Bayesian Problem-Solving Applied to Scheduling

by

Othar Hansson

A. B. (Columbia University) 1986
M.S. (University of California at Los Angeles) 1988

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Stuart J. Russell, Chair
Professor Steven R. McCanne
Professor Kenneth Y. Goldberg

Fall 1998

The dissertation of Othar Hansson is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 1998

Bayesian Problem-Solving Applied to Scheduling

by

Othar Hansson

Abstract

Bayesian Problem-Solving Applied to Scheduling

by

Othar Hansson

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Stuart J. Russell, Chair

This dissertation describes several advances to the theory and practice of artificial intelligence scheduling and constraint-satisfaction techniques. I have developed and implemented these techniques during the construction of DTS, the Decision-Theoretic Scheduler, and its successor, SchedKit, a toolkit of scheduling algorithms and data structures.

The dissertation describes and analyzes the three orthogonal approaches to improving a scheduler’s performance. These are: (1) reducing the size of the state space to be searched, (2) reducing the per-state cost of state generation and evaluation, and (3) reducing the number of states examined by selective search.

To reduce the size of the state space, I have developed several new preprocessing algorithms designed to exploit resource constraints, including resource capacity and resource/ task compatibility. Experiments show that it is possible to exploit resource capacity constraints efficiently despite their inherently disjunctive nature.

To reduce the cost of state generation, I employ computational geometry data structures that optimize incremental heuristic evaluation, constraint-checking and state-variable maintenance. These data structures can be compiled from a formal attribute grammar specification of the heuristics and constraints. Experience with these techniques in DTS shows significant speedups and other advantages over manually-coded software.

Finally, to reduce the number of states examined during search, I have applied the Bayesian Problem-Solving (BPS) approach to the problem of search ordering in backtracking algorithms. The approach estimates, for each subtree, the search cost and probability that a solution exists. These estimates are conditioned on raw heuristic features used by other ordering techniques from the literature. Experiments with the BPS ordering heuristic on a state-of-the-art propositional satisfiability solver show that it overcomes a performance anomaly of an existing strong heuristic on two sets of benchmark problems.

Professor Stuart J. Russell
Dissertation Committee Chair

Dedication

I dedicate this dissertation to my wife I-Chun Lin and my parents and brothers and nieces and grandparents and aunts and in-laws. Their many examples of tenacity, achievement, generosity and love continue to be the greatest inspiration for me.

Acknowledgements

As Gene Lawler said of himself, “I entered graduate school with little purpose and much innocence.” I must acknowledge the great number of people who have rectified that situation for me, and thank most of them.

I am indebted to a number of excellent academic mentors. I met Rich Korf in my first day at Columbia, and he, along with Zvi Galil and Moti Yung, ensured that Andy Mayer and I could pursue our research in excellent facilities while still undergraduates. At UCLA, Milos Ercegovac, Eli Gafni and Judea Pearl (along with USC’s Ward Edwards) stuck their necks out for me during my difficult final year as I arranged my move to Berkeley.

Stuart Russell gave me the miraculous opportunity to transfer to Berkeley, and has treated me as a colleague since our first meeting. Although we have not always agreed on everything, I thank him for the considerable time he has invested in me. The other members of my examining and dissertation committees have also shown me great grace and patience.

I have learned a lot from many excellent fellow students and colleagues at Berkeley, HRI and Thinkbank: I particularly thank Jordan Hayes, Kirsten Neilsen and Chuck Ocheret, who have supported me through good times, bad times and truly weird times. My many NASA collaborators were the first to give me access to real-world problems: I must thank John Bresina, Wray Buntine, Peter Cheeseman, Mark Drummond, Peter Friedland, Roger Malina, Steve Minton, Nicola Muscettola, Eric Olson, Barney Pell, Keith Swanson and Monte Zweben for their help and support.

Finally, I thank all of the great teachers and great students I have had over the years.

CHAPTER 1. .... **Introduction** 1

1.1. Contributions and Structure of the Dissertation 1
1.1.1. A Note on the Title 4
1.2. The Scheduling Problem 5
1.3. Constraint-Satisfaction 8
1.3.1. Continuous CSPs 9
1.3.2. Discrete CSPs 9
1.3.3. Examples of CSPs 12
1.3.4. Complexity of CSPs and Scheduling Problems 12
1.4. Constrained Optimization 13
1.5. Representing Scheduling Problems 17
1.5.1. Temporal Constraints 19
1.5.2. Resource Constraints in Scheduling 19
1.5.3. Oversubscribed Scheduling Problems 20
1.5.4. Task-Introduction Scheduling 21
1.6. Components of a Scheduling Search Algorithm 22
1.6.1. Preprocessing Algorithm 22
1.6.2. Search Algorithm 22
1.6.3. State Generator 24
1.7. Summary 25

CHAPTER 2.Preprocessing Tractable Conjunctions of Disjunctive Constraints 27

2.1. Structure of the Chapter 29
2.2. Background: Shortest Paths and Infinite Capacity 29
2.3. Incorporating Resource Capacity in Preprocessing: MPC1 ...31
2.4. Aggregating Tasks: MPC2 38
2.5. MPC Extensions 47
2.5.1. Handling other Temporal Constraints 47
2.5.2. Handling Multiple Resources 47
2.5.3. Handling Attribute-Based Resource Selection 49
2.6. Evaluation 50
2.7. Related Work 58
2.8. Summary 59

CHAPTER 3. .... **State Generation using Compiled Interval Data Structures** 61

3.1. Structure of the Chapter 64
3.2. Background: State Generation 64
3.3. Background: Interval Trees 68
3.4. State-Generators using Interval Trees 71
3.4.1. Representing Heuristic Functions 71
3.4.2. Representing Constraint Checks 73
3.4.3. Generalization to State Variable Maintenance 74
3.5. Background: Attribute Grammars 76
3.6. Implementation 78

3.6.1. Balanced Parse Trees 82
3.6.2. Node-Oriented Trees 82
3.6.3. Adding Persistence to the Data Structure............................83
3.7. Examples 85
3.8. Evaluation...87
3.9. Related Work..89
3.10. Summary..92

CHAPTER 4. .... **Decision-Analytic Search Ordering** 93

4.1. Structure of the Chapter...95
4.2. Background: Branch-and-Bound 95
4.3. Background: Decision Theory...99
4.4. Applying Decision Theory to Search 104
4.4.1. The Utility of Computation Time.....................................104
4.5. The Search Ordering Decision...108
4.5.1. Test-Sequencing with Deadlines 111
4.6. Evaluation...116
4.6.1. Search Ordering in POSIT 116
4.6.2. Search Ordering using BPS...117
4.6.3. Earlier Experiments..121
4.7. Related Work...123
4.8. Summary..124

CHAPTER 5. .... **Summary and Future Directions** 125

5.1. Summary..125
5.2. Future Directions 129
5.2.1. Preprocessing Resource Constraints 129
5.2.2. State-Generation and Resource Management 129
5.2.3. Decision Analytic Search Control.....................................131

References... 135

1 Introduction

Time is the greatest innovator.
---------------- Francis Bacon

1.1. Contributions and Structure of the Dissertation

Scheduling—the assignment of time and other resources to activities—is a ubiquitous and important problem, and it has been a focus of theoretical and applied work in operations research, theoretical computer science and artificial intelligence for several decades. Many advances in these fields were inspired by the challenges of representing and solving problems of planning and scheduling. Scheduling is among the top applications of AI techniques in industrial settings, but theory lags far behind practice in this important domain.

This dissertation describes several advances to the theory and practice of artificial intelligence scheduling techniques. I have developed and implemented these techniques during the construction of DTS, the Decision-Theoretic Scheduler, and its successor, SchedKit, a toolkit of scheduling algorithms and data structures. These tools have been inspired by and applied to several NASA science planning problems, including the Extreme Ultraviolet Explorer (EUVE), the Cassini interplanetary mission, and the DS1 asteroid rendezvous [23, 24, 28, 94, 95]. The technology described here is largely experimental, but is gradually being integrated into the standard releases for use by NASA scientists. In this dissertation, I have used science planning as well as classic scheduling and constraint-satisfaction problems to illustrate the techniques I describe.

Because orbiting and interplanetary telescopes are expensive, shared, highly-constrained telerobotic devices, they are excellent test-cases for scheduling and planning techniques. Man-years of effort were devoted to scheduling single days of early science missions, such as the Voyager mission (1977-present). The astronomical cost and complexity of planning such missions presents a challenging opportunity to develop automated scheduling techniques to support or supplant human expert schedulers. For interplanetary missions, automatic scheduling capabilities are even more important, as human expert schedulers cannot dynamically reschedule given the high-latency of earth/spacecraft communications.

I have structured the dissertation into three main chapters, each describing one of three orthogonal approaches to improving a scheduler's performance. These are: (1) reducing the size of the state space to be searched, (2) reducing the per-state cost of state generation, and (3) reducing the number of states examined by selective search. Other important aspects of DTS and SchedKit, such as user interface, database integration and extension language, are described elsewhere [56, 57].

These three main chapters are best understood in light of a stylized "problem-solving cost equation":

$$\frac{\text{seconds}}{\text{problem}} = \frac{\text{seconds}}{\text{state expansion}} \cdot \frac{\text{state expansions}}{\text{state space size}} \cdot \frac{\text{state space size}}{\text{problem}}$$

We can reduce problem-solving cost (“seconds/problem”) by reducing any of the three terms on the right-hand-side. The first term is “seconds/state expansion”, a measure of search cost per state expanded: this is the topic of Chapter Three. The second term is “state expansions/state space size”: minimizing this is the traditional goal of heuristic search research, studied here in Chapter Four. The third term, “state space size/problem,” is a broad characterization of the goal of problem formulation and preprocessing, studied here in Chapter Two. Finally, an important theme of this dissertation is that if we intend to “learn” or tune techniques for reducing any one of these terms, we require even greater efficiency in the other two terms in order to allow for efficient experimentation and sampling of problem-solving episodes.

Chapter Two describes my research on *reducing the size of the scheduling state space*: I have developed several new preprocessing algorithms designed to exploit resource constraints, including resource capacity and resource/task compatibility. For example, one algorithm extends the familiar critical path method to incorporate capacity constraints. Simple *gedanken* experiments show that exponential improvements in individual time bounds are possible: non-*gedanken* experiments confirm this. These algorithms are similar in spirit, and easy to implement.

Chapter Three describes my research on *reducing the cost of state generation in scheduling search*: I have introduced the use of data structures that optimize heuristic evaluation, constraint-checking and state-variable maintenance by exploiting incremental computation. The data structures derive from computational geometry research, but I also show how they may be formally specified using attribute grammars. By using this technology, we can compile data structures that are sophisticated, yet efficient and reliable. The data structures are optimized for the needs of a particular scheduling application. I describe how the approach can be used to specify and compile specialized data structures for a number of scheduling applications, and discuss how the technology might be used to design application-specific schedule editors that check constraints interactively but efficiently.

Chapter Four describes my research on *reducing the number of states examined during search*: I apply the Bayesian Problem-Solver [53, 54, 83] approach to the problem of search ordering in scheduling and constraint-satisfaction search. To evaluate BPS in this context, I focus my experiments on the Davis-Putnam-Loveland backtracking algorithm because the original algorithm has been studied so extensively and efficient realistic implementations are available for instrumentation.

Finally, Chapter Five summarizes the dissertation.

1.1.1. A Note on the Title

The title of this dissertation is unusual: it seems to describe only the results in Chapter Four. In fact, I have retained my original title to remind myself of the lesson I have learned in this research.

The initial goal for my dissertation work was to experiment with the techniques of Chapter Four in a significant real-world setting such as scheduling. But in order to do meaningful experiments, or attempt optimization in sophisticated search algorithms for complex scheduling problems, I found it necessary to take advantage of every speedup available during preprocessing or state generation. My experience suggests that sophisticated search techniques have not, to date, been fruitful in scheduling precisely because of gaps in the low-level software infrastructure that is needed to support search applications, particularly adaptive search applications. Chapters Two and Three describe advances toward a better software infrastructure.

It is heartening that the three main chapters can describe independent routes to the improvement of scheduling systems. It suggests that unique expertise in graph theory (Chapter 2), data structures (Ch. 3), computational geometry (Ch. 3), programming language semantics (Ch. 3), decision analysis (Ch. 4), etc., can be applied to the independent sources of computational cost in scheduling problems. Although I have presented initial results in this dissertation, it would be outrageous for me to claim expertise in all these fields, and I eagerly await improvements to this work by other researchers.

1.2. The Scheduling Problem

The scheduling problem—assigning resources to tasks to achieve certain goals despite a set of constraints—is of obvious practical importance and has been a focus of theoretical and applied work in operations research and artificial intelligence for several decades. Because of scheduling’s ubiquity and complexity, small improvements to the state-of-the-art are greeted with enormous interest by practitioners and theoreticians.

There are many varieties of scheduling problems, but most could fit within the following definition. A scheduling problem is specified by a set of decision variables representing the time and resource assignments of a set of tasks, together with a conjunctive logical formula on the decision variables. The conjunctive formula is interpreted as a set of constraints that must all be satisfied by any legal assignment to the decision variables. The problem is to find a schedule—an assignment to the decision variables—such that the constraints are satisfied, if such an assignment exists. Optionally, an objective function (a function of the variables in an assignment) is stated as part of the problem.

The complexity of a scheduling problem depends on the permitted constraints. If the constraint formula consists only of linear inequalities on the time assignments for tasks (precedence constraints), then the problem is just that of finding a feasible solution to a linear program, as I will discuss in Chapter Two. As we add resource constraints, the complexity of the problem increases. For example, the Minimum Job Shop Scheduling Problem is a well-studied NP-Complete problem [50]. The problem consists of a set of jobs, each consisting of an ordered set of tasks with specified durations. Each task requires the exclusive use of one processor from a set of processors. The problem is to minimize the latest completion time over the set of tasks.

I will use the EUVE Science Planning problem as an example scheduling problem in this chapter.[1] The Extreme Ultraviolet Explorer (EUVE) is a satellite launched into Earth orbit in 1992, and observes in a relatively unexplored spectral band: the extreme ultraviolet (70-760 Å).

The tasks in the EUVE scheduling problem are astronomical observations. Although an initial sky survey employed short observations, the remainder of the mission consists of

[1] This section is based on personal communication with Dave Meriwether, Eric Olson, Mary Samuel and Gary Wong of the Center for Extreme Ultraviolet Astrophysics, Berkeley, CA, beginning in 1991.

lengthy observations. In practice, however, observations are broken into approximately 30 minute chunks by a variety of unavoidable and largely unpredictable interruptions.

The resources in this scheduling problem are observational instruments. Although the EUVE has several onboard instruments, the primary concern is to schedule pointed guest observations which are restricted to the EUV spectrometer instrument. Another set of instruments, the scanning telescopes, are perpendicular to the spectrometer, and can be used simultaneously under the obvious geometric constraints.

The constraints in the problem are determined by the positions of observational targets, the position and attitude (orientation) of the Observer platform, and the positions of obstacles such as planets, the sun and atmospheric anomalies. Inter-task constraints include observation of a moving target: a constrained sequence of “pointings.”

Additional constraints ensure the safety of the spacecraft itself. Power constraints ensure that the batteries have enough power to continue safe operation. Thermal constraints ensure that sensitive instruments do not point toward the sun.

Finally, the schedule for any NASA science mission must address subjective criteria such as “science return” (percentage of time spent on observations), “fairness” (in allocating time to competing observing projects), and “priority” (of scheduled observing projects).

This description addresses only the high-level science planning problem. At a lower level of detail, the command sequencing problem involves the preparation of actual instructions to send to the spacecraft. For example, to fulfill a single science-planning task (e.g., observe the star *tau ceti*), one must “uplink” a long sequence of command blocks to the spacecraft (slew to the target, find a calibration “guide star” to lock onto the target, perform the observation). Each command block in turn includes tremendous detail (before slewing, the slew table must be filled with the precise spacecraft trajectory, etc.).

Many approaches have been proposed for solving scheduling problems. Operations Research practitioners use such tools as mathematical programming (e.g., mixed-integer programming) or heuristic dispatch (e.g., greedy rules applied to construct a schedule online). In some cases, even the performance of the heuristic approaches can be characterized exactly on restricted problem types (see the extensive survey by Lawler *et al.* [77]).

The study of scheduling, in academia and industry, includes a number of other important topics which I do not address in this dissertation. These range from the design of the operations environment to support efficient scheduling (e.g., reduction of setup times in auto plants [19]), to the management of personnel and supply-chains [51, 135].

1.3. Constraint-Satisfaction

Although other approaches are used, scheduling problems are often posed in the language of *constraint satisfaction* so as to apply the wealth of research in artificial intelligence on general-purpose constraint-satisfaction algorithms. Practitioners typically customize these algorithms by introducing domain-specific heuristics that help guide the search for efficient and parsimonious schedules. Constraint satisfaction techniques are explained in detail in textbooks by Marriott and Stuckey [82], Poole *et al*. [104] and Tsang [125].

Formally, a constraint-satisfaction problem (CSP) consists of a set of variables together with a set of constraints on the legal values of those variables. The CSP is solved when the variables have been instantiated to a set of values that violate none of the constraints. If all variables are discrete, most of the common algorithms can generate all solutions if desired, but this can require exponential time as there may be exponentially many solutions. A wide variety of problems can be phrased as CSPs, including scheduling, graph-coloring, diagnosis, circuit verification, interpretation of visual scenes, etc. (Van Hentenryck [126] provides an interesting survey of applications.)

1.3.1. Continuous CSPs

The continuous CSP, with linear constraints, is the simplest to describe. It is equivalent to a linear program without an objective function. A continuous CSP consists of a vector of variables $\boldsymbol{V} = [V_1, V_2, \ldots, V_n]$, together with a set of linear constraints. The constraints can be specified as linear inequalities of the form: $b_i \le \boldsymbol{V} \cdot \boldsymbol{C}_i$. A consistent set of constraints specifies a convex region. Any point in this region is a solution to the CSP.

1.3.2. Discrete CSPs

Constraint satisfaction are more commonly applied to problems involving discrete variables. Formally, a discrete CSP consists of a triple:

$$\begin{pmatrix} V = \{V_1, V_2, \ldots, V_n\} \\ D = \{D_1, D_2, \ldots, D_n\} \\ C \subseteq \{D_1 \times D_2 \times \ldots \times D_n\} \end{pmatrix}$$

where V is a set of variables, D is a set of domains, one for each variable, and the relation C is a subset of the Cartesian product of the domains, specifying legal combinations of variables. For simplicity, assume that all the domains are equal, assume that all domains consist of the integers 1 through *k*, and assume that $k > 1$, unless otherwise specified.

The standard problem is to find a tuple in $D_1 \times D_2 \times \ldots \times D_n$ that is in *C*. The enumeration problem is to find all such tuples. The decision problem is to determine if there exists such a tuple: it is only for problems with special structure (such as 4-coloring a planar graph) that the decision problem may be solved without exhibiting such a tuple. An optimization problem can be posed by adding an objective function over tuples in $D_1 \times D_2 \times \ldots \times D_n$. (The standard satisfaction problem is a special case of optimization, where the objective function is known to be uniform over legal tuples.)

Typically, *C* is given more compactly, as a set of logical statements that must be satisfied by a "legal" variable assignment. The well-known Propositional Satisfiability Problem (SAT) [43, 50, 67] is an example: constraints on a set of binary variables are given as a boolean formula over the variables, using connectives "not", "and" and "or." The decision problem is to determine if the formula is satisfiable: is there an assignment of values so that the formula evaluates to true? If we convert the formula into conjunctive normal form (a conjunction of a set of clauses, each of which is a disjunction over variables and their negations), then the clauses are precisely the constraints in a CSP: each of the clauses (constraints) must be satisfied by the variable assignment.

To model structure among the constraints, the standard view of constraint satisfaction in artificial intelligence uses a graph-theoretic model, developed by Montanari [89], of a network of binary relations (arcs) among pairs of discrete variables (nodes). The conjunction of the constraints on each arc equals the relation *C*. The generalization from binary to higher-order constraints is in most respects straightforward.

