems and the Leading Edge in Production Planning and Control*. 1987.

[Smith 93] Smith, S. and Cheng, C., Slack-based Heuristics for Constraint Satisfaction Scheduling. *Proceedings of the Eleventh National Conference on Artificial Intelligence*. 1993.

[Stallman 77] Stallman, R. and Sussman, G., Forward Reasoning and Dependency Directed Backtracking. *Artificial Intelligence* **9** 135-196, 1977.

[Stroustrup 91] Stroustrup, B., *The C++ Programming Language*. Addison Wesley, 1991.

[Van 84] Van Hentenryck, P., *Constraint satisfaction in Logic Programming*. Addison-Wesley, 1984.

[Waltz 75] Waltz, D., Understanding Line Drawings of Scenes with Shadows. P. Winston (editor), *The Psychology of Computer Vision*McGraw-Hill, 1975.

[Williams 86] Williams, B., Doing Time: Putting Qualitative Reasoning on Firmer Ground. *Proceedings of the Fourth National Conference on Artificial Intelligence*. 1986.

[Xiong 92] Xiong, Y., Sadeh, N. and Sycara, K., Intelligent Backtracking Techniques for Job Shop Scheduling. *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning*. 1986.

[Zweben 89] Zweben, M. and Eskey, M., Constraint Satisfaction with Delayed Evaluation. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. 1989.

[Zweben 92] Zweben, M., Davis, E., Daun, B. and Deale, M., *Rescheduling with Iterative Repair*. Technical Report FIA-92-15, Artificial Intelligence Research Branch, NASA Ames Research Center, 1992.

[Zweben 92a] Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M. and Eskey, M., Learning to improve constraint-based scheduling. *Artificial Intelligence* **58** 271-296, 1992.

[Zweben 94] Zweben, M., Davis, E., Daun, B. and Deale, M., Iterative Repair for Scheduling and Rescheduling. *IEEE Transactions on Systems, Man, and Cybernetics.* 1994. To Appear.

# Appendix

## A.1 ODO Implementation Details

ODO is implemented in C++. Its Gantt charts are displayed using the X Library. Parsing of the input commands is performed using the LEX and BISON (a YACC-compatible parser) utilities. ODO has been compiled and tested in Gnu C++ in the UNIX environment on a Digital DECstation. Execution times are reported for a DECstation 5000/240.

ODO currently consists of approximately 10,000 lines of source code. This code compiles into an executable of approximately 2.5 MB.

## A.2 Complete ODO Grammar

ODO's complete grammar for problem declaration and problem solving is listed below. Keywords and literal text are written in **`courier bold`** font, identifiers and integers are written in normal font, nonterminals are written in *italic*, and optional arguments are placed within square brackets ([ ] - one optional argument; [ ]* - zero or more optional arguments; [ ]+ - one or more optional arguments).

```
commands                    [command ; ]*

command                       problem_declaration
                          |   problem_solver_declaration
                          |   problem_solver_action

problem_declaration           task task_string [parent_string]
                          |   duration task_string duration_integer
                          |   resource_class resource_class_string
                              [parent_resource_class_string]
                          |   resource_pool resource_pool_string
                              initial_amount_integer [resource_class_string]+
                          |   use_resource task_string resource_class_string
                              amount_integer
```

```
                                |  consume_resource task_string resource_class_string
                                   amount_integer change_start
                                |  produce_resource task_string resource_pool_string
                                   amount_integer change_start
                                |  before task_string task_string
                                |  earliest_start task_string earliest_start_integer
                                |  latest_end task_string latest_end_integer

problem_solver_declaration         var_selection [var_selection_option]+
                                |  backward_var_selection [var_selection_option]+
                                |  val_generation val_generation_option
                                |  scoring_function scoring_function
                                |  selection_function [selection_function]+
                                |  propagation_method [propagation_method_option]+
                                |  accept_criteria accept_criteria_option
                                |  cost_function cost_function_option
                                |  search_termination_criteria expression

problem_solver_action              construct
                                |  repair
                                |  randomize [seed_integer]

change_start                       st | et

var_selection_option               all | random | arbitrary | violated | orr |
                                   most_recent_failure |
                                   smallest_pv_cardinality | unassigned |
                                   all_predecessors_assigned |
                                   all_successors_assigned | none

val_generation_option              all_pv | all_but_current_pv | arbitrary_pv |
                                   random_pv

selection_function                 min_score | max_score | earliest_value |
                                   latest_value | earliest_latest_value | random
                                   | arbitrary

propagation_method_option          none | temporal_pv_unassigned |
                                   full_temporal_pv_unassigned |
                                   temporal_v_assigned |
                                   unit_resource_pv_unassigned |
                                   binary_resource_pv_unassigned
```

| | | |
|---|---|---|
| *scoring_function* | | **`none`** \| **`cost_lookahead`** \| **`fss`** |
| *accept_criteria_option* | | **`always`** \| **`cost_leq`** \| **`no_empty_pv`** |
| *cost_function_option* | | **`num_vars_unassigned`** \| **`num_violated_constraints`** |
| *expression* | | *expression binary_operator expression* |
| | \| | *unary_operator expression* |
| | \| | integer |
| | \| | *problem_state_measurement* |
| | \| | **`(`** *expression* **`)`** |
| *binary_operator* | | **`==`** \| **`!=`** \| **`<`** \| **`>`** \| **`<=`** \| **`>=`** |
| *unary_operator* | | **`!`** \| **`-`** |
| *problem_state_measurement* | | **`cost`** \| **`search_time`** \| **`search_iterations`** \| **`num_backtracks`** \| **`exhaust_search`** |

Table 8 outlines legal operators for the while-loop conditional expressions. Operators in the same box have the same precedence. Operators in a given box have higher precedence than operators in lower boxes in the table. All operators are left-associative except ! and -, which are right-associative. The symbols, precedence, and associativity are identical to that found in C and C++. Precedence can be overridden with the use of parentheses.

Table 8. Conditional-expression operators and precedence.

| Operator | Usage | Description |
|---|---|---|
| ! | ! *expr* | unary logical negation |
| - | - *expr* | unary arithmetical negation |
| < | *expr* < *expr* | binary less-than |
| > | *expr* > *expr* | binary greater-than |
| <= | *expr* <= *expr* | binary less-than-or-equal |
| >= | *expr* >= *expr* | binary greater-than-or-equal |
| == | *expr* == *expr* | binary equal |
| != | *expr* != *expr* | binary not-equal |
| && | *expr* && *expr* | binary logical and |
| \|\| | *expr* \|\| *expr* | binary logical or |