A simple problem can be used to explain the terminology. A set of five variables (V_1, V_2, …, V_5) take numeric values, with V_1 and V_2 having the domain {1, 2, 3}, and V_3, V_4 and V_5 having the domain {1, 2}. The constraint graph in Figure 1-1 illustrates the constraints of the problem. A solution to the problem is, for example, the assignment {V_1 = 1, V_2 = 1, V_3 = 3, V_4 = 1, V_5 = 2}, that satisfies all of the inequality constraints.

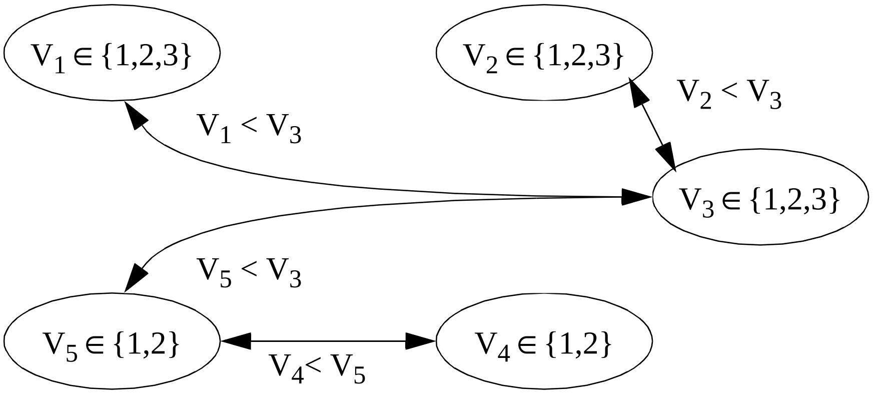

Figure 1-1. Sample CSP Problem.

A more familiar problem is the classic Eight Queens problem—placing eight queens on a standard chessboard such that no two of them “attack” each other by the rules of chess (i.e., each row, column or diagonal on the chessboard contains at most one queen). Since it is obvious that there will be exactly one queen in each row, we may number the queens V_1, V_2, ..., V_8, and let D_i = {1, 2, ..., 8} be the possible column numbers that we can assign to each queen. In this problem, the constraint graph is completely connected.

1.3.3. Examples of CSPs

CSP algorithms can be applied to interpretation or diagnostic problems as well as optimization (allocation and assignment) problems. A textbook example of an interpretation problem is to assign labels (values) such as "concave," "convex" and "shadow" to the edges (variables) in a polyhedral scene. The constraints on these labels are imposed by solid geometry and lighting [131].

Rather than expressing knowledge which can be used to constrain interpretations, the constraints found in optimization problems are typically capacity constraints or deadline requirements. A good didactic example is the Eight Rooks problem, in which eight rooks must be placed on a incomplete chessboard (with missing squares) so that no two occupy the same row or column. This corresponds to a straightforward scheduling problem, in which tasks (rooks) must be assigned to resources (columns), so that they can be completed within a deadline (the width of each row). This problem illustrates resource contention but not precedence constraints: Fox and Sadeh [47] have discussed extensions of the Eight Rooks problem that correspond to more sophisticated scheduling problems.

1.3.4. Complexity of CSPs and Scheduling Problems

Even the best algorithms for solving many CSP and scheduling problem classes will require exponential computation time in the worst case. Consider the special case of propositional satisfiability (SAT). SAT is in the class of problems NP: it can be solved in polynomial time by a nondeterministic algorithm [50]. In the case of SAT, such an algorithm would nondeterministically guess a truth assignment for the variables, and then check whether the assignment satisifes all the clauses.

SAT is also said to be NP-Complete, because every other problem in NP can be reduced to SAT in polynomial time [50]. Thus, if a polynomial-time algorithm exists for SAT, we can use it on any problem in NP by first reducing problem instances to SAT problem instances (in polynomial time) and then applying the hypothetical polynomial-time SAT algorithm. On the other hand, if any problem in NP can be proved to be intractable (no polynomial-time algorithm exists for it), then SAT would be proven to be intractable as well.

Many interesting scheduling problems are also known to be NP-Complete: Garey and Johnson [50] list over 20 NP-Complete scheduling problems, including the Minimum Job Shop Scheduling Problem discussed previously. The fact that these problems are NP-Complete strongly suggests that there is no worst-case polynomial-time algorithm for solving them, and that exponential-time search procedures are our best hope. But heuristic search techniques are often quite effective at reducing the cost of these searches, particularly when average-case performance is taken into account. This dissertation is concerned with techniques for reducing the cost of heuristic search procedures for constraint-satisfaction and scheduling problems.

1.4. Constrained Optimization

Constraint satisfaction problems are hard because the proportion of assignments that satisfy the constraints shrinks rapidly as the problem size (e.g., the number of tasks to be scheduled) is increased. The difficulty of solving such problems by heuristic search forces some practitioners to be content with finding *any* satisfying solution.

Constrained optimization refers to the search for that satisfying assignment of the variables which maximizes an objective. A widely-known optimization technique is branch-and-bound: after each solution is found in a branching search tree, add a constraint requiring subsequent solutions to have a higher objective function value, and continue searching until no better solution is possible.

The constrained optimization problem is at least as hard as the constraint satisfaction problem, of course. The number of satisfying assignments can be exponential in the problem size, making an exhaustive search for the best solution expensive. In addition, the constrained optimization problem *requires* the specification of an objective function, which can ignite political disputes over priorities, costs and risks. For example, the objective function for a NASA science mission requires negotiation between project scientists (who favor science return) and spacecraft controllers (who favor spacecraft safety).

Many scheduling practitioners protest that optimization is futile for one of three reasons:

1• The cost of finding the optimal solution exceeds its advantages over the first satisfying solution found.

2• A precise objective function cannot be formulated for the problem.

3• Multiple, conflicting objectives must be considered, and thus a single objective function cannot be formulated.

These attitudes can be heard at any gathering of scheduling researchers, particularly those working on real problems in industrial or governmental settings. Yet the sentiment is over fifty years old, dating back to the birth of mathematical programming after World War II. As George Dantzig wrote in a retrospective [35]:

> *Initially there was no objective function;* broad goals were never stated explicitly in those days because practical planners simply had no way to implement such a concept. Noncomputability was the chief reason, I believe, for the total lack of interest in optimization prior to 1947.

Fundamentally, the efficiency of search determines whether optimization is feasible for a problem. I claim that recently developed techniques, including those introduced here, make it possible to strive for optimization, rather than settling for constraint satisfaction.

The second and third arguments suggest that it is impossible to represent the preferences and objectives of an organization for use in a scheduling system. But the past 25 years of research in multiattribute decision analysis have produced preference elicitation techniques for modeling conflicting objectives. The fact that many scheduling problems involve multiple parties (for example, scientists and spacecraft controllers) suggests that there *are* still fundamental theoretical difficulties (e.g., Arrow’s theorem on possible intransitivity of group preferences [8]). But multi-party decision analyses are routine, and there is no reason to believe that scheduling is impervious to such modeling.

A more common problem is political: how would one explicitly represent the objectives for scheduling the surgery department of a hospital, without rankling individual patients? Or how would one represent a corporation’s objectives in labor scheduling without upsetting workers (or violating union contracts). This argument is essentially that “some things are better left unspoken,” thus banning explicit modeling of objective functions.

But these theoretical and political difficulties should not be considered without acknowledging that objectives are already encoded, implicitly, in existing scheduling and constraint satisfaction applications. One would be hard-pressed to design a problem-solving system that did not exhibit implicit objectives.

For example, objective functions are implicitly encoded in the search strategies of many constraint satisfaction systems. Consider a search strategy that attempts to find a satisfying solution that includes all priority 1 targets, and only if necessary, one that replaces some of these with priority 2 targets. Such a system assumes a “lexicographic” objective function that would trade an arbitrary number of priority 2 targets for an additional priority 1 target. Because the objective function is encoded within the search algorithm or its heuristic functions, it is difficult to characterize, let along change, the system’s behavior [52].

Moreover, scheduling objectives are often compiled into “policy” constraints in implemented systems. In science missions, a common constraint concerns sun angle. Because of objectives of instrument health, one prefers not to point data collection instruments too close to the direction of the sun. This preference is typically encoded as a constraint on the minimum angle between the telescope boresight and the sun. The fact that it is a *preference*, not a *constraint*, is evidenced by changes in the minimum angle “constraint” over the lifetime of the mission, as mission planners relax their attitudes toward risk.

As this discussion suggests, in this dissertation (particularly in Chapter Four) I advocate the use of objective functions. But in fact many of the results are intended to satisfy the rather universal objective of minimizing computation time. Over time, improved search efficiency will make constrained optimization a feasible alternative to simple constraint satisfaction for a growing number of domains.

1.5. Representing Scheduling Problems

A large class of scheduling problems can be represented as constraint-satisfaction problems, by representing attributes of tasks and resources as variables. A schedule is represented as an assignment of values to the variables. To give the reader some intuition about CSP applications, I describe problem representation in some detail here: much of the rest of this dissertation focuses on search and reasoning rather than representation.

Task attributes include the scheduled start and end time, and a resource assignment. We represent the beginning and end of task T_y with variables BT_y and ET_y. The primary attribute of resources is availability (i.e., down-time and work schedules). A schedule is constructed by assigning times and resources to tasks, while obeying the constraints of the problem.

Constraints capture logical requirements: a typical resource can be used by only one task at a time. Constraints also express problem requirements: task T_x requires N units of time, must be completed before task T_y, and must be completed before a specified date. Van Hentenryck [126] provides concise examples of scheduling problems represented as CSPs.

Figure 1-2 depicts a portion of the constraint graph for a simple scheduling problem. Only the temporal constraints are shown: the completion of task T_1 must precede the beginning of task T_2, the completions of task T_4 and T_5 must coincide, etc. In addition, the resource requirements for T_5 are shown: that task requires seven hours to complete, and uses four resources for those seven hours (one from resource class A, and three from B).

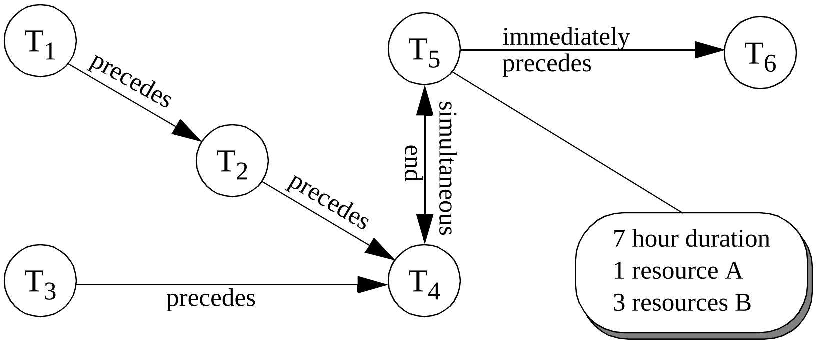

Figure 1-2. Partial Constraint Graph for a Scheduling Problem.

Representing resources explicitly is a convenient abstraction which avoids the construction of a complete constraint graph over the tasks. Such a graph would be used to prevent the simultaneous assignment of a pair of tasks to the same resource. In the explicit representation, indistinguishable resources can be grouped in pools or classes, whose attributes maintain the number of resources available and in use during an interval.

A scheduling problem may also require the representation of attributes (or state) of domain objects over time (e.g., the configuration of a production line). For example, tasks T_1 and T_5 may require a domain object to be in a particular state, while task T_2 may require a different state: in this case, T_1 and T_5 could legally be scheduled in parallel, while task T_2 may not. If the domain objects can change state without intervention by a scheduled task (e.g., a battery discharging itself), such behavior must be modeled as well [92, 93].

1.5.1. Temporal Constraints

A scheduling problem consists of two types of temporal constraints:

- *Temporal constraints between tasks or fixed time-points*. For example, Figure 1-2 shows that T_1 must precede T_2. This is equivalent to the constraint $ET_1 \leq BT_2$.
- *Task durations*. For example, Figure 1-2 shows that T_5 has duration 7. This is equivalent to the constraint $BT_5 + 7 = ET_5$. It is of course possible for a task to have an uncertain duration, specified by an interval: $BT_5 + 7 \leq ET_5 \leq BT_5 + 9$.

In some cases, it is necessary to produce specific (i.e., numerical) start- and end-times as the output of a scheduling system. I will call such a schedule a *ground* schedule, in contrast to a *least-commitment* schedule, which consists of bounds on the start- and end-times. Least-commitment schedules are more practical in many settings, as most ground schedules will be violated as soon as schedule execution begins. As a definition, I will say that a least-commitment schedule must have the property that a ground schedule can be extracted from it in polynomial time (by repeatedly fixing times and then propagating constraints).

A technique suggested by the work of Smith and Parra [122] simplifies the representation of least-commitment scheduling problems. Instead of associating two variables (start- and end-time) with each task, we can associate four variables (earliest-start, latest-start, earliest-finish and latest-finish), and rewrite the inter-task constraints in the obvious way. A precise assignment to these four variables is a set of bounds on the original two variables (start-time and end-time).

1.5.2. Resource Constraints in Scheduling

Resource constraints are the primary source of *disjunction* in scheduling problems, and through disjunction, computational complexity. In the case of resources with unit capacity (i.e., non-sharable resources), the disjunction is easily represented: if T_x and T_y require the use of resource R_z, which has unit capacity, then the executions of T_x and T_y cannot overlap. In other words, it is either the case that T_x precedes T_y, or that T_y precedes T_x.

In the job-shop scheduling literature, such constraints are often represented in a *disjunctive graph* [27]. The nodes of the disjunctive graph represent tasks. Two tasks are joined by an undirected arc if they require the use of a common resource. Any legal schedule implies an assignment of direction to the undirected arcs: these arc directions must be mutually consistent, i.e., there can be no directed cycles. The direction imposed on the arc between two tasks represents the tasks' ordering in the schedule.

1.5.3. Oversubscribed Scheduling Problems

It is also useful to represent problems which do not, or cannot, require solutions in which every task is scheduled. Such oversubscription problems are the norm in science planning.

To represent this problem, we introduce a binary variable ST_y for each task T_y. ST_y is a binary variable indicating whether T_y is scheduled. Every constraint $C(T_y)$ on T_y can then be rewritten as the logical sentence $ST_y \Rightarrow C(T_y)$: the constraint $C(T_y)$ is enforced if T_y is scheduled.

This technique is related to Lagrangian relaxation, where constraints move into the utility function. A natural utility function for oversubscribed problems would be to maximize the weighted sum of the ST_y variables (interpreting the binary variable as a 0-1 variable).

1.5.4. Task-Introduction Scheduling

Traditionally, scheduling is distinguished from planning by a focus on reasoning about a an externally generated *fixed set* of tasks; planning is concerned with the *generation* of a partially-order set of tasks to achieve some goal. Fox [46] distinguishes them as follows:

> Planning selects and sequences activities such that they achieve one or more goals and satisfy a set of domain constraints.
>
> Scheduling selects among alternative plans and assigns resources and times for each activity so that the assignments obey the temporal restrictions of activities and the capacity limitations of a set of shared resources.

Thus a planning system might feed into a scheduling system.

However, many scheduling applications require the fairly simple selection of additional activities that must be introduced to meet schedule constraints. I call this *task-introduction scheduling*. Task-introduction scheduling has been implemented in the COLLAGE [75] and HSTS [93] systems, among others. For example, if each task requires a resource to be in a particular state, then a state-change task (i.e., a reconfiguration) might need to be added to the schedule.

In the course of my research, I have developed several simple mechanisms that support task-introduction scheduling. These techniques are described in Chapter 3.

1.6. Components of a Scheduling Search Algorithm

The three main chapters of this dissertation address three central components of a scheduling system. To help the reader, this section describes how those components interact.

1.6.1. Preprocessing Algorithm

Preprocessing is the computation of implicit constraints prior to search. For example, in Figure 1-1, the two explicit “less than” constraints $V_4 < V_5$ and $V_5 < V_3$ can be composed to form the implicit constraint $V_4 < V_3$. This is simply logical inference on the constraints, together with the transitive semantics of the “<” relation. Preprocessing techniques are valuable because they reduce the size of the search space: even a brute-force search can outperform an intelligent “selective” search if preprocessing has sufficiently reduced the search space.

The DTS and SchedKit systems incorporate several novel preprocessors that combine resource and temporal constraints. The underlying approach is described and evaluated in Chapter 2.

1.6.2. Search Algorithm

After preprocessing, a search phase is typically required to find a solution to the CSP. There are essentially two types of search algorithms: backtracking and repair-based.

Backtracking search algorithms partition the space of possible schedules and systematically search through the hierarchy of partitions for a schedule. We can distinguish between simple backtracking and constraint-posting search algorithms. A simple backtracking algorithm assigns a variable to a single value at each step of the search (e.g., assigning the start time of a task to 8AM on a particular day). A constraint-posting search algorithm instead has the option of posting a constraint on the variable's value (e.g., assigning the start time to a particular week, or constraining the task to precede some other task). Much recent research has focused on the efficiency of simple backtracking and constraint-posting search, particularly in the AI planning community, where these techniques correspond to total-order and partial-order planning [10, 85]. Because of the large granularity of its search space, a constraint-posting algorithm is able to prune large portions of the search tree (e.g., by finding a dead-end as a result of constraining the order of two tasks).

Repair-based search combines the explicitness of simple backtracking with some of the flexibility of constraint posting. A typical repair-based search algorithm begins with a randomly generated initial schedule. If no constraints are violated, the algorithm terminates. Otherwise, a "repair" is made by reassigning a variable (e.g., changing the start-time for a task), and the algorithm repeats this step until a solution is found, or until a timeout is exceeded and the algorithm restarts from a new initial schedule. Repair-based search has been the focus of much scheduling research in recent years [4, 86, 137].

Because they are systematic, backtracking schedulers can determine if a solution does not exist (given enough time). On the other hand, repair-based schedulers, because they are stochastic, can often provide a shorter expected running time per problem instance. This is because they both randomize the input (by choosing an initial schedule) and randomize their decisions. This combines the advantages of the Sherwood algorithms and Las Vegas algorithms studied in theoretical computer science [26].

Chapter 4 of this dissertation describes a decision-analytic approach to controlling the ordering of search in a backtracking algorithm.

1.6.3. State Generator

An important practical consideration in designing search algorithms is the speed with which individual states are “generated.” When a state is generated from its parent, the effects of changes must be propagated so that (1) constraint violations can be detected, and (2) heuristic evaluations can be made on the resulting state. Slow constraint propagation and/or heuristic evaluation can make even the most intelligent search algorithm impractical, even though state generation is rarely more than polynomial time in complexity. Many well-known AI scheduling systems take several seconds to propagate the effects of a change to the schedule. This restricts these systems to performing only token amounts of search, for example, by relying on dispatch rules [16, 20].

In retrospect, efficient state generation has been crucial to the performance of most practical applications of heuristic search. Optimizations of the state generator and heuristic evaluator play an enormous role in applications as widely varying as chess [81], mathematical programming [133, 134] and speech understanding [78].

This dissertation describes a very efficient state generator and heuristic evaluator, relying on data structures that I have adapted from computational geometry applications. I have also designed a “compiler” that can produce similar data structures for other scheduling applications. These techniques are described in Chapter 3.

1.7. Summary

As this discussion of search components suggests, there are several ways to improve the effectiveness of scheduling search algorithms: better *preprocessing*, better *state-generation* and better *search control* are the three techniques I have focused on in the three main chapters of this dissertation.

2 Preprocessing Tractable Conjunctions of Disjunctive Constraints

The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.
---------------- *Igor Stravinsky*

Preprocessing is the opening gambit in conquering a constraint-satisfaction problem. An investment in preprocessing time may reveal valuable implicit constraints that can be deduced from the problem statement. Preprocessing techniques are typically of much lower time complexity (polynomial time) than the subsequent constraint satisfaction search algorithms (exponential time), but in extreme cases, strong preprocessing techniques can make search unnecessary or extraordinarily efficient. More typically, preprocessing serves to reduce the size of the state space that must be searched to find a legal schedule, by eliminating variables, reducing the size of variable domains, or adding constraints. Finally, even in manual scheduling systems, preprocessing provides valuable guidance to the human scheduler by eliminating possibilities or detecting inconsistencies.

I have developed several new preprocessing algorithms designed to exploit *resource constraints*, including resource capacity and resource/task compatibility. This chapter describes those algorithms. One algorithm, MPC1, extends the familiar critical path method to incorporate capacity constraints. Another, MPC2, finds precedence constraints between subprojects based on their internal resource requirements. These algorithms are similar in spirit, straightforward to implement, and could be independently applied to improve many existing scheduling systems.

As I discuss at the end of the chapter, resource capacity constraints have often been viewed as a source of complexity, because they introduce disjunctive constraints. Disjunction is a classic correlate of complexity in computer science. But this chapter suggests that tractable *conjunctions* of these *disjunctive* constraints can be identified, and their joint implications used to derive tight bounds on the search space during preprocessing or search. In other words, rather than assume that disjunction leads inevitably to complexity, I have searched for, and found, methods to *group* disjunctive constraints so that useful constraints (implications) can be derived from the group at low cost. I anticipate that other, similar techniques can be developed for constraint satisfaction using this basic approach.

Because the input and output of preprocessing algorithms are of the same data type (both the input and output are sets of constraints, i.e., descriptions of constraint satisfaction problems), they can be composed or pipelined quite flexibly. In validating one of the algorithms described here, I found that it discovered new constraints that could be pipelined to increase the effectiveness of a pre-existing preprocessing algorithm.

2.1. Structure of the Chapter

The background for this chapter is work on the critical path method and temporal reasoning, as decribed in Section 2.2. In Section 2.3, I introduce the simplest version of the new technique, MPC ("Method for Preprocessing Capacity"), for handling unit-capacity resources. The remaining sections describe the handling of resource calendars, subprojects, and other complications and extensions. I conclude the chapter with an empirical validation of the effectiveness of two MPC techniques.

2.2. Background: Shortest Paths and Infinite Capacity

Following Meiri [84], we begin with the Simple Temporal Problem (STP): a constraint-satisfaction problem in which each pair of time points is constrained by at most one constraint. That constraint must specify a convex interval of the form:

$$a_{ij} \le X_j - X_i \le b_{ij} \tag{2-1}$$

where X_j and X_i are time points, and a_{ij} and b_{ij} are the minimum and maximum gap, respectively, between them. A simple constraint such as X_i *must precede* X_j is modelled by setting a_{ij} to zero, and b_{ij} to positive infinity.

A solution to the STP is easily obtained by solving the corresponding set of linear inequalities, which produces the tightest constraints between each pair of time points.

As Dechter, Meiri and Pearl [39] observed, this system of linear inequalities has a special structure: it is a set of "difference constraints," and can be solved by solution of the *all-pairs shortest-paths problem* in a "distance graph" constructed based on the constraints. This is the key insight of their DMP preprocessing algorithm for finding the tightest temporal constraints for an STP. The efficient solution of a system of difference constraints was first demonstrated by Bellman [13], and is described in many combinatorial algorithms textbooks [7, 32].

How is DMP useful in a scheduling problem? Given a problem, we will separate out the temporal difference constraints, apply the DMP preprocessing algorithm to them, and then use the resulting tighter temporal constraints when solving the original scheduling problem. The tighter temporal constraints may force some task ordering choices, or simply reduce the range of possible start times for each task.

The DMP algorithm works by constructing a "distance graph" based on the STP. If d_{ij} is the length of the shortest path from node i to node j in the graph, node 0 is a designated "origin" node, and e_{ij} is the length of the edge from node i to node j, then by the definition of a shortest path:

$$d_{0j} \le d_{0i} + e_{ij}\,. \tag{2-2}$$

This implies that $d_{oj} - d_{oi} \le e_{ij}$. By defining d_{oj}=X_j and d_{oi}=X_i and e_{ij}=b_{ij}, we can create a one-to-one mapping between edges in the graph and the original difference constraints (in this case, the original constraint $X_j - X_i \le b_{ij}$). By similar reasoning, the constraint $a_{ij} \le X_j - X_i$ can be represented by setting e_{ji}=$(- a_{ij})$. Solving the single-source (node 0) shortest-path problem in this graph results in an assignment of minimal values to the X_i variables such that the original difference-constraints are satisfied. Solving the *all-pairs* shortest-path problem can be used to find the tightest possible difference constraints between every pair of time points [39].

All-pairs shortest-paths algorithms include the familiar Floyd-Warshall algorithm. Most such algorithms are based on the use of dynamic programming to infer shortest-path lengths based on a recurrence-relation definition. The Floyd-Warshall algorithm has time complexity in O(n^3), where n is the number of time points. (The complexity of some other all-pairs shortest-paths algorithms is also a function of m, the number of constraints or edges in the original problem.) Listing 2-1 presents the Floyd-Warshall algorithm, and how it is used in the DMP preprocessing algorithm.

In many scheduling applications, we are interested only in constraints on events in relation to an *absolute* time-line, and not *relative* constraints between all pairs of events. If absolute constraints suffice, the problem of temporal preprocessing is made even simpler. The tightest absolute constraints can be derived by solving the *single-source* and *single-destination* shortest-path problems, using a “project-start” event as the source and destination. These problems are efficiently solved using Dijkstra’s algorithm. Using Fibonacci heaps [32], Dijkstra’s algorithm has time complexity in O($m + n \log n$). This is almost certainly negligible compared to the exponential cost of a search to solve the remainder of the original scheduling problem. In addition, one can solve the all-pairs shortest-path problem as n single-source shortest-path algorithms, with resulting time complexity in O($n(m + n \log n)$).

2.3. Incorporating Resource Capacity in Preprocessing: MPC1

The DMP preprocessing algorithm is a generalization of the “critical path method” (CPM) that has been used in project planning and scheduling for 40 years [71, 72]. The critical path between a pair of time points (typically the start and end of a schedule) is the length

```
void DMP(Problem p) {
    Graph g = ConstraintGraph(p);
    FloydWarshall(g);
}

void processTriangle(Node x, Node y, Node j, Graph g) {
    if arcExists(y,j,g) {
        int newLen = arcLen(x,y,g) + arcLen(y,j,g);
        if (!arcExists(x,j,g) || (newLen<arcLen(x,j,g))) {
            setLen(x,j,g,newLen);
        }
    }
}

void FloydWarshall(Graph g) {
    Node x,y,j;
    List intermed = nodes(g);
    while (y = pop(intermed)) {
        List sources = nodes(g);
        while (x = pop(sources)) {
            if arcExists(x,y,g) {
                List destinations= nodes(g);
                while (j = pop(destinations)) {
                    processTriangle(x,y,j,g);
                }
            }
        }
    }
}
```

Listing 2-1 DMP Algorithm.

(This is “pseudo” C++ code which is compilable with suitable auxiliary functions.) The DMP preprocessing algorithm takes a problem *p* as input, extracts the constraint graph *g* from it, and uses the classic Floyd-Warshall algorithm to tighten the pairwise temporal constraints represented in *g*. The processTriangle subroutine checks and updates the shortest-path information stored on *g*’s arcs. After calling DMP, subsequent preprocessing steps (or a scheduling search algorithm) will be able to use the tightened temporal constraints.

of the longest precedence-constrained sequence of tasks that must occur between the two time-points. Specifically, we can compute the critical path by constructing a graph of tasks, labeling each task node with a “cost” corresponding to its duration, and placing directed arcs between tasks to represent precedence constraints. The project-start and project-end are represented by dummy task nodes. The critical-path for the project is the highest-cost directed path through the task nodes between the project-start and project-end nodes.

A major shortcoming of both DMP and the critical path method is that they ignore capacity constraints. For example, in a complex assembly schedule, the temporal dependencies form a tree, with raw materials becoming parts, parts becoming subassemblies, etc., until a completed product has been assembled [90, p. 190]. The critical path through such a schedule may be very short, but the resource requirements can be enormous.

As a result, the temporal constraints derived by the DMP algorithm are often very poor. Some examples will show the intuition behind the first MPC algorithm, which I call MPC1. Imagine an idealized assembly schedule (Figure 2-1), in which pairs of subassemblies are repeatedly combined, with each step requiring one dedicated worker and one time unit. After 2^N-*1* individual steps, a completed product has been made out of the 2^N original parts. A critical path through this schedule is only O(N) steps long. But if we have a resource constraint that only c workers can be used in assembling the entire product, the product will not be complete until at least $(2^N\text{-}1)/c$ time units have passed. The ratio of the new bound $((2^N\text{-}1)/c)$ to the old bound O(N) shows an exponential increase in the lower-

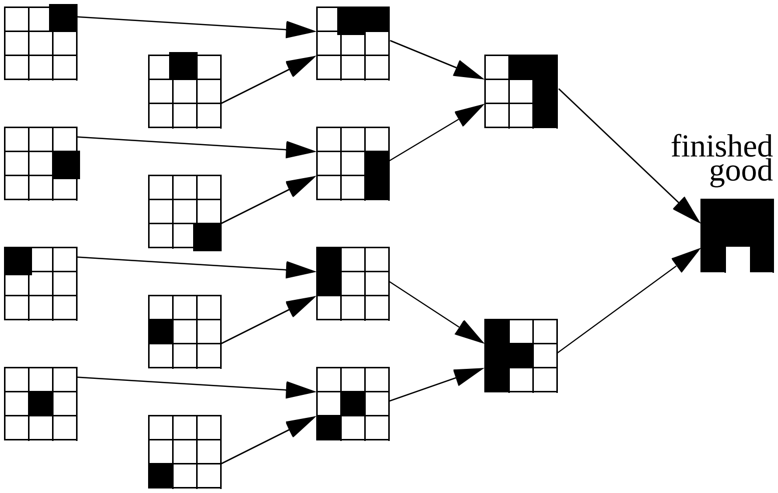

Figure 2-1. Idealized “assembly” schedule.

In this stylized assembly schedule, a finished good is assembled from 2^N parts (in this case, eight parts). In each assembly step, two subassemblies are joined. The number of assembly steps is simply the number of non-leaves, or 2^N-1.

bound on the schedule length. This more-informed bound may reduce the cost of the search for an optimal schedule.

Another case where MPC performs well is a series-parallel network, as depicted in Figure 2-2. If there is a resource constraint that only *c* workers are available, MPC will exploit this constraint to derive more-informed bounds. Here, MPC’s strong bounds can improve the feasibility of breaking scheduling problems into multiple pieces, to be solved in parallel (by human experts or machines). If we define this series-parallel network as four binary trees (each with 2^N-1 nodes), and consider the CPM and MPC bounds on the node X as *N* increases, we again see an exponential ratio: 2*N* compared to 2x2^N.

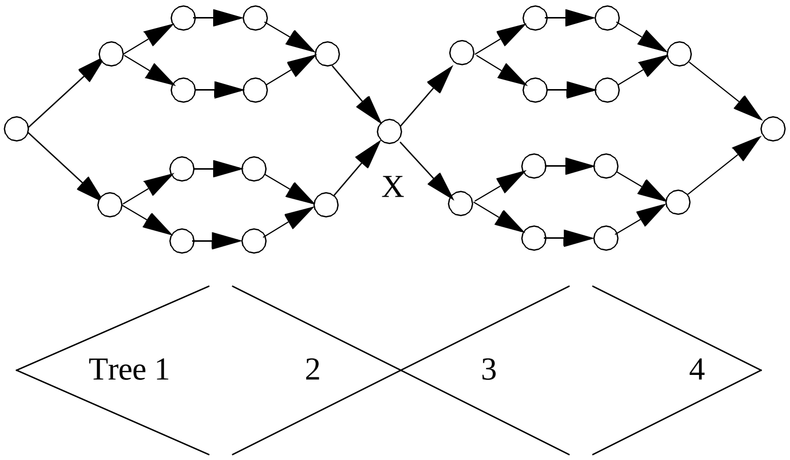

Figure 2-2. Series-Parallel precedence network.

MPC1 can arrive at improved bounds (compared to the critical path method or the DMP algorithm) for the start-time of task X, permitting this problem to be decomposed into two portions, e.g., to be solved by two human scheduling experts. In the generalization of this example, four binary trees are arranged as shown: the bounds for X are exponentially better using MPC1 rather than temporal preprocessing alone.

These examples show the intuition behind the first MPC algorithm, which I call MPC1. It considers each set of tasks that (1) occur between two temporal events, and (2) require a common resource. The summed duration of these tasks, divided by the capacity *c* for that resource, is a lower bound on the time between the two events.

In other words, we examine a set of temporal constraints and resource requirements, and derive (possibly tighter) temporal constraints. A simple alternating algorithm can be used to exploit these constraints in polynomial time:

1. Compute precedence relations using DMP preprocessor.

2. Add new constraints derived from precedence, resource sharing, and capacity.
3. Optionally repeat by returning to step 1.

Pseudocode for this algorithm is presented in Listing 2-2. This simple algorithm is trivial to implement, and as suggested above, it can yield exponentially tighter bounds on the start-times of tasks under resource constraints. This corresponds to a smaller search space for automated scheduling algorithms, as well as information on bounds that can aid a human scheduler.

In addition to alternating between the DMP and MPC1 algorithms, we can run them concurrently, with each feeding updated constraints to the other. To do this very efficiently, we require a DMP algorithm implemented using a *dynamic* all-pairs shortest path algorithm. To communicate between DMP and MPC1, we add a dynamic "MPC1" edge between every pair of nodes in the original constraint graph. We run DMP as before, but notify the MPC1 algorithm whenever a new precedence relation is discovered (there can be at most $O(N^2)$ pairs of nodes in the precedence relation, so MPC1 is invoked at most $O(N^2)$ times). MPC1 processes the precedence relations, and whenever a new MPC1 constraint is discovered, it modifies the length of the corresponding MPC1 edge. The dynamic all-pairs shortest path algorithm is responsible for incorporating this edge, and may discover new precedence relations as a result.

Later in the chapter, we present results that validate the effectiveness of MPC1 on a test suite of scheduling problems. Surprisingly, the effectiveness of MPC1 appears to be in discovering slightly tighter temporal constraints that in turn enable DMP to be even more

```
void MPC1(Problem p) {
    Graph g = ConstraintGraph(p);
    DMP(g);
    List sources = tasks(p);
    while (n1 = pop(sources)) {
        List destinations = tasks(p);
        while (n2 = pop(destinations)) {
            if (precedes(n1,n2,g)) {
                int durationSum = 0;
                List intermediate = tasks(p);
                while (n3 = pop(intermediate)) {
                    if (precedes(n1,n3,g) && precedes(n3,n2,g)) {
                        durationSum = durationSum + duration(n3,g);
                    }
                }
                int capacity = 1; // unit capacity
                updateLowerBound(n1,n2,durationSum/capacity,g);
            }
        }
    }
}
```

Listing 2-2 Single-Stage MPC1 algorithm.

The algorithm has the same loop structure as the Floyd-Warshall algorithm, but applies resource constraints in addition to temporal constraints to each <source, intermediate node, destination> triple.

effective. Repeated runs of DMP, interleaved with runs of MPC1, produce tighter and tighter constraints.

2.4. Aggregating Tasks: MPC2

We can use techniques similar to MPC1 to reason about the *aggregate* resource requirements of groups of tasks. In multi-project scheduling, such as scheduling of astronomical observations by different astronomers, largely independent subprojects compete for the same resources. If these subprojects can be interleaved arbitrarily, the search space will be very large. If, however, we can establish, during preprocessing, that two subprojects cannot overlap, there are only two possible orderings.

For example, two subprojects with N loosely constrained tasks can be interleaved in $C(2N,\ N)$ ways.[1] The number of possible orderings $C(2N,\ N)$ is in $O(N\,!)$.

Figure 2-3 illustrates a simple example with two subprojects: Light and Dark. Each subproject consists of a chain of four-day tasks, with the constraint that consecutive tasks can be separated by at most three days.

While there may be no explicit constraints between the two subprojects, it is clear after quick inspection that they cannot overlap. The following informal proof-by-contradiction shows that this intuition is true (simpler proofs are possible, but this one is similar to the proof constructed by the MPC2 algorithm).

- Assume that subprojects Light and Dark overlap for N days.
- If Light and Dark overlap at least seven days, then there is a period of seven days in which eight unit-days of resource capacity are required. This cannot happen.

[1] Pronounced “2N choose N”, $C(2N,\ N)$ corresponds to how many ways there are to place N identical pegs into 2N distinct holes [32, p. 101].

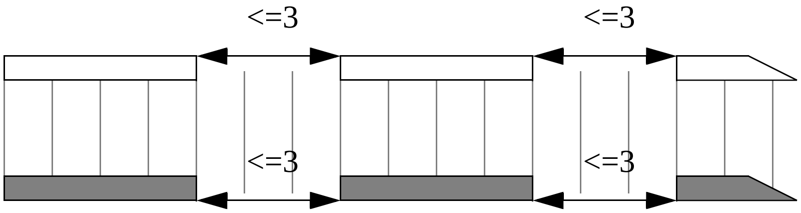

Figure 2-3. Two subprojects compete for a resource.

Subprojects Light and Dark are shown. Each is a chain of four-day tasks. Adjacent tasks may be separated by a gap of at most three days. Because Light and Dark each require the resource for at least four out of any seven days, they cannot overlap by seven days or more.

- If, on the other hand, the subprojects overlap from one to six days, the first task of the latter project and the last task of the earlier project will require two units of capacity on at least one day. In either case, the subprojects exceed the resource capacity if they overlap. This cannot happen.
- Thus, they cannot overlap.

How can we formalize and automate this reasoning? The basic conceptual tools I have developed are three kinds of *resource usage profiles*: the start-profile, the end-profile and the intermediate-profile.

The start-profile is rs(t_α,σ), a lower-bound on cumulative resource-usage as a function of t_α, time since the start of the subproject σ. The end-profile is re(t_ω,σ), a lower-bound on cumulative *remaining* resource usage as a function of t_ω, time *remaining* on the subproject σ. The intermediate-profile is ri(δ,σ), a lower-bound on cumulative resource usage over *any* interval [t...t+δ] during which the subproject is executing (i.e., such that t is not before the start time of the subproject, and t+δ is not after the end time of the subproject). These profiles are illustrated in Figure 2-4.

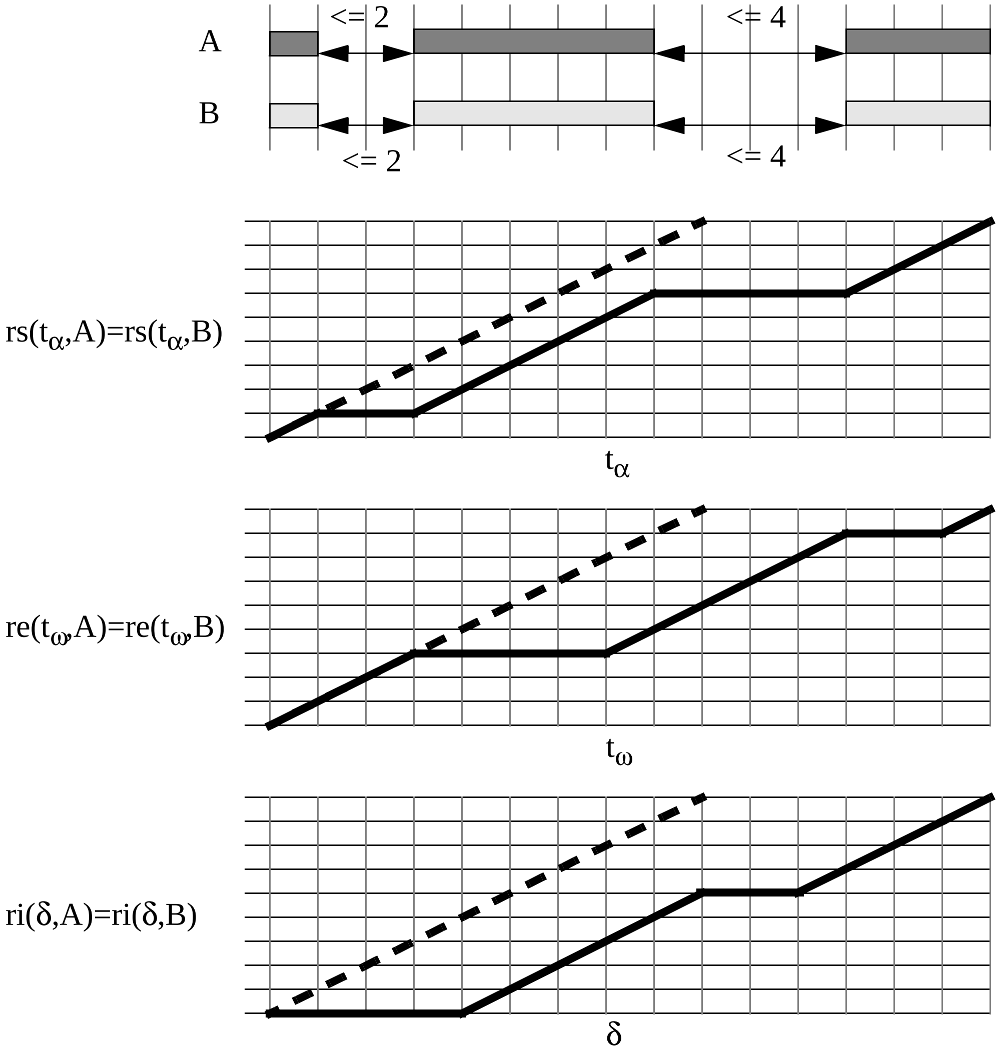

Figure 2-4. Resource Usage Profiles for Isomorphic Subprojects A and B.

Two subprojects are shown at the top of the figure. In subsequent figures, I examine whether they can overlap. The three graphs are (top to bottom), the start-profile, end-profile and intermediate-profile for the two subprojects. Because the two subprojects are identical, their resource profiles are identical. The dashed line in each profile is a reference point, corresponding to constant resource usage. Note that t_a *measures time from the beginning of the subproject, while* t_w *measures time from the end of the subproject.*

MPC2 is based on the fact that two subprojects A and B cannot overlap (with A ending before B) if the following conditions hold for any value δ, and the subprojects require the same unit-capacity resource:

1. $rs(t_\alpha=\delta,B)+ri(\delta,A)>\delta$
2. $\forall\tau,\ 0\leq\tau\leq\delta\ [rs(t_\alpha=\tau,B)+re(t_\omega=\tau,A) > \tau]$

The first condition is that the overlap of the *first* δ time-units of subproject B with *any interval* of the same length in subproject A will require a total resource capacity greater than the δ available. Thus any legal overlap must be less than δ time-units. In our algorithm, the first step is to find the minimum value δ for which this condition holds.

The second condition is that any overlap of *less than* δ time-units requires more resource capacity than is available (by considering resource usage for the *first* τ time-units of subproject B and the *last* τ time-units of A). An example of these calculations is illustrated in Figure 2-5 and Figure 2-6.

Because they are lower-bounds, there are many ways to compute the resource-profiles, and a tradeoff exists between tight lower-bounds, and lower-bounds that can be computed and manipulated efficiently. I have used algorithms based on Piecewise Linear Trees, a simple adaptation of the Interval Tree data structure described in Chapter Three. Briefly, a Piecewise Linear Tree is the integral of a corresponding interval tree: i.e., it computes the cumulative height (as *x* increases) of a set of intervals over the *x* axis. An interval can be inserted in the tree in logarithmic time.

Testing the first condition requires the start-profile and end-profile only. The start-profile is easily constructed as a PLT by inserting, for each task, an interval of resource usage

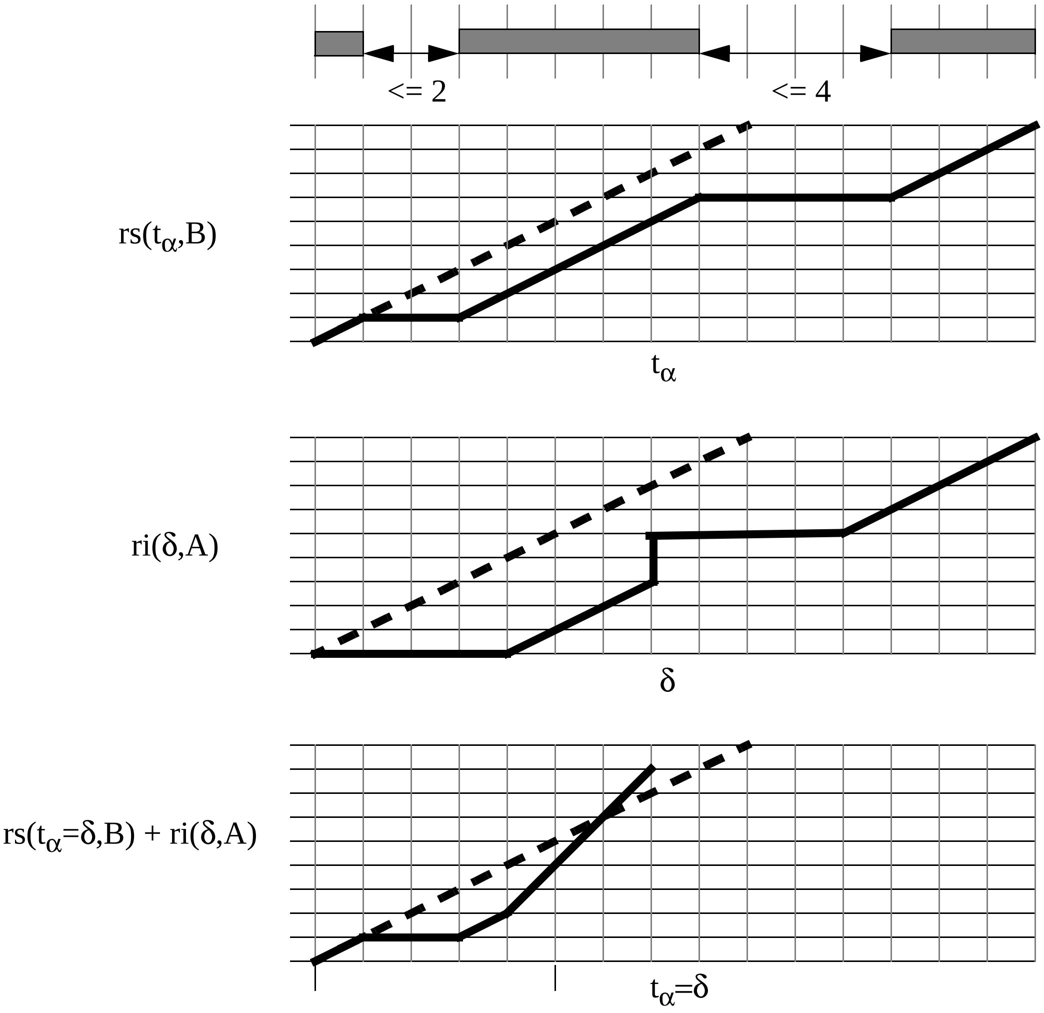

Figure 2-5. Calculation of Upper-Bound Permitted Overlap between A and B.

This figure explores the question: if Subproject A begins before B, but they overlap by at least δ *time units, will they exceed resource capacity? This can be answered for each value of* δ*, by adding rs(t_a=δ,B) to ri(δ,A). The greatest value of* δ *for which rs(t_α=δ,B)+ri(δ,A) does not exceed* δ *(for unit capacity resources) is an upper-bound on the possible overlap between A and B. In this case, the upper-bound is* δ≤6*.*

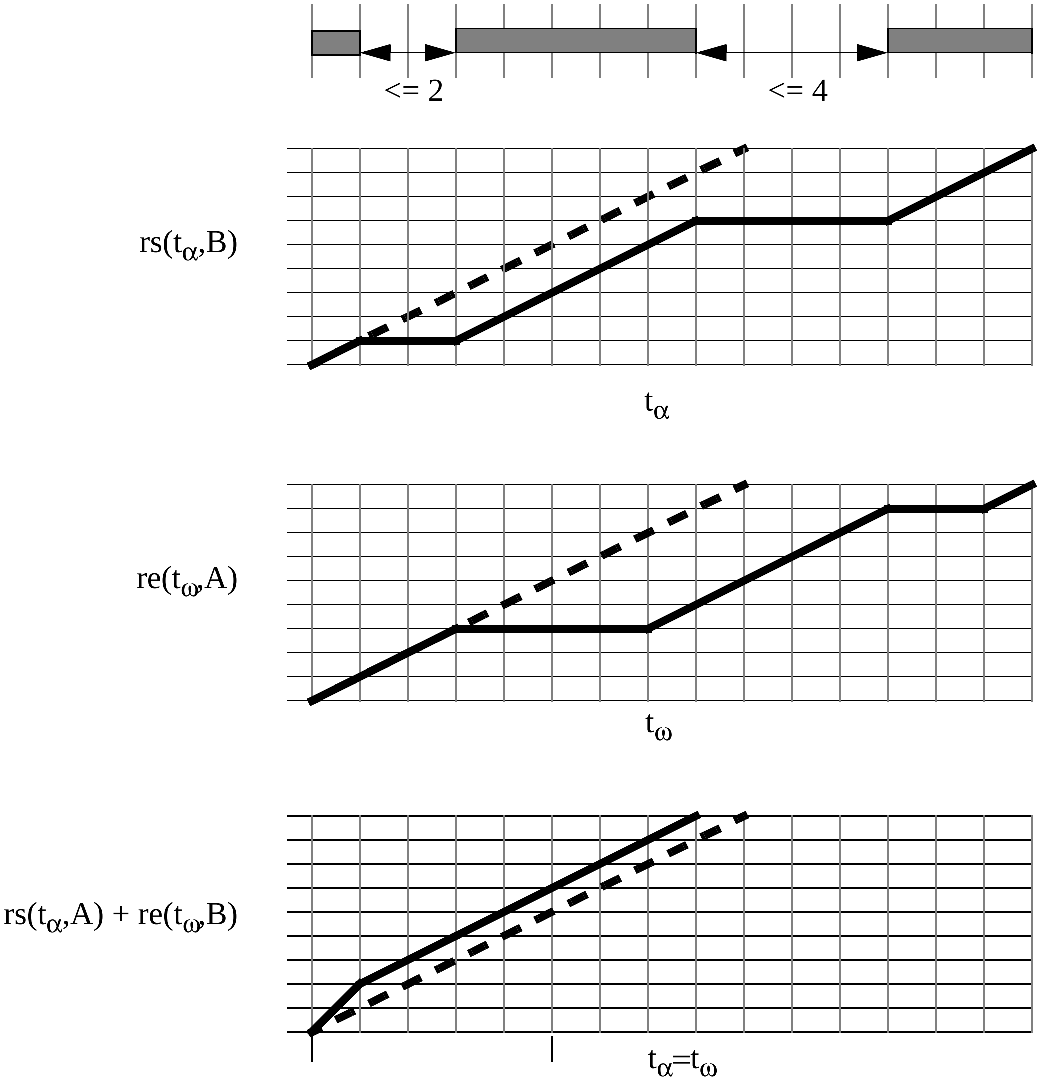

Figure 2-6. Exact Check of Permitted Overlap between A and B.

This figure explores a question related to the one explored in the previous figure: if Subproject A begins before B, but they overlap by exactly δ *time units, will they exceed resource capacity? This can be answered for each value of* δ*, by adding* $rs(t_a=\delta,B)$ *to* $re(t_w=\delta,A)$*. Legal values for* δ *must satisfy the constraint* $rs(t_\alpha=\delta,B)+re(t_w=\delta,A) \leq \delta$ *(for unit capacity resources). In this case, values* $0\leq\delta\leq6$ *violate the constraint. Together with the result in the previous figure, we see that no value of* δ *satisfies the constraints, and thus A and B cannot overlap.*

beginning at the latest start time (relative to the start of the subproject). The end-profile is computed as the start-profile of a “reversed” version of the original subproject. The two PLTs with *N* intervals can be summed in O(*N* log *N*) time by repeated insertion. To test condition 1 above, the summed PLT can be searched in O(*N*) time to determine whether it crosses the *y*=*x* diagonal.

Testing the second condition is more complex because the intermediate-profile is more difficult to construct. I have developed a construction algorithm that operates on a graph representing the subprojects. Arcs in the graph represent either task executions, or gaps between tasks. An arc is present between two tasks only if one must precede the other. The resulting graph is directed and acyclic. Rather than consider all possible schedules for the subproject, in computing the lower-bound, the algorithm restricts consideration to the set of simple paths (i.e., directed and non-looping paths) through this directed graph. Resource usage along any simple path is a lower-bound on the resource usage for a complete subproject schedule.

If we plotted the lower-bound resource profile for each simple path through the graph, the result would be as depicted in Figure 2-7. For singly-connected graphs (such as in the figure) with *N* nodes, there are N simple paths, each of length less than N. The calculation of the intermediate profile can be done by creating a path profile for each path, and then computing the minimum by sweeping across the path profiles in parallel, looking for intersections and updating the minimum of all the profiles. The minimum of all the path profiles is the intermediate profile.

This latter step of combining path profiles is an instance of the computational geometry problem of finding the “lower envelope” of a set of functions [22, pp. 355 ff.]. From a theorem by Sharir and Agarwal [119, p. 21], we know that there are at most $O(N^2 a(N))$ intersections in the lower envelope of our N N-segment path profiles, where $\alpha(N)$ is the extremely slow-growing inverse of Ackerman’s function (for all practical purposes, this factor is a constant). At each intersection, we must do some updating to maintain the minimum profile at each intersection: this adds a “sorting” factor of $O(\log(N))$ (again following a theorem of Sharir and Agarwal [119, p. 136]). The time complexity of this step for singly-connected graphs is thus polynomial: specifically, the complexity is in $O(N^2 a(N) log(N))$. Once we have constructed the intermediate profile, it may have $O(N^2 a(N))$ segments: to add it to the start profile of another subproject has time complexity in $O(N^2 a(N)\ log(N a(N)))$, which is the time complexity of checking for an MPC2 ordering constraint between two subprojects with at most N tasks each. Recall that N is the number of tasks *in the subproject*: this is likely to be much smaller than the number of tasks in the project.

There is one major open problem with this approach. How do you choose subprojects or other aggregations of tasks? In some cases, we may be able to use a task hierarchy specified by the user (as in a task-decomposition or hierarchical planner). Our motivation for this approach is in astronomy applications, where the “observing programs” of different groups of astronomers are natural subprojects that do not require a general theory of abstraction and aggregation. In satellite scheduling, a similar subproject type is known as a “periodic task” [103].

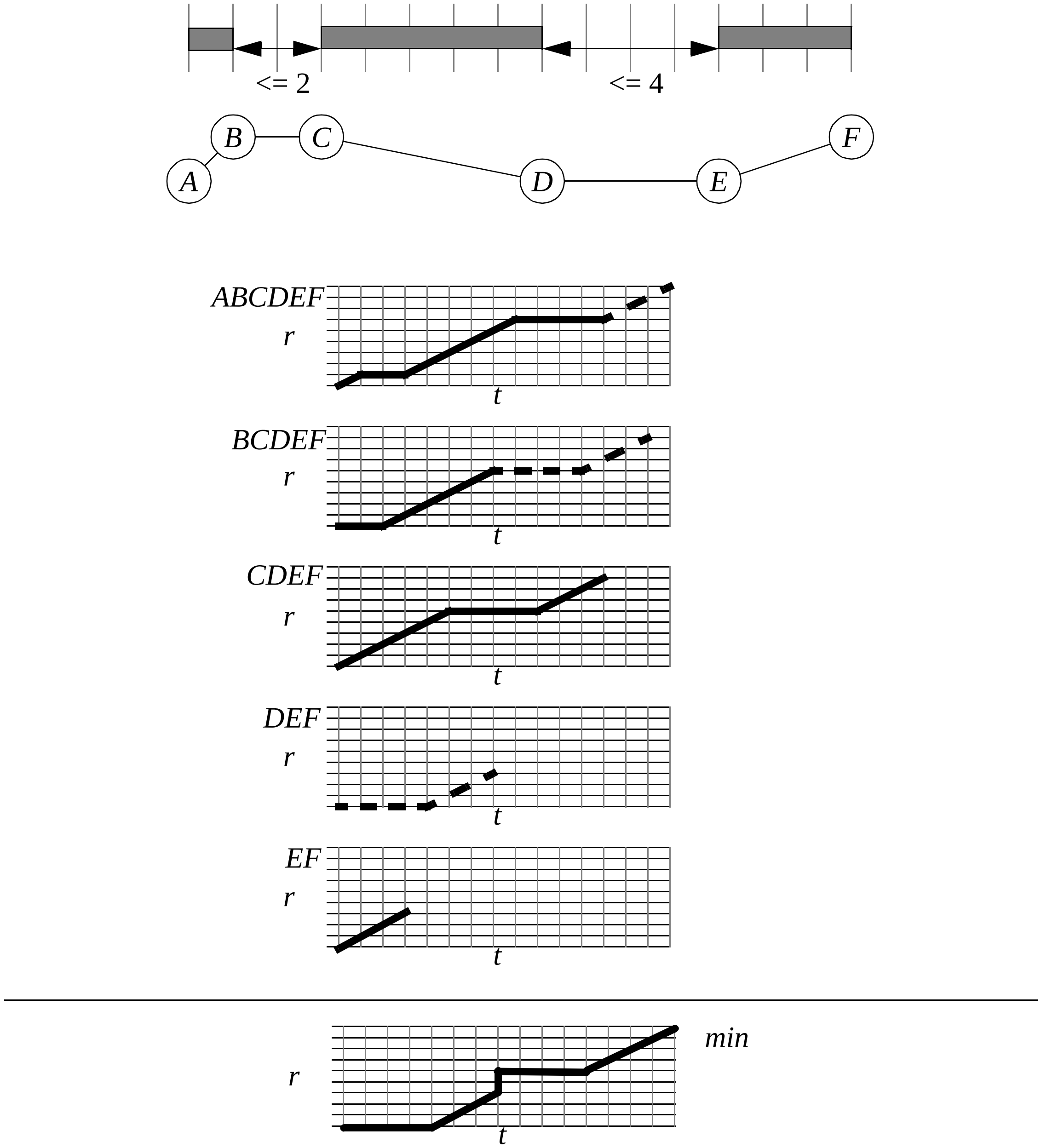

Figure 2-7. Lower-bound on simple paths through subproject constraint graph.

Dashed portions of the graphs are parts of the lower-bound intermediate-profile.

At the end of this chapter, I describe experiments that validate the effectiveness of MPC2 in finding new precedence constraints between subprojects in a test suite of problems.

2.5. MPC Extensions

The basic MPC techniques can be extended in a number of additional ways. Three straightforward extensions are sketched in this section.

2.5.1. Handling other Temporal Constraints

As described, the MPC technique combines resource constraints with precedence constraints to derive tighter temporal constraints. By examining the set of tasks *Y* such that *X* precedes *Y*, and *Y* precedes *Z*, MPC1 can establish a stronger constraint between *X* and *Z*. Strict precedence is a convenient simple case: it allows us to determine that the task *Y*, with duration *D*, will use *D* units of a resource between the time points *X* and *Z*.

But if other constraints allow us to determine that at least *D'* units of *Y* occur between times *X* and *Z*, similar reasoning can be used. I have not explored this possibility, but Figure 2-8 depicts an example of the computation involved.

2.5.2. Handling Multiple Resources

For clarity of presentation, my description of the MPC approach has assumed a single resource. The extension to multiple resources is fairly straightforward. For example, in Listing 2-2, the loop over intermediate tasks must be enclosed in a loop over resources. For each resource, only intermediate tasks which use the resource are considered when accumulating resource usage. If an individual task requires multiple resources, it will be counted in several of the resource loops.

Constraints: X + d1 <= e, d1 >= 0
s + d2 <= Z, d2 >= 0
s + d <= e

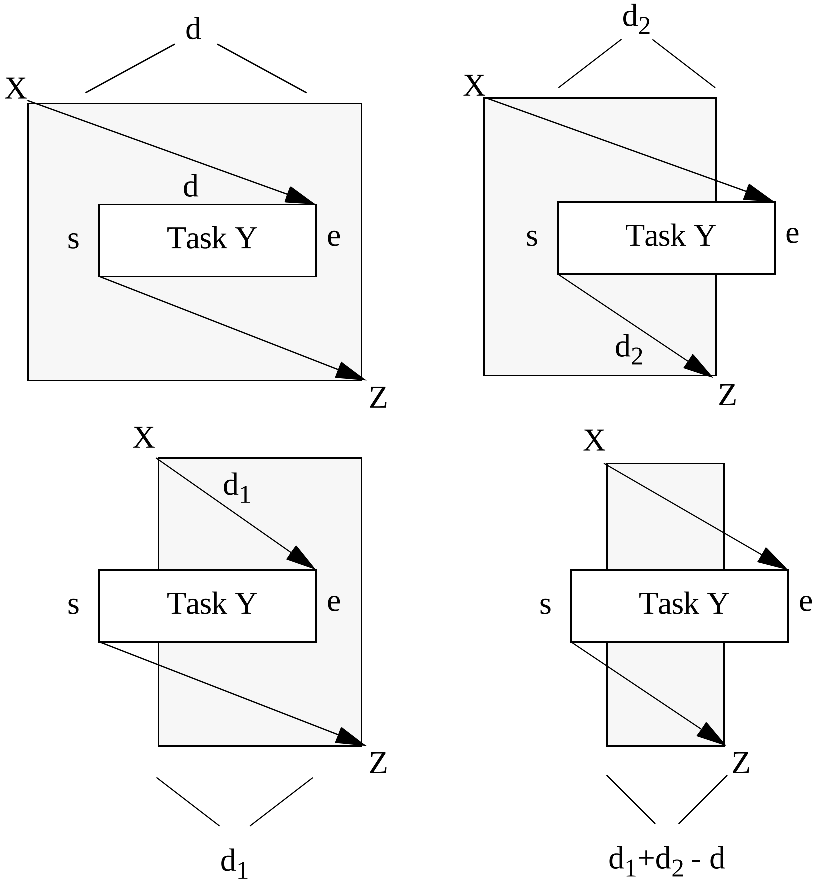

Figure 2-8. MPC with non-precedence constraints.

In this example, we have that X precedes Z, X precedes e, s precedes Z, and of course, s precedes e, where s and e are the start and end of task Y. Reasoning from the four cases for the ordering of X, s, e, and Z, we see that at least D'=min(d1, d2, d, d1 + d2 - d) units of resource are used by task Y between X and Z.

2.5.3. Handling Attribute-Based Resource Selection

Finally, a common problem in large-scale scheduling problems is reasoning about polymorphous resources: i.e., where similar resources are grouped into pools, and a task's resource constraints specify which pools can be drawn upon to provide a resource. For example, this problem arises in airline crew scheduling and maintenance, where pilots and engineers must be certified for the safe operation of individual types of equipment.

Figure 2-9 illustrates a simple example from the construction of a building. Among the ten construction workers, five are carpenters, with three of those certified as master carpenters. Another two construction workers are electricians. While some tasks can be done by any worker, others require a carpenter: a few tasks might specifically require a master carpenter.

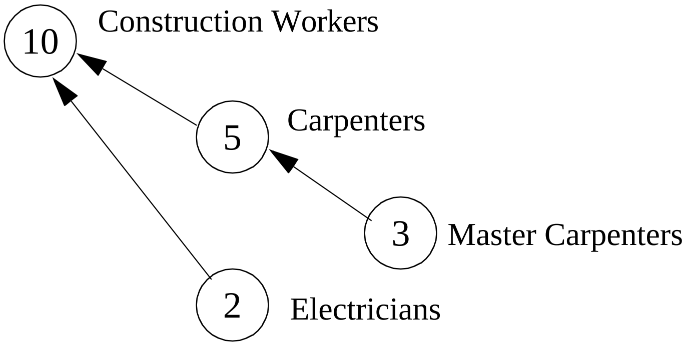

Figure 2-9. Hierarchy of available resources.

The hierarchy includes ten construction workers, of whom five are carpenters, with three of those certified as master carpenters. Another two construction workers are electricians. Note that three remaining construction workers are undifferentiated.

The basic MPC technique generalizes nicely to the problem of handling resource pools, although resource usage information must be propagated through the hierarchy. For example, a task requiring seven construction workers leaves at most three carpenters free. Similarly, a task requiring three carpenters leaves at most seven construction workers free.

The MPC1 algorithm extends to hierarchical resource pools by examining the entire resource hierarchy for every pair of events *X* and *Y*. For each pair and resource type, the resource requirements are computed for the set of tasks *S*={*Z*:precedes(*X*,*Z*) ^ precedes(*Z*,*Y*)}, which is then used to provide a possibly tighter constraint between *X* and *Y*.

2.6. Evaluation

I have evaluated MPC1 and MPC2 on two related test suites of scheduling problems.

It can be difficult to separate the effectiveness of preprocessing algorithms from the search algorithms that follow them. Certain discovered constraints may yield no benefit in certain algorithms, or even slow down the algorithm. For example, a preprocessing algorithm may be geared to discover a certain type of constraint, but this constraint may only be valuable if it is used in a forward-checking phase of the search algorithm (in other words, tested at each search tree node to determine if no feasible solutions exist below that node). As another example, stochastic search algorithms are better able to exploit reduced domain sizes than new precedence constraints. On the other hand, precedence constraints are more crucial to certain backtracking search algorithms which search over the space of partial orders.

Accordingly, I have used a *factored benchmark* approach. I first quantify the effects of the preprocessing algorithm in terms of a search space parameter (number of constraints, size of domains, etc.), and then run separate tests to see how the preprocessing algorithm improves search efficiency. By analyzing both the search space parameters and search efficiency for different problem parameters, we can better understand the mechanism by which the preprocessing algorithm increases search efficiency, and how this effect varies for different problem types. When deciding to use a preprocessing technique on a *new problem class* with a *new search algorithm*, one can separately decide whether the problem class is suitable for the preprocessing algorithm, and whether the output of the preprocessor will influence the particular search algorithm.

MPC1 was evaluated by generating random constraint graphs over tasks which all required a single unit-capacity resource. For each such graph, there is at least one schedule over the tasks which satisfied the constraints: this is called the nominal schedule and it is generated first so as to guarantee feasibility. The test problems varied along three parameters:

- resource utilization, varying from 0 to 1, indicates the proportion of time the resource is used in a feasible schedule. Higher resource utilization should increase the effectiveness of MPC1, as there is more resource contention in the problem. Resource utilization was varied by introducing random-duration gaps between tasks in the nominal schedule.
- constraint density, varying from 0 to 1, indicates the probability that a precedence constraint consistent with the nominal schedule would be present in the constraint graph.
- number of tasks in the schedule. Tasks have a mean duration of 10.0 time units, and resource utilization is varied by altering the mean duration of the inter-task gaps (for utilization of 0.67, the inter-task gaps are chosen to have mean duration of 5.0, half the mean duration of the tasks).

Because no problem test suite is perfect, I have varied the test suite parameters in my experiments, so that their effect on results and conclusions might be seen more clearly.

Experimental results for MPC1 are summarized in Table 2-3. The table presents data on the average “search tree size” under varying test problem parameters and preprocessing algorithms. The search tree size is calculated as the product of the domain size of each task in the problem (domain size is the difference between latest start time and earliest start time for a task in the problem). The table shows the base 2 logarithm of search tree size, which we would expect to be roughly proportional to the log of search cost, although this will depend on the number of acceptable solutions, and the precise search algorithm being used. Halving the log search tree size (as often occurs in this table) would correspond to, for example, reducing the domain size for each task to the square root of its original size, or reducing the depth of the search tree by a factor of two (logarithm of original search tree size = $\log(b^d) = 2 \log(b^{d/2}) = 2 \log ((b^{1/2})^d)$.

			number of tasks								
			30			40			50		
constraint density = .2			Start	DMP	MPC1	Start	DMP	MPC1	Start	DMP	MPC1
	resource utilization	.50	246.2	229.0	220.7	346.8	325.9	312.5	451.9	427.2	407.0
		.75	240.6	218.2	200.2	341.4	313.9	283.7	443.8	412.1	368.0
		.90	246.8	221.8	197.8	348.5	318.6	279.1	454.2	418.5	360.7
		1.00	250.9	226.5	201.3	352.2	320.2	277.5	458.4	421.5	359.3
constraint density = .5											
	resource utilization	.50	245.5	211.5	202.3	345.9	308.9	294.3	451.2	404.7	386.8
		.75	241.4	196.2	169.3	341.1	284.0	245.1	444.1	374.7	323.7
		.90	247.0	193.2	149.8	348.4	281.4	217.5	453.2	374.2	290.4
		1.00	250.1	195.6	145.0	353.1	286.5	211.8	460.7	375.6	279.8
constraint density = .8											
	resource utilization	.50	246.1	187.6	184.0	347.1	278.1	272.9	450.9	370.4	363.5
		.75	242.2	162.8	150.3	341.2	238.7	221.1	444.3	322.8	300.2
		.90	247.5	149.1	124.8	348.7	216.9	186.1	454.9	301.5	256.8
		1.00	250.3	144.2	116.9	353.0	215.6	176.8	459.7	305.8	246.6

Table 2-3. MPC1 performance: reduction in *log*(search tree size).

The values in this table are the base 2 logarithm of search tree size for the original problem ("start"), the output of DMP preprocessing ("DMP"), and the output of 3 iterations of MPC1 and DMP ("MPC1"). Each table entry is a mean over 60 experiments. Three subtables show results for varying constraint densities (.2, .5, and .8). The table shows that MPC1's benefits improve as resource utilization increases.

The results show that a significant reduction in search tree size is attributable to DMP preprocessing of temporal constraints. When MPC1 is run, it generates additional temporal constraints by analyzing resource capacity: this enables a subsequent run of DMP to generate even tighter constraints. In these tests, the algorithms alternate, with MPC1 run three times in the alternating sequence, and DMP starting and finishing the sequence.

As the table shows, MPC1, when combined with DMP, reduces search tree size considerably. This effect is greatest when resource utilization is high and there is much resource contention.

MPC2 was evaluated using pairs of subprojects. Each subproject was generated using the same mechanism as for generating problems for MPC1. However, the order of tasks was constrained to be identical to the nominal schedule, and each N-task subproject had N-1 internal constraints, each reflecting a maximum gap permitted between successive tasks.

The experimental question is how often does MPC2 find a constraint between pairs of subprojects, and how does this effect vary with resource utilization (the size of gaps), and with subproject length (the number of tasks). Experimental results are summarized in Table 2-4. The table shows that for resource utilization over .5, we are quite likely to find a constraint between two randomly generated subprojects. The probabilities imply that a large proportion of such subprojects will have order constraints that are a consequence of their internal resource requirements. The effect diminishes as subproject length increases, essentially because the odds of finding a sequence of long gaps increases in a longer subproject. Nonetheless, the benefits of finding such a constraint are correspondingly greater for longer subprojects.

		number of tasks in each subproject					
		2		4		8	
resource utilization	.1	.000	(100)	.000	(100)	.000	(100)
	.3	.435	(1200)	.107	(400)	.000	(100)
	.5	.893	(1400)	.779	(1600)	.638	(1500)
	.7	.995	(500)	.995	(500)	.996	(400)
	.9	1.000	(100)	1.000	(100)	1.000	(100)

Table 2-4. MPC2 performance: probability of discovering a constraint.

This table gives the probability of finding a precedence constraint between a pair of subprojects using the MPC2 algorithm. Experiments were run (in batches of 100) until the 95% confidence interval around the estimated probability was +/- .025 (sample sizes are shown in parentheses).

These results show that the MPC1 algorithm can sharply reduce domain sizes, and thus search tree sizes, and that the MPC2 algorithm can discover implicit ordering constraints by combining resource capacity and temporal constraints. The overall payoff of better preprocessing and additional constraints depends on the search algorithm used. As a “vanilla” test vehicle, I have used clp(FD), a constraint-logic programming system developed at INRIA [31, 41]. It implements a simple backtracking search, together with forward-checking, and has been applied to solve job-shop scheduling programs. The use of an off-the-shelf search algorithm limits the risk of confounding these results with implementation quirks.

Results for MPC2 are shown in Table 2-5. As the table shows, the MPC2 advantage is negligible in small problems (60 ms versus 30+18 ms in the upper-left-hand cell) but up to a factor of 10 in larger problems (3658 ms versus 376+47 ms in the lower-right-hand cell).

number of tasks in each subproject

		8			16			32		
		CLP	with MPC2	Overhead	CLP	with MPC2	Overhead	CLP	with MPC2	Overhead
resource utilization	.5	64	30	(18)	439	100	(26)	5740	378	(50)
	.7	60	30	(18)	368	100	(26)	4338	376	(49)
	.9	55	30	(18)	314	100	(26)	3658	376	(47)

Table 2-5. MPC2 performance: search costs with MPC2-discovered constraint.

This table gives the average search cost (in milliseconds) for a clp(FD) scheduling algorithm, with and without MPC2 preprocessing. Also shown are the overhead costs for the MPC2 preprocessing step. 1000 experiments were run for each of the experimental conditions.

Results for MPC1 are shown in Table 2-6. The table shows that MPC1's advantage is in large problems (here, 50 tasks), and when the problem is otherwise relatively underconstrained (constraint density .2 or .5). In these cases, MPC1 provides a factor of 2 speed-up, even on these relatively easy problems.

number of tasks

		30			40			50		
constraint density = .2		DMP	MPC1	Overhead	DMP	MPC1	Overhead	DMP	MPC1	Overhead
resource utilization	.50	178	116	23	398	227	63	790	381	113
	.75	184	127	23	406	227	63	828	395	113
	.90	180	122	23	408	236	63	826	395	112
	1.00	170	110	24	391	220	63	797	373	115
constraint density = .5										
resource utilization	.50	128	86	27	296	141	71	606	227	127
	.75	137	84	27	318	142	71	527	229	127
	.90	130	84	28	297	141	71	548	223	128
	1.00	138	73	28	306	144	71	564	220	128
constraint density = .8										
resource utilization	.50	92	78	28	194	151	72	357	233	129
	.75	98	77	28	193	142	72	319	228	130
	.90	94	79	28	204	150	72	323	227	130
	1.00	98	80	28	182	146	72	324	218	130

Table 2-6. MPC1 performance: search costs with MPC1 preprocessing.

The values in this table are search times after varying preprocessing algorithms have been applied. Search costs (in milliseconds) are shown for the output of DMP preprocessing (“DMP”), and the output of 3 iterations of MPC1 and DMP (“MPC1”). Each table entry is a mean over 60 experiments. A third column (“Overhead”) shows the cost of the additional preprocessing for MPC1, before search begins. Three subtables show results for varying constraint densities (.2, .5, and .8).

2.7. Related Work

In addition to the research mentioned elsewhere in the chapter, there are a few related research efforts worth noting.

Several AI planning & scheduling researchers have attempted to exploit resource constraints. For example, Howard A. Beck [12] has developed the *habograph*, an eponymous data structure for monitoring resource capacity constraints. His concern is to detect constraint violations during search (based on the earliest start, latest end, and capacity requirements of a set of tasks), or to detect infeasible problems which will need to be relaxed before solving. The actual calculation performed by the habograph can also be accomplished using the compiled data structures described in Chapter 3 of this dissertation.

Several researchers in constraint satisfaction and O.R. have examined resource constraints. Carlier & Pinson [*27*] developed several innovative techniques for analyzing resource capacity: for example, they identify tasks ("steps" in the job-shop jargon) which *must* occur before or after all other tasks in a job-shop scheduling problem, using a similar mix of resource capacity and temporal constraints as I have exploited here. A task with an early due-date, for example, might not be feasible if it starts after another task that requires access to the same exclusive resource. In other words, their techniques find necessary precedence constraints between tasks that require the same unit-capacity resource. Several constraint logic programming researchers [9] have also examined such disjunctive constraints in job-shop scheduling problems.

The classic O.R. approaches to scheduling and production planning have, of course, studied resource constraints. For example, Morton and Pentico's book [90] describes heuristic approaches to production planning and project management. But, tellingly, they do not mention resource capacity until p. 457 [emphasis added]: "*So far* we have considered only the important special case of project scheduling in which: (a) Activities need not compete for limited resources..." One major subfield of O.R. is concerned with "bottleneck dynamics", the identification and amelioration of resource bottlenecks [51]. As these researchers wisely point out, one can sometimes change resource capacity by renting equipment, paying overtime, or modifying processes: these are beyond the scope of the work presented here.

2.8. Summary

Reasoning about resource constraints *appears* to be far more difficult than reasoning about temporal constraints. Temporal constraints conjunctively describe a convex region. But resource constraints are disjunctive: the generic resource constraint dictates that *if* two tasks *X* and *Y* share a unit-capacity resource, *then either X* precedes *Y or Y* precedes *X*.

One lesson of computer science is that disjunction breeds complexity. For example, the problem of 3-satisfiability is far harder than 2-satisfiability simply because the former involves disjunction: the 3-clause $(X \vee Y \vee Z)$ is equivalent to $(\neg X \Rightarrow Y \vee Z)$, while the 2-clause $(X \vee Y)$ rewrites to $(\neg X \Rightarrow Y)$. After instantiating X in the 2-SAT clause, the assignment for the other variable is either forced or inconsequential: this leads to a linear-time algorithm for solving 2-SAT problems. In contrast, 3-SAT appears to require a branching search through a disjunctive tree of variable assignments, as we deal with the disjunctive consequences of our choices in search.

While disjunction is generally a complication, as in the 3-satisfiability example, the results in this chapter demonstrate that complex reasoning about resource constraints is far from impossible, and can be quite fruitful even with an investment of polynomial-time computation. The MPC techniques described here can be viewed as grouping the disjunctive constraints into bundles that can be treated conjunctively. An example conjunctive constraint that results is of the form: *regardless of whether X* precedes *Y or Y* precedes *X*, we know that *X* and *Y* will require so much resource capacity in a specified time interval.

In experimental tests, I demonstrated how MPC1 and MPC2 operate by identifying additional constraints and shrinking the size of the state-space. Even in small problems, these advantages result in speedups by factors of 2 to 10 in some cases. For larger problems, where the cost of preprocessing is easier to overcome, the additional constraint information uncovered by MPC should be even more valuable.

I expect that future work will reveal a number of improvements to MPC, including better lower bounds for the constraint techniques I have proposed, as well as better techniques for finding and exploiting tractable conjunctions of disjunctive constraints. The space of conjunctions over constraints is quite large of course (a powerset), but I expect that algorithm designers will be able to identify natural and tractable conjunctions of constraints.

3 State Generation using Compiled Interval Data Structures

> Search is ubiquitous in artificial intelligence, and the performance of most AI systems is determined by the complexity of a search algorithm in their inner loops.
>
> ----------------Richard E. Korf

Search algorithms are omnipotent but at the expense of being omnivorous: powerful generality comes at the price of being computationally intensive. If, as Korf suggests, search algorithms dominate the performance of most AI systems, search algorithms are in turn dominated by the cost of generating and evaluating states.

Scheduling problems suffer most because they have very large state descriptions, in contrast to textbook search applications such as puzzles or board games (a chess position is described by slightly more than 32 variables, one for each piece). In a scheduling search application, it can be extremely costly to generate one state from its parent in the search tree, particularly if heuristic functions must be re-evaluated over hundreds or thousands of variables in the state description.

Yet the improvement of state-generation is a non-issue from the standpoint of theoretical computer science. State generation is rarely more expensive than polynomial in the branching factor *b*, while search is exponential time. For example, if *c* is a constant, (b^d states) x (b^c computations per state) is still in O(b^d). Despite this, no practitioner would turn down a constant number of levels of search for free: and in fact, we can use improvements to state-generation to reach deeper levels of search in the same amount of time (in this example, up to *c* levels deeper). We also note that searching deeper by generating states faster has been one major focus in computer chess research, where custom hardware has been used to pipeline or parallelize the process of generating and evaluating board positions. Clever engineering of state generation has also proved crucial in successes of CSP research, such as the solution to the million Queens problem[1] and the KTS military deployment scheduler.[2]

This chapter describes a novel approach to building rapid state generators for scheduling applications. I have demonstrated that simple compilation techniques can be designed to produce efficient and bug-free state generator code from formal specifications. The compilation is easily accomplished by hand for a specific application, but automated compilation is a requirement to build practical yet general-purpose scheduling systems. As I explain in this chapter, the underlying technology suggests techniques to build incremental state-generators for search algorithms (or equivalently, incremental constraint checkers for manual schedule editors) from formal specifications of the constraints and heuristics in use.

[1] Andrew Phillips, then of NASA Ames Research Center, personal communication.

[2] Douglas Smith, Kestrel Institute, personal communication.

As is the case with preprocessing, rapid state generation is an improvement that is visible even to humans who interact with a manual scheduling system, as they must wait for constraint propagations between manual schedule changes. The problem is even more vital for automated schedulers, which generate many more states in their search for a good schedule. Finally, rapid state generation is a prerequisite to the use of statistical or machine learning techniques in scheduling: millions of partial schedules might have to be sampled before reliable search control information can be learned. The slow state-generation techniques available today make statistical learning techniques infeasible for complex constraint-satisfaction problems. Perhaps this explains why comparatively little progress has been made in learning for complex scheduling or constraint-satisfaction applications.

The chapter borrows from techniques in both computational geometry and programming-language semantics. While the work presented here is only a first step, I feel the techniques developed in these areas could have broad practical impact on AI scheduling and planning applications. Much of scheduling is concerned with intersection, ordering and other fundamentally geometric concepts: algorithms and data structures from computational geometry have proven very useful in DTS and SchedKit (recall that computational geometry data structures were also applied in Chapter Two). In addition, schedules and plans are strikingly similar to computer programs: simple compiler theory techniques originally developed for incremental editing and checking computer programs have given DTS and SchedKit a fast state-generation capability that is vital for both automated and manual scheduling.

3.1. Structure of the Chapter

Sections 3.2 and 3.3 give background information on the state generation problem, and on the interval tree data structure. In Section 3.4, I show how interval trees can be used to develop efficient state generators. The interval tree computations can be specified formally using the notation of attribute grammars, and Section 3.5 gives brief background information on attribute grammars. In Section 3.6, I describe how attribute grammar specifications of heuristics and constraints can be compiled into augmented interval tree data structures. In Section 3.7, I present a number of example state-generators developed in this way. The chapter concludes with an evaluation of the compilation approach, and a discussion of related work.

3.2. Background: State Generation

In scheduling, the major task involved in generating states is the re-evaluation of heuristic functions and constraint checks. I focus first on the issue of heuristic functions, returning to the checking of state variable constraints later in the chapter.

A heuristic function *h* is simply a function of a state. For example, a heuristic function that recommends the next task to assign is a function from states to tasks. As state *S* is modified to *S'* by applying operator *O*, we need to compute *h(S')* quickly, using *h(S)*, *S* and *O* if possible. (This is an instance of the general problem of *incremental computation*, as discussed in the Related Work section below.)

For example, consider the following heuristic:

> SIMPLE-MOST-CONSTRAINING: The task participating in the maximum number of constraints with unassigned tasks.

One approach to computing this heuristic incrementally is to maintain a heap of tasks, ordered by the number of constraints with unassigned tasks. This heap is maintained separately for each state, and includes only those tasks that are unassigned in that state. We also use a global constraint graph, where the nodes are tasks, and two nodes are connected by an edge if the two tasks participate in a constraint. When a task is assigned, we remove it from the heap, and decrement the heap-key of each of its graph neighbors.

SIMPLE-MOST-CONSTRAINING is a function of the set of unassigned tasks in a state. Its input domain is limited, and the inputs change slowly from state to state. The more expensive heuristics are those that are functions of the partial schedule represented by the state: SIMPLE-MINIMUM-DOMAIN is an example.

> SIMPLE-MINIMUM-DOMAIN: The task with the smallest range of legal start times. A legal start time is one that does not conflict with any constraints or with assigned values for other variables.

The MIN-CONFLICTS heuristic for value-selection is another example:

> MIN-CONFLICTS: For the given task, the start-time which conflicts with the minimum number of constraints.

I will describe a tree data structure to rapidly calculate and incrementally maintain the MIN-CONFLICTS heuristic. This kind of data structure will be used throughout the chapter as the basis for state-generation. Figure 3-1 depicts a tree structure imposed over the set of overlapping constraints on a single task. Each constraint might be the result of a simple precedence constraint between tasks. In the center of the diagram is a bar-chart indicating the number of constraints stacked over each interval. A new interval is begun whenever a constraint begins or ends. An “interval tree” data structure is imposed over the interval endpoints: an interval tree is simply a balanced binary tree, with each internal node representing the interval spanned by its extreme left and right descendants. I will describe the interval data structure in the next section

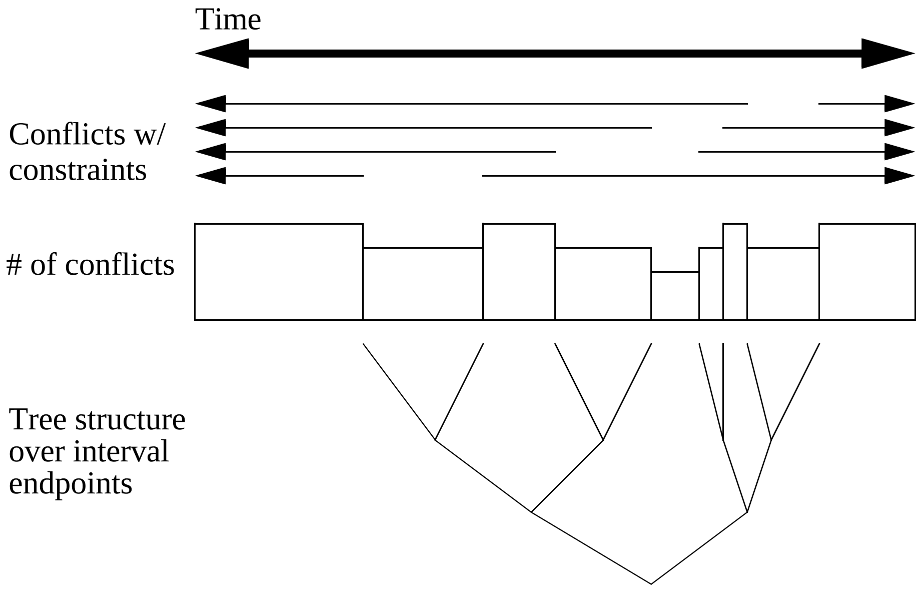

Figure 3-1. Sketch of Data Structure for Computing Min-Conflicts Heuristic.

Researchers in computational geometry developed the interval tree to deal with a variety of *intersection problems* [32, 105]. It is straightforward to “augment” an interval tree *manually* to maintain the information required to compute the Min-Conflicts or other simple heuristics. The notion of augmentation is thoroughly described by Cormen, Leiserson and Rivest [32]. In brief, augmenting a data structure involves maintaining additional information at each node, and updating that information when the tree is modified. An example would be augmenting a balanced binary tree to store the size of each subtree: the size can easily be defined for leaf nodes, defined for the parent of two subtrees, and updated when the tree is rebalanced or nodes are inserted or deleted. Given this augmentation, we can now perform size-related queries on the tree. One example is querying the "rank" of a datum: how many entries are to the left of a given datum in the tree's inorder traversal?

The DTS system which I developed several years ago used augmented interval tree data structures to efficiently compute a number of such heuristics and constraint-checks. This chapter describes and generalizes these data structures.

A naive implementation of min-conflicts is an O(R T) scan over R resources and T time points after sorting the constraints. The tree data structure for this problem (formalized in a later section) instead permits an O(1) operation to find the min-conflicts time, and O(log C) time to post or remove the C th constraint. Similar speedups are possible for other constraints and heuristics in the literature. For example, Sadeh [117] describes algorithms for calculating the "texture" heuristics that he developed with his colleagues. The algorithms presented by Sadeh involve explicit scans over resources and times. But an augmented interval tree can again provide O(1) heuristic evaluation functions and O(log C) update costs for the C th constraint.

In each such case, the augmentations of the interval tree are different. Efficient calculation of the augmentations requires just enough code optimization to require a trained programmer, and even so, the coding process is error-prone. If several heuristics and constraints are to be calculated, they can in some cases be folded into a single data structure. This would be error-prone if done by hand.

The challenge, as I see it, is to *automate* this process so that a user may specify a heuristic at run-time in a formal language (or with a graphical user interface), and have a compiler produce an efficient state-generator that relies on augmented interval trees and other data structures as necessary. Such a capability would streamline the process of adapting schedulers or planners to new application domains. In a later subsection, I explain how to apply the technology of *attribute grammars* to this compilation problem. Attribute grammars are a particularly well-understood portion of a general theory of incremental graph evaluation [62]. In essence, the techniques I describe here use attribute grammars for formalization, and interval trees for their efficient treatment of ordered attribute evaluations and particularly, associative functions such as those that arise in state generation.

Because this chapter involves techniques from two disparate fields, I have simply interleaved the discussion, as necessary, with background sections on Interval Trees and Attribute Grammars.

3.3. Background: Interval Trees

An interval tree [105, p. 352] supports queries over a set of intervals. In our application, intervals will represent the time assignment for a task, illegal time periods prohibited by a constraint, resource usage over an interval of time, etc.

An interval i is defined as a real-valued start-point ($i.l$) and end-point ($i.r$), together with an arbitrary key or datum ($i.k$). We define the sets $L(I)$, $R(I)$ and $K(I)$ as the start-points, end-points and keys of a set of intervals I. An interval tree supports natural queries on a set I, such as the following:

- Which intervals in I are *contained* in the interval $[l, r]$?
  (One interval $[a.l,a.r]$ is contained in another $[b.l,b.r]$ iff $b.l < a.l < a.r < b.r$.)
- How many intervals in I *overlap* at the point x?
  (An interval $[l, r]$ overlaps the point x iff $l < x < r$.)

Other queries manipulate the keys associated with the intervals. For example, if $f\,()$ is an associative function of a set of keys, then it is natural to ask for $f\,(\text{keys}(\ I\ ))$, where I is the set of intervals overlapping a point.

An interval tree is a binary search tree over the endpoints of a set of intervals I.[1] (I assume, for simplicity, that the endpoints are unique.) Define the list of endpoints $E(\ I\ )$ as the sorted members of the union of sets $\{i.l, i.r\}$ over all intervals i. It is convenient to number the elements of $E(\ I\ )$ as $e_1, \ldots e_n$. Each leaf corresponds to an element of $E(\ I\ )$, and the inorder traversal of the tree's leaves recreates $E(\ I\ )$. We also assume that each leaf is labeled as a right or left endpoint.

Each interior node represents the interval $[l, r]$ where l is the leftmost descendant in the subtree, and r is the rightmost descendant. Augmentations of the basic interval tree are used to compute properties of an interval. For example, each node $[l, r]$ can maintain the number of intervals that intersect the time r (i.e., the number of intervals i such that r precedes $i.r$ in the ordering $E(\ I\ )$ but r does not precede $i.l$ in the ordering $E(\ I\ )$). Call this function Balance(e_1, r): it is the basis for computing the MIN-CONFLICTS heuristic.

Note that we can rewrite Balance( ) using the following recursive definition:

[1] Some textbook descriptions of interval trees store one interval (i.e., two endpoints) per node. I have found it useful to store one interval endpoint per node instead.

$$\begin{aligned}
\text{Balance}(e_i,e_j) &= 0 && \text{if } i > j. \\
&= 1 && \text{if } i{=}j \text{ and} \\
& && e_j \in \text{L}(\ I\ ). \\
&= \text{-}1 && \text{if } i{=}j \text{ and} \\
& && e_j \in \text{R}(\ I\ ). \\
&= \text{Balance}(e_i,e_{j\text{-}1}) + && \text{otherwise} \\
&\quad\ \text{Balance}(e_i,e_j) &&
\end{aligned}$$

Note that the function decomposes into an associative function (addition) of individual terms (+1 or -1) for each endpoint. This suggests that a tree structure can be used to associate the terms into subtrees, and maintain Balance(l,r) for each subtree. Any query for Balance(e_i,e_j) can then be computed by adding the values for at most O(log N) subtrees, if the tree has N leaves. (If the tree is balanced, the nodes between any pair e_i and e_j are entirely contained in a set of O(log N) subtrees.)

As in other tree data structures, various balancing mechanisms (splaying, red-black markers, etc.) can be used to ensure that the insertion, deletion and membership operations can be accomplished within logarithmic time.

Finally, the “*X*-axis” in the interval tree need not be time, although it is in all of my applications to date. It can more generally be any total-ordering criterion. For example, if we are searching through the possible total orders on a single machine in a job-shop problem, we can construct an interval tree for each total order. A properly augmented tree could compute machine states and setup costs on the fly as jobs are inserted and deleted in the total order.

For non-total-orders, both the specification and computation of functions such as Balance becomes more complicated. However, there is significant related work to draw on in elaborating that extension of these techniques: I describe some of this related work at the end of the chapter, and until then, focus entirely on total orders.

3.4. State-Generators using Interval Trees

In this section, I describe how to represent several heuristics and constraints using interval trees. The section focuses on intuitive or geometric descriptions of how the interval trees would be used. In a later section, I give formal specifications for the tree structures using attribute grammars.

3.4.1. Representing Heuristic Functions

I will use two examples to clarify the use of interval trees to represent heuristics.

The first example is the min-conflicts heuristic, popularized in papers by Minton and his colleagues [86]. We seek a data structure that supports a query for the time point that is “in conflict” with as few constraints as possible. By maintaining such a data structure for each task, we can easily find the task that is in the minimum number of conflicts with posted constraints.

In fact, the use of intervals helps to clarify the definition of the heuristic. The data structure maintains, for each constraint on the task, one or more intervals representing the time periods excluded by the constraint. The number of conflicts for a given time is simply the number of intervals stacked over that point. This is one of the most basic queries on an interval tree.

A second example is the resource contention heuristic proposed by Muscettola and Smith [91, 92] and later studied by Sadeh [117]. This heuristic identifies the time at which contention for a resource is heaviest. Contention is defined by assuming a uniform probability distribution over the possible start times for the task. (This is obviously a heuristic assumption, and not necessarily plausible: in fact, it is not even clear that the probability distribution has a well-defined meaning.)

The simplest definition of resource contention is based on the following: if a task requires one resource unit over a duration of *d*, and it can be scheduled at any time within an interval of length *p*, then it "contributes" *d*/*p* to the contention on that resource throughout the length *p* interval. The contention over a set of tasks is defined as the sum of their individually contributed contentions for the resource.[1]

This heuristic is easily implemented by maintaining an interval tree where each interval has a height *d*/*p*. The time intervals of maximum contention are easily maintained for a given resource, and the resources can be stored in a heap, keyed by contention, to maintain the bottleneck resource as scheduling progresses.

[1] A slightly more subtle definition assumes uniform probability for the start-time of the task, subject to constraints, and yields a trapezoidal or triangular contention function.

3.4.2. Representing Constraint Checks

In addition to heuristics, I have used interval trees to support incremental constraint checks, and I describe three examples here.

The first example is the DefExec constraint that I developed while building DTS. The constraint is a dynamic form of the preprocessing techniques described in Chapter Two. Consider the two tasks that are depicted in Figure 3-2, along with their associated time bounds. The constraint is based on "definite-execution intervals" for each task. The definite-execution interval is bounded by the latest-start and earliest-finish for the task. If this interval exists (i.e., latest-start precedes earliest-finish), the task will definitely be executing during the interval. Consequently, the sum of the definite-execution intervals must satisfy resource constraints.

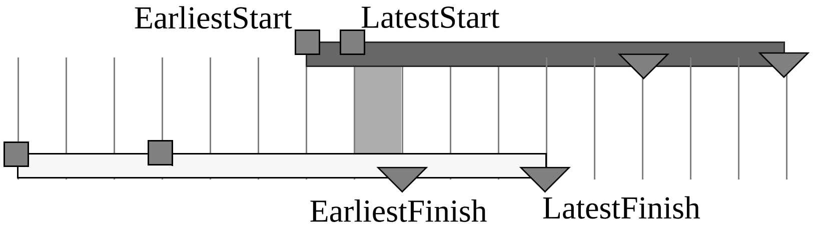

Figure 3-2. DefExec Constraint Example.

The figure shows time bounds on the start and finish of two tasks, Light and Dark. The LatestStart of the Dark task precedes the EarliestFinish of the Light task, and so it is certain that the two tasks will overlap for one time unit (grey vertical bar), even though specific start and finish times have not yet been assigned. Thus if the tasks require a common resource, two units of resource capacity will be required.

This constraint check is easily implemented using interval trees. A single interval tree stores the definite-execution interval for every task. The period during which the greatest number of such intervals overlaps provides a lower-bound on the required resource capacity. If this exceeds the available capacity, the resource constraint has been violated.

The second example is a more refined resource capacity constraint, using resource calendars. Assume that resource availability varies over time. In this case, resource availability can be considered as negative intervals (e.g., with a “height” of -*k* if *k* resource units are available), while resource usage is modeled as positive intervals (height of +*c* if *c* units are used). Resource usage intervals are the definite-execution intervals. The sum of the positive and negative intervals during any period cannot exceed zero, or the resource is oversubscribed.

A third example is provided by the Piecewise Linear Trees discussed in the previous chapter for use in the MPC2 algorithm. In this case, we want not the number of intervals over a point, but the cumulative or prefix sum of the intervals up to a point. Again, this is an extremely straightforward associative function that can be represented by an augmented interval tree.

3.4.3. Generalization to State Variable Maintenance

Interestingly, the same tree structure that is so beneficial for maintaining heuristic information, and simple constraints, can also be applied to maintain the value of state variables as tasks are inserted and removed from the schedule. Figure 3-3 illustrates the concept for a simple integer-capacity state variable (an idealized model of a battery).

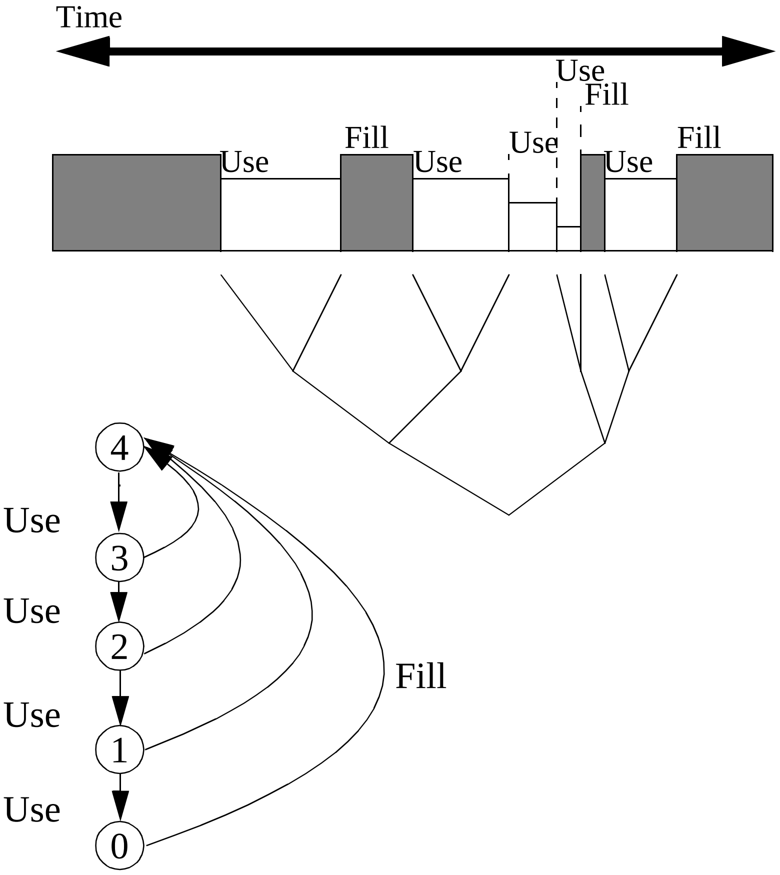

Figure 3-3. Maintaining State Information with an Augmented Interval Tree.

The figure illustrates a sequence of “fill” and “use” events on a battery with capacity of 4 units. The finite state diagram for the battery is illustrated at lower left: each state is an integer value for the remaining charge in the battery. A “fill” event recharges the battery completely. The bar chart at the center of the diagram illustrates the current battery level. To compute the bar chart, the tree structure simply maintains two values for each subtree: a flag indicating whether a fill event occured, and the number of use events after the last fill (if any).

In the example used here, a finite-state-machine characterizes the state variable's behavior. To be efficient, it must be possible to describe a sequence of transitions compactly using the state machine. In this case, the state machine is restricted: it is a "jump down-counter" in the parlance of digital logic design [69]. Thus, given a starting state, one can compute the updated state after an arbitrary sequence of transitions, if one knows the number of down transitions *since the last jump*. In terms of the tree structure, there is a compact way to encode the effect of the transitions in the interval represented by a subtree.

The problem of defining restricted languages that permit easy implementation in a data structure is the major motivation for the use of attribute grammars in this chapter. Specifically, attribute grammars fulfill the need to represent state variable behavior formally, and the grammars also provide a path to automating data structure design.

Finally, I believe that characterizing the complexity of maintaining state information will be a valuable research result in adding limited *planning* capabilities to scheduling systems. Formalization using state machines and attribute grammars may be a first step toward that goal.

3.5. Background: Attribute Grammars

I will specify heuristics and constraint checks using attribute grammars, and these specifications will then be compiled into runtime data structures. This section introduces attribute grammars.

Attribute grammars are one method for formalizing the semantics of compilers for programming languages. They were first described by Knuth in 1968 [74], and are discussed in compiler design textbooks such as those by Aho, Sethi and Ullman [6] and Waite [129]. Although they are sometimes used to formalize compilation techniques in textbooks and dissertations, it appears that attribute grammars are little used in practice.

An attribute grammar is constructed from a language grammar (e.g., a BNF) by defining attributes at each node in the parse tree (i.e., each left-hand-side of a grammar rule), and then specifying the dependencies among attributes. The attributes *X* and *Y* of node *Sched* would be denoted as *Sched.X* and *Sched.Y.* The attributes at a node can depend on its subtrees (synthesized attributes) or its parents and siblings (inherited attributes).

A classic example of attribute grammars is parsing of binary numbers [17, 74, 110]. The basic grammar rules are as follows (adapted from Bischoff's tutorial [17]):

```
bit         : ZERO
              bit.value = 0;
bit         : ONE
              bit.value = pow(2,bit.scale);
bitlist     : bit
              bitlist.value = bit.value;
              bit.scale = bitlist.scale;
              bitlist.length = 1;
bitlist     : bitlist bit
              bitlist.0.value = bitlist.1.value + bit.value;
              bit.scale = bitlist.0.scale;
              bitlist.1.scale = bitlist.0.scale + 1;
              bitlist.0.length = bitlist.1.length + 1;
num         : bitlist
              num.value = bitlist.0.value;
              bitlist.scale = 0;
num         : bitlist DOT bitlist
              num.value = bitlist.0.value + bitlist.1.value;
              bitlist.0.scale = 0;
              bitlist.1.scale = 0 -bitlist.1.length;
```

In this grammar, italicized items are non-terminals, and uppercase items are terminals. Individual bits have *value* and *scale* attributes associated with them. The *value* of the bit is 2 raised to the power *scale*. Thus, in the binary number 100.01, the *scale* for the leftmost 1 is 2, and for the rightmost 1 is -2. Figure 3-4 depicts the attributed parse tree for this number. As the figure shows, bits are assembled into bitlists, such that the *scale* of a bitlist is the scale of its rightmost bit, the *length* of the bitlist is the number of contained bits, and the *value* of the bitlist is the sum of the values of its bits.

There is a natural relationship between computations on interval trees and the computation of attributes in an attribute grammar. (In fact, I conjecture that attribute grammars can be useful as a formal means for specifying augmentations for many other tree-structured data structures, outside of the applications described here.) As demonstrated below, I have exploited this relationship to formalize the specification of efficient data structures used in state generation.

3.6. Implementation

In the case of scheduling, we can consider a sequence of events as a string in a language (or, if you prefer, a program in a language). Schedule constraints can then be expressed using the grammar itself. For example, if task *A* must precede task *B*, the grammar can be written so as to generate only schedules in which *A* precedes *B*:

```
<schedule> :=          <s> <A> <s> <B> <s>
<s> :=                 <task> <s>
                       | <emptyString>
<task> :=              <C>
                       | <D>
                       | <E>
                       | ...
```

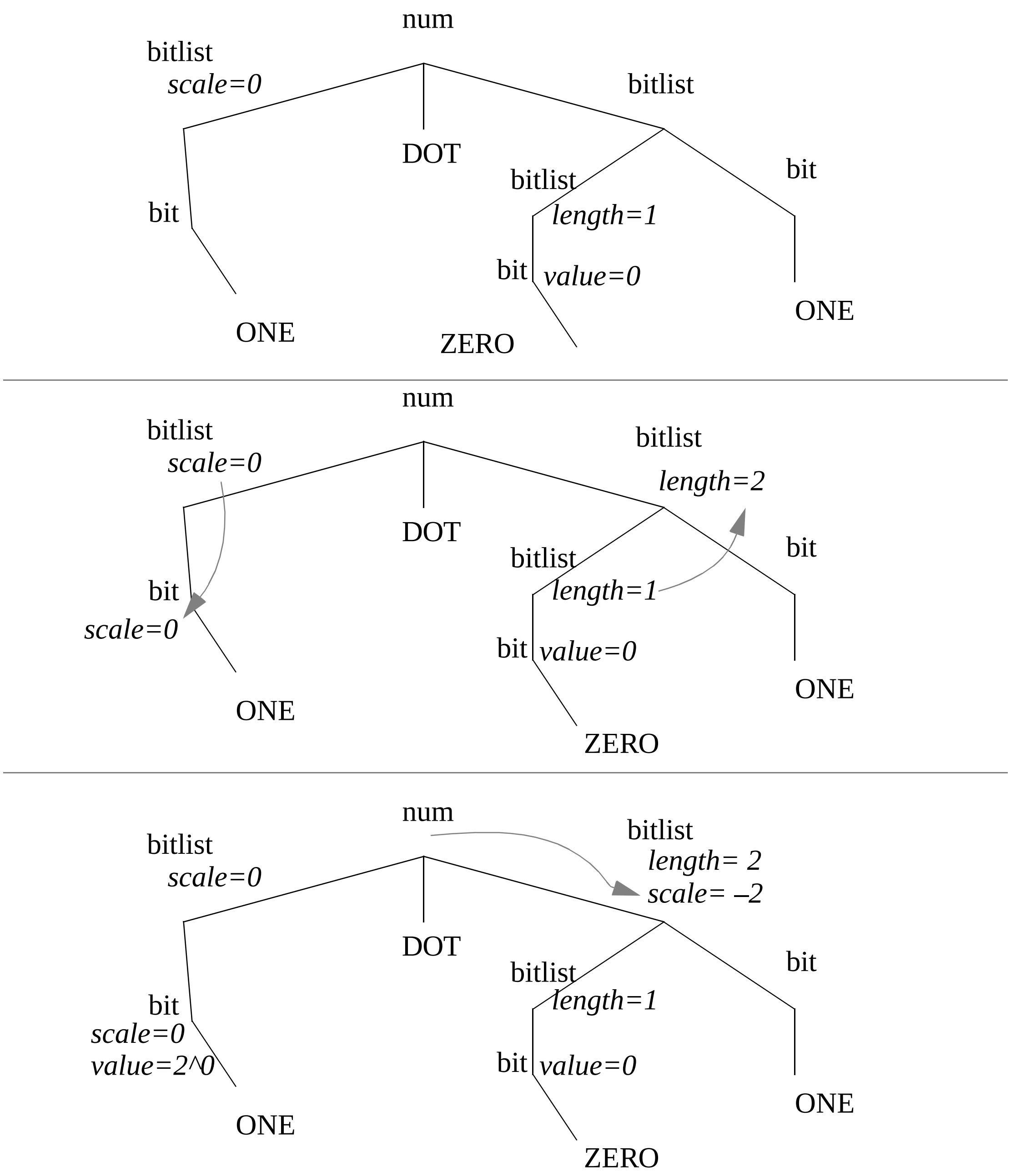

Figure 3-4. Attributed Binary-Number Parse Tree.

Attribute values are shown here in italics. The first panel shows constant attribute values from the grammar. In the second panel, attribute definitions compute the length of a left-recursive bitlist, and the scale of the leftmost bit. In the third panel, a ONE-bit with known scale computes its value, and the fraction's scale is determined by its length. Several more stages of attribute computation would evaluate the entire tree.

In practice, the difficulty with this approach is that it is cumbersome to incorporate multiple constraints. Some grammatical formalisms overcome this problem by having the grammar over-generate strings, which are then pruned back by additional constraints.

For these reasons, I will view the problem as defining a separate grammar for each state variable or constraint. The feasible schedules would then be contained in the intersection of the languages defined by these grammars. But I will also permit constraint violations in the language, which, however, will be flagged as errors using error attributes in the grammar.

Error attributes are useful for several reasons. In a manual schedule editor, the user will often be generating schedules that temporarily violate constraints: rather than constrain editing, it is better to isolate and flag errors where they occur, and permit the user to continue. The analogy to editors for computer programs is obvious. An editor that detects and marks semantic errors would be a godsend. But an editor that prevents any kind of error from being introduced, even temporarily, would hamper productivity.

Another reason to permit errors is to support search algorithms that are incremental (and repair violated constraints), or use a "repair" approach (starting with a random and probably illegal schedule, and gradually repairing it during a search).

The first scheduling example of an attribute grammar is a model of the behavior of a capacity state variable, with "fill" and "use" events. The statements in curly brackets are the specification of attributes.

```
<schedule> :=          <fillEventSeq> <usage>
                           {   schedule.end = capacity - usage.used; }
<fillEventSeq> :=      <eventSeq> <fill>
<eventSeq> :=          <fillEventSeq>
                       | <useEventSeq>
                       | null
<useEventSeq> :=       <eventSeq> <use>
<usage> :=             <usage> <use>
                           {   usage.0.used = usage.1.used + 1; }
                       | null
                           {   usage.used = 0; }
```

In this example, the *end* attribute of a schedule defines the remaining capacity after the schedule executes. This attribute is defined using the constant capacity and the *used* attribute of the usage substring. An <fillEventSeq> node spans any event sequence terminated by a <fill> event. A <usage> node spans any number of <use> events. As in a YACC grammar [80], the attribute definitions for <usage> use the notation “usage.0” and “usage.1” to refer to the parent and child <usage> nodes respectively.

This grammar unambiguously defines the behavior of the capacity state variable after any sequence of <fill> and <use> events. To maintain the intermediate value of the variable at any point in the parse tree, one would use inherited attributes to incorporate “left context” (in the compiler jargon) in the attribute definition. The next example—a balanced grammar—includes left context.

The tree structure of grammars suggests that it would be possible to compile this grammar into a tree data structure for maintaining state information. Two steps remain before we can consider this mapping between attributed parse trees and augmented data structures.

3.6.1. Balanced Parse Trees

The first step is to keep a balanced parse tree. Because of the last production rule in the previous grammar, one obtains a left-skewed parse tree, which would produce O(*N*) access time for a tree with *N* leaves. In a data structure, however, one wants to maintain balance to ensure rapid O(log *N*) processing of queries. We can permit balance by introducing ambiguity (multiple parses) into the grammar, as follows:

```
<schedule> :=          <schedule> <schedule>
                           {   schedule.1.beg = schedule.0.beg;
                               schedule.2.beg = schedule.1.end;
                               schedule.0.end = schedule.2.end; }
                       | <fill>
                           {   fill.beg = schedule.beg;
                               fill.end = capacity;
                               schedule.end = fill.end; }
                       | <use>
                           {   use.beg = schedule.beg;
                               use.end = use.beg - 1;
                               schedule.end = use.end; }
                       | null
                           {   schedule.end = schedule.beg; }
```

In fact, this grammar is significantly simpler than the previous one, in that it is easier to both design and read. The *beg* and *end* attributes represent the remaining capacity at the beginning and end of the interval spanned by each parse-tree node.

3.6.2. Node-Oriented Trees

The second step is to convert from a leaf-oriented tree to a node-oriented tree. Most parse trees are leaf-oriented: portions of the text being parsed are represented at the leaves, while internal nodes represent abstract aggregations (e.g., clauses and phrases). Most tree data structures, however, are node-oriented. Consider a binary tree for storing a sorted list: the sorted items are conventionally stored at each node in the tree, not just at the leaves.

We can get the usual node-oriented tree by ternary rules such as those in the following grammar (shown without attribute dependencies):

```
<schedule> :=          <schedule> <fill> <schedule>
                       | <schedule> <use> <schedule>
                       | null
```

3.6.3. Adding Persistence to the Data Structure

Persistence is a valuable feature in data structures that are to be rapidly modified while retaining previous versions. Different types of search algorithms place different persistence requirements on state generation facilities:

- Hillclimbing algorithms focus on a single state at all times, and never re-examine old states. State generation can be a simple "write-in-place" operation, and there is no need for persistence of old versions of the state.
- Backtracking algorithms visit states in an order that is best understood using a stack of states. State generation is a "push" on this stack. A backtrack is a "pop," visiting the state underneath the top-of-stack. State generation can be a modified "write-in-place" operation, where every state retains sufficient information to undo the operation that generated it from its search tree parent.
- Best-first algorithms visit states in an arbitrary order. State generation is at best a "copy-on-write" operation, where unmodified portions of states are shared to reduce storage requirements.

Interestingly, the field of scheduling has many more hillclimbing search applications than are found in other application areas, and many fewer best-first search applications. This is because scheduling states (partial schedules) are both complex and large, thus imposing high generation and high storage costs if best-first search is used. To offset these costs, I have applied standard techniques for making tree data structures "persistent," that is to reduce the costs of making and undoing incremental changes. One standard technique [42] is to have a modification to the original data structure create a second version of the data structure, with the two versions sharing many subtrees in common. Figure 3-5 illustrates the use of copying and sharing in the technique. As the figure suggests, insertions or deletions in a balanced tree require no more than O(log *N*) nodes to be copied.

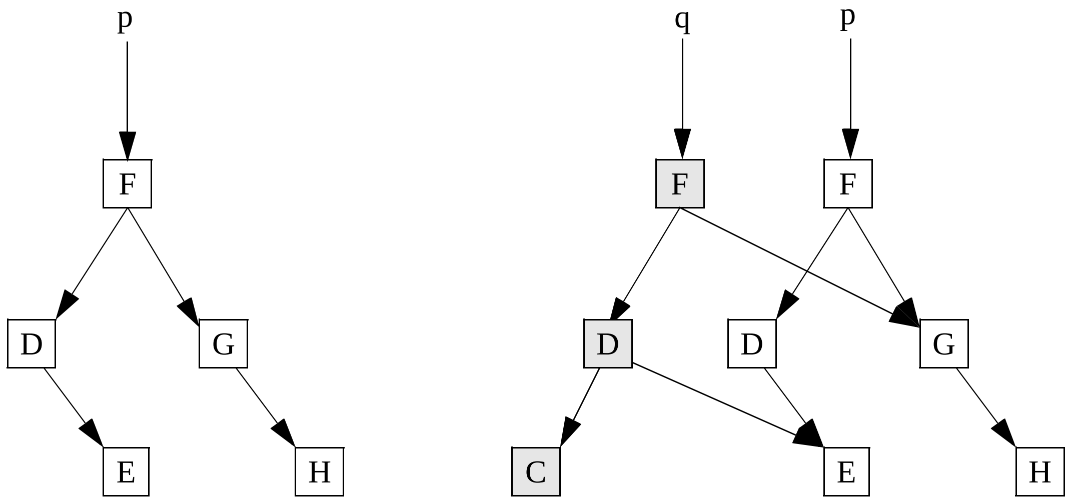

Figure 3-5. Persistent Data Structure Example.

The data structure at left is a classic binary tree, sorting the letters "D,E,F,G,H." The pointer p refers to this data structure. If we wish to insert the letter "C," and create a new data structure q, without modifying p, we can proceed by copying the tree nodes containing pointers that must be changed. In this case, the leftward-pointer in node D must be changed: so D is copied. Because D is now ambiguous, q must point to the new version of D. As a result, the nodes on the path back to the root must be copied as well. These nodes are shaded in the diagram at left. Nodes E, G, and H are shared by data structures p and q.

One difficulty with persistent data structures is that a tree-node has multiple parents (one for each version that shares the node). In Figure 3-5, nodes E and G have two parents. This makes it difficult to rebalance the tree: the rebalancing operation typically requires access to a parent pointer. I use a technique that I call *dynamic parenting* to permit algorithms for a simple, non-persistent tree to be applied to a persistent tree structure, without loss of correctness or efficiency. Briefly, dynamic parenting sets the parent pointer for node X to node Y, whenever node Y is used to reach node X. Dynamic parenting permits one to traverse, modify and then rebalance a dynamic data structure, and only consider the dynamic aspect when modifying nodes. Dynamic parenting does require a guarantee that the tree will be accessed via a single root pointer at a time: the dynamic parent pointers would become invalid by simultaneous traversals that began at different root pointers.

3.7. Examples

In this section, I provide formal specifications of the examples discussed in Section 3.4.

Several of the examples (Min-Conflicts, DefExec, and ResourceUsage) are very similar in implementation: each involves maintaining a sum of interval heights. In the case of Min-Conflicts, intervals of height one represent periods of constraint conflict. In the case of DefExec, each interval of height *k* represents a task that uses *k* units of the resource during the "definite execution" interval. ResourceUsage extends these two by allowing for variable resource availability, as well as intervals representing resource usage.

I define ResourceUsage with the following grammar. The ".k" syntax for usageOn and usageOff refers to the "key" associated with the event (in this case, the resource capacity required). Each event also has a ".t" attribute which stores the time index for the event.

```
ATTRIBUTE ResourceUsage {
<schedule> :=                    <schedule> <usageOn> <schedule>
                                    {   schedule.1.beg = schedule.0.beg;
                                        usageOn.beg = schedule.1.end;
                                        usageOn.end =
                                        usageOn.beg + usageOn.k;
                                        schedule.2.beg = usageOn.end;
                                        schedule.0.end = schedule.2.end; }
                                 | <schedule> <usageOff> <schedule>
                                    {   schedule.1.beg = schedule.0.beg;
                                        usageOff.beg = schedule.1.end;
                                        usageOff.end =
                                        usageOff.beg - usageOff.k;
                                        schedule.2.beg = usageOff.end;
                                        schedule.0.end = schedule.2.end; }
                                 | null
                                    {   schedule.end = schedule.beg; }
}
```

A ResourceCapacity attribute is easily defined using an almost identical grammar. We can then define a ResourceUnderflow attribute as follows:

```
ATTRIBUTE ResourceUnderflow = ResourceCapacity < ResourceUsage
```

This syntax defines a local boolean attribute ResourceUnderflow, which can be computed from the other two attributes.

The second example is the Piecewise Linear Tree, used in Chapter Two. These are used to calculate the “start-profile” and “end-profile” of subprojects in the MPC2 algorithm. The start profile is defined by computing ResourceUsage (as above), then applying the average operator to compute average usage, and then multiplying the average usage by each event’s time.

```
ATTRIBUTE StartProfile = (avg(ResourceUsage) * t)
```

The “average” operator simply expands into specialized code which maintains and recalculates the average (and also creates and maintains a “length” attribute for the length of each subtree). The StartProfile is used to maintain the maximum cumulative resource usage over the first *t* seconds of the subproject.

The third example is the representation of the “battery” state variable, discussed above. This attribute models the state of a battery given a sequence of Fill and Use events. The grammar is as follows:

```
ATTRIBUTE BatteryLevel {
<schedule> :=                   <schedule> <fill> <schedule>
                                    {   schedule.1.beg = schedule.0.beg;
                                        fill.beg = schedule.1.end;
                                        fill.end = capacity;
                                        schedule.2.beg = fill.end;
                                        schedule.0.end = schedule.2.end; }
                                | <schedule> <use> <schedule>
                                    {   schedule.1.beg = schedule.0.beg;
                                        use.beg = schedule.1.end;
                                        use.end = use.beg - 1;
                                        schedule.2.beg = use.end;
                                        schedule.0.end = schedule.2.end; }
                                | null
                                    {   schedule.end = schedule.beg; }
}
```

3.8. Evaluation

It is somewhat difficult to evaluate a compilation technique such as the one described in this chapter. The main claim is that the technique is general, and applies to many heuristics and constraints: this has been demonstrated in the previous section.

The second claim is that the technique produces code that is expensive to produce by hand. My own experience confirms this, and was the inspiration for trying to apply formal incremental computation techniques (such as attribute grammars) to the problem of designing incremental data structures for use in search algorithms.

Specifically, in the DTS system, I manually coded augmented interval tree data structures for a subset of the heuristics and constraints specified in the previous section. These data structures and supporting infrastructure were about 6000 lines of C++ code and header files, which went through another 6000 lines of source-code changes/additions/deletions over an elapsed year of development and debugging. You may now guess my motivation for developing better tools for this problem.

Yet I had to develop such efficient data structures in order to improve the performance of my scheduling search algorithms: I was attempting to use learning techniques, and even if the search was sufficient to solve problems efficiently, it had to be 100s or 1000s of times faster in order to generate sample data for learning. Over the course of that year of manual development, I speeded up my naive implementations by over three orders of magnitude on the problems I was solving (typically hundreds of tasks, with complexity reductions from roughly O(N) to O(log N) as described earlier).

To confirm this performance improvement in isolation, I developed a new "naive" Java implementation of the min-conflicts heuristic, and compared it to an interval-tree Java implementation. I then timed the basic operations for varying number of constraints (each constraint corresponds to an interval that must be reasoned about): results are shown in Table 3-1.

Finally, I note that the data structures produced by the attribute grammar compiler are almost identical in memory and CPU usage to the manually coded versions, largely because they have been designed to use the same infrastructure of red-black balanced trees.

		naive heuristic evaluation	interval-tree heuristic evaluation	interval-tree update (new constraint)
number of constraints	10	15.8 ms	.2 ms	3.6 ms
	100	146.7 ms	.2 ms	3.8 ms
	1000	1512.9 ms	.2 ms	5.6 ms
	10000	19664.2 ms	.2 ms	8.5 ms

Table 3-1. Performance of naive and interval-tree min-conflicts heuristics

This table gives the average time for the basic operations in a naive implementation and an interval-tree implementation of the min-conflicts heuristic. All operations are coded in Java and times are averaged over 10000 function calls in unrolled loops. As the table shows, the naive evaluation cost is linear in the number of constraints. The two interval-tree operations are O(1) and logarithmic in the number of constraints.

3.9. Related Work

The problem of computing heuristics efficiently has some extremely interesting special cases. Consider mixed-integer programming. Branch-and-bound solutions to mixed-integer optimization problems use the “relaxed model” formalism to compute lower-bound heuristics on the optimal solution to each subproblem. The relaxed model in this case is a linear program, derived by assuming that the integer variables can in fact take on non-integral values [134].

The effort expended to devise efficient algorithms to calculate this heuristic efficiently, and if possible, incrementally, illustrates the problem I am addressing in this chapter. If the linear program heuristic could not be calculated efficiently, branch-and-bound techniques would be hopelessly impractical. Furthermore, the heuristic value for similar subproblems should be used when recalculating the heuristic at a nearby node of the search tree. Again, without techniques for doing so, branch-and-bound techniques would be limited to extremely small problems. Of course, researchers *have* directed prodigious efforts toward the rapid evaluation of these linear program heuristics. For example, after man-months of reformulation and approximation efforts, a group at Georgia Tech managed to solve each linear program relaxation of their airline fleet allocation problem in only a few *minutes*. It is hard to imagine, but each of these several minute periods is inside a subroutine within the state-generation routine of a branch-and-bound search algorithm. Luckily, the branch-and-bound search for this problem generates only a few dozen nodes before finding a solution.[1]

A similar example, but with vastly different speeds, arises in computer chess programs. The computer chess advocates of "brute-force" intelligence are in fact relying on advanced hardware solutions to the problem of generating states and evaluating heuristics. Even the software-only computer chess programs (such as the Hitech simulator [15]) dedicate well over 50% of their source code to the problem of generating and evaluating states.

Even in the lowly Fifteen-Puzzle, similar problems arise in state generation. The Fifteen-Puzzle implementations I used in previous research explores 4.5 million states per second on a 270-Mhz UltraSPARC-IIi CPU.[2] A naive implementation—without incremental state-generation and heuristic evaluation—is fifteen times slower, searching only 300,000

[1] George Nemhauser, invited talk, ORSA/TIMS 1992, San Francisco.

[2] Thanks to Rich Korf for making the original versions of these programs available to me.

states per second. These examples suggest to me that the lack of techniques for developing state generators and heuristic-evaluators hinders the development of new heuristic search applications, impedes the statistically significant evaluation of such applications, and effectively blocks the development of learning systems, some of which must run millions of problem instances to tune their performance (e.g., Deep Thought [98], Neurogammon [123]).

Among the applications of attribute grammars is the specification of language-specific editors [110, 111, 112]. The basic idea in such an editor is to represent the program (or a file) as an attributed parse tree. As the file is edited, the editor ensures that attributes are updated as necessary. Often, an intermediate version of the program will be ungrammatical. This is modelled in such an editor by extending the grammar (so that every program is "grammatical", i.e., generated by the extended grammar) but incorporating "error attributes" to flag the errors. An attributed tree whose error attributes are not set corresponds to a legal program in the original grammar.

In other practical work on attribute grammars, Bischoff [17, 18] has developed a software tool, OX, which is an attribute grammar extension to the standard YACC/LEX tools. In addition, attribute grammar specifications have been done for a number of languages, including ADA, Simula and Pascal [110, p. 24].

3.10. Summary

The initial motivation behind this work was the problem of computing heuristics faster. One obvious approach is to compute them incrementally, using an appropriate data structure that is inexpensive to update. A tree-based data structure suggests itself for many heuristic functions: if properly designed, the heuristic value can be read off the root node. The resulting data structure is unusually general. For example, the tree structure can be used to maintain information on state variables (e.g., battery power, machine configuration), with most heuristics being a simple kind of state variable. In this chapter, I have shown that the specification of the tree structure can be formalized using techniques derived from compiler design. It is possible to compile a state-generator from a formal specification of the heuristics and state variables that must be maintained as each state is generated.

In short, I have examined a flexible data structure for efficient and incremental calculation of heuristics and constraint checks, and then described how the data structure can be compiled from a declarative specification of the heuristic functions and constraints.

I have found that many scheduling heuristics that I have encountered can be represented within this interval tree framework. In fact, the specification syntax and use of attribute grammars was inspired by reverse-engineering several hand-coded incremental state-generators that I developed in a previous scheduling system. I believe that the technology described here will prove useful in a number of search and mixed-initiative problem-solving applications.

4 Decision-Analytic Search Ordering

> My thesis is that some rather straightforward, simple, nonesoteric analysis of complex decisions can make a net positive difference in society.
>
> ----------------- Howard Raiffa

Few complex decisions are more in need of “straightforward, simple, nonesoteric analysis” than the millions of decisions made each second by computer programs acting on our behalf. After all, if we are to take seriously the notion of software *agents* [45, 120], then software should behave as the *agent* to our *principal*, i.e., with our preferences and beliefs in mind. Where possible, individual decisions should be made *as if* the human were making them, but at speeds that humans could never achieve.

As Russell and Wefald [116] point out, computer programs take actions not only at the “object-level” (decisions about external entities, such as actions in the world) but also at the “meta-level” (decisions about internal entities, such as computations and beliefs). The view that computations should be considered as actions—subject to deliberation and rational choice—has been particular appealing to designers of search algorithms [11, 55, 58, 83, 113, 115, 116] because of the potential for rationalizing the control of search.

This chapter describes the application of decision theory to the computational model underlying common search algorithms such as branch-and-bound and backtracking. My goal in using decision-theoretic search algorithms in scheduling applications is to reduce the number of states examined: my hope is that decision-theoretic tools can provide this benefit by explicitly trading off the cost of search against the gamble that a better solution will be uncovered by continued search, and by ordering the search in the hope of finding good solutions as quickly as possible. In this chapter, I focus on search ordering.

I began this research as part of the Bayesian Problem-Solver (BPS) project [53, 54, 83]. The motivation for BPS can be summarized as follows:

- Search algorithms, and problem-solvers in general, recommend actions to their users (or take actions on their behalf): they solve the “action selection” problem. In most interesting cases, there is insufficient information to prove that the selected action is optimal, or even that it will achieve the required goals or satisfy the maintenance of other necessary conditions.
- Action selection is thus a problem of decision-making under uncertainty. Basic consistency properties of rational decision-making under uncertainty require that a consistent “rational” decision-maker will choose the alternative with maximum expected utility.
- To implement this approach, the technical problem is to compute expected utilities given the available information (heuristic information, previous problem-solving experience and background knowledge).
- Expected utilities for alternatives can be computed after first computing distributions over utility attributes. These distributions can be computed in appropriately designed Bayesian Network structures [101, 102]. These networks include “evidence nodes” for the raw heuristic evaluation functions. Conditional probabilities constrain the relation between evidence nodes and utility attributes, based on learning from experience, as well as constraints of the problem domain (e.g., in path-planning, the triangle inequality constrains the *solution-length* utility attribute).

In previous work with Andrew Mayer, I applied this approach to single-agent problem-solving and two-player games, in each case using information from partially-explored search trees to estimate expected utilities, which are then used to drive maximum-expected-utility decision-making. In Mayer's dissertation [83], this approach is demonstrated to improve decision quality in a single-agent limited-time search problem, albeit with considerable computational overhead. Interestingly, the estimates of expected utility from the BPS-inference algorithm are so accurate that it can predict its own decision quality. Mayer's dissertation focused solely on the problem of inference from partial search trees. In this chapter, I use very simple inference techniques, but focus on how to use inferred probabilities and utilities to control search.

4.1. Structure of the Chapter

Sections 4.2 and 4.3 present background information on branch-and-bound and decision theory. Section 4.4 motivates the application of decision theory to heuristic search. Section 4.5 analyzes the decision trees for the search ordering problem, and describes ordering heuristics for the case of linear and exponential utility functions. Finally, Section 4.6 provides a demonstration of the BPS search-ordering heuristic in the context of a propositional satisfiability search algorithm. Section 4.7 describes related work, and Section 4.8 summarizes the chapter.

4.2. Background: Branch-and-Bound

The theoretical contributions described in this chapter have been applied to improve the speed of branch-and-bound search for scheduling applications. The branch-and-bound technique is used in a wide variety of algorithms. As Nau, Kumar and Kanal [97] have shown, algorithms as varied as *A**, *AO**, *B** and *SSS** can be brought under the branch-and-bound framework. In addition, branch-and-bound is the main loop of important, practical algorithms for solving integer programming (IP) and mixed integer programming

(MIP) problems (see chapters 7 and 8 of the textbook by Williams [133]). In future work, I hope to apply decision-analytic techniques to all of these areas because of the broad impact it could have.

An instantiation of the general branch-and-bound algorithm is composed of a specific branching rule and a specific bounding rule. Like all search algorithms, branch-and-bound is concerned with efficiently exploring a large, implicitly specified set of possible solutions. The branching rule simply takes a *set* of possible solutions and returns a set of exhaustive and mutually exclusive *subsets* of possible solutions. The bounding rule computes a bound on the maximum value of the objective function within a set of possible solutions. The intuition behind branch-and-bound is that subsets can be ordered by their bounds so as to find a good solution quickly, and after a solution is found, the updated bounds can eliminate entire subsets from consideration. In this way, the entire set of possible solutions can be searched faster, because many subsets (or nested subsets) can be eliminated by the ever-improving bounds.

A simple depth-first branch-and-bound algorithm is presented in Listing 4-1. The branching rule, together with the initial set of solutions, determine a tree-structured search space. The bounding rule permits this tree to be searched selectively. Specifically, the algorithm maintains a current bound, indicating the objective function value of the best solution found so far. When exploring any set of solutions, step 2 tests whether the set can possibly contain a solution of greater value. If not, the set is said to be *fathomed*. Otherwise, the set will be branched into exhaustive and mutually exclusive subsets, which will each be searched recursively. As solutions are found, the bound is updated if necessary.

```
Bound DFBB(Set s, Bound b) {
    Bound sb=upperbound(s);
    if (sb <= b) {
        return sb;            // return a value <= b to indicate fathomed node
    } else if (terminal(s)) {
        if (sb > b) { return sb; } else { return b; }
    } else {
        List subs = branch(s);                    // create a list of subsets of s
        Set ss;
        while (ss = pop(subs)) {                           // iterate over subsets
            Bound newb = DFBB(ss, b);          // recursively find bound in subset
            if (newb > b) { b = newb; }       // found better solution; raise bound
        }
        return b;
    }
}
```

Listing 4-1. Depth-First Branch-and-Bound.

The example of *A** [100] should suffice to make branch-and-bound concrete for readers with an AI background. The *A** algorithm searches for a minimum-cost path through a graph, between an initial state *I* and any member of a set of goal states *G*. The graph is defined as a set of states *S*, and a set of operators *O*. The states and operators define a partial function: $S \times O \rightarrow S$ that maps states and operators to states. We can define the space of possible solutions as containing all operator sequences. The *A** algorithm begins with the set *s* defined as the set of all sequences. The branching rule partitions this set based on the first operator chosen. Subsequent applications of the branching rule result in a prefix-tree of operator sequences: each node in the tree represents all sequences with a given prefix. Because *A** is best-first, it uses the slightly modified algorithm schema presented in

Listing 4-2. In best-first branch-and-bound, a set of sets is maintained, representing a global partition of the original set. At each step, the set with the lowest upperbound is chosen from the global partition. The branching rule then partitions this set, adding the resulting subsets to the global partition. This can be visualized as a best-first search of the prefix-tree of operator sequences.

```
Bound BFBB(Set s, Bound b) {
    Heap open;
    Bound sb = upperbound(s);
    insert(open, s, sb);
    while (s = delete-min(open)) {
        sb = upperbound(s);
        if (sb <= b) {
            return sb;
        } else if (terminal(s)) {
            if (sb > b) { b = sb; }
        } else {
            List expansion = branch(s);
            while (node = pop(expansion)) {
                insert(open, node, upperbound(node));
            }
        }
    }
    return b;
}
```

Listing 4-2. Best-First Branch-and-Bound

In A*, admissible (under-estimating) heuristic functions provide lower-bound information on the set of sequences with a given prefix. Specifically, if the cost of an operator prefix plus the heuristic function value exceeds the current bound, the prefix or path cannot contain a solution better than the bound. After a solution is found, it provides an upper bound on acceptable solutions, and some subtrees will be *pruned* from the search tree, if their lower bounds exceed the upper bound.

4.3. Background: Decision Theory

I introduce the topic of decision theory by focusing on the specific problems of reasoning about uncertainty that are associated with branch-and-bound algorithms. For a good general introduction to decision theory, see the textbook by Clemen [30].

Traditionally, branch-and-bound algorithms such as A* have used *objective functions* to rate solutions. Because an objective function is used only to *compare* alternative branches or solutions, it is only required to impose a reasonable ordering on solutions. Any order-preserving (i.e., monotonically increasing) transformation of the objective function yields identical behavior.

Branch-and-bound algorithms also use *heuristic functions* and *bounding functions*. In some cases, all three (objective, heuristic and bounding functions) are computed using the same function. Like objective functions, the definition of heuristics is incomplete and underconstrained: heuristics need only provide an ordering on subtrees in the branch-and-bound search. Any order-preserving transformation yields identical behavior. Bounding functions (e.g., admissible heuristics) must provide a true lower or upper-bound, but that still underconstrains their definition.

If we wish to handle either uncertainty or computation time in our branch-and-bound algorithm, we must make additional demands on the heuristic function. It so happens that these demands are sufficient to force a precise definition for the information desired from a heuristic function.

If uncertainty is added, then we must be able to assign preference to *gambles*, decisions in which the solution or objection function value is not certain. Every decision in branch-and-bound is a gamble. For example, to order search after branching is to gamble that one search order is better than another. Also, to terminate search early, or set an aspiration level, is to gamble that more investment in search time would not yield a better solution.

Specifically, a gamble is a probability distribution over the possible outcomes of a decision. An outcome can be defined as the measurable attributes of the solution that will result from a decision. A choice between two simple gambles is shown in Figure 4-1.

By recording preferences over such gambles, we can construct a *utility function* that encodes the user's preferences, assuming that the user's preferences obey natural axioms of consistency in decision-making (specifically, the axioms of utility theory and decision theory [118]). The utility function can then be used to compare any two gambles with the user's preferences in mind. The utility function will satisfy the fundamental property that the decision-maker will be indifferent between a *known* solution with utility *u*, and a gamble between uncertain solutions whose *expected* utility is *u*. I will typically define a utility function as a function of a set of utility attributes (such as computation time, schedule feasibility and schedule cost). Because the expected utility is only used to compare gambles, the utility function can be scaled or offset (i.e., affine-transformed) without changing the

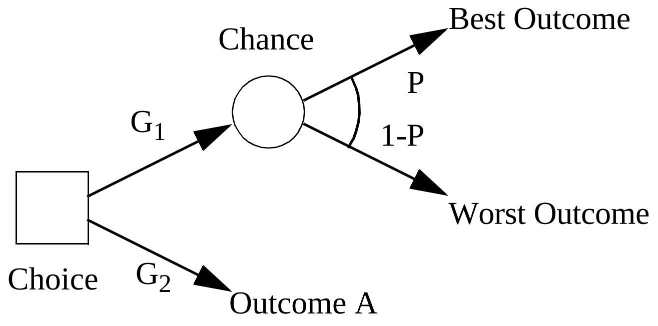

Figure 4-1. Simple Gambles used for Eliciting Utility Function

The diagram illustrates a choice between two gambles. Gamble 2 (G_2) offers a certain chance at a particular outcome. Gamble 1 offers an uncertain chance between the worst and best outcome. By definition, a decision-maker will prefer Gamble 1 if P=1, and Gamble 2 if P=0. Therefore, there must be an intermediate value for P which makes the decision-maker indifferent. This value is called the *preference probability,* or *utility* for outcome A, i.e., U(A)=P.

decision. But by convention, U(best outcome)=1 and U(worst outcome)=0, which makes the utility function identified by the constraints. The details of the elicitation and use of utility functions are sketched in Figures 4-1 and 4-2.

Decision theory defines a *rational decision-maker* as satisfying specific consistency properties in decisions. The fundamental theorem of decision theory states that rational decision-makers act *as if* they (1) assign *utilities* to the possible outcomes of actions, (2) assign *probabilities* to the outcomes that might occur, and (3) choose an action that *maximizes expected utility.* The theorem follows from the consistency properties, which are known as the axioms of utility theory.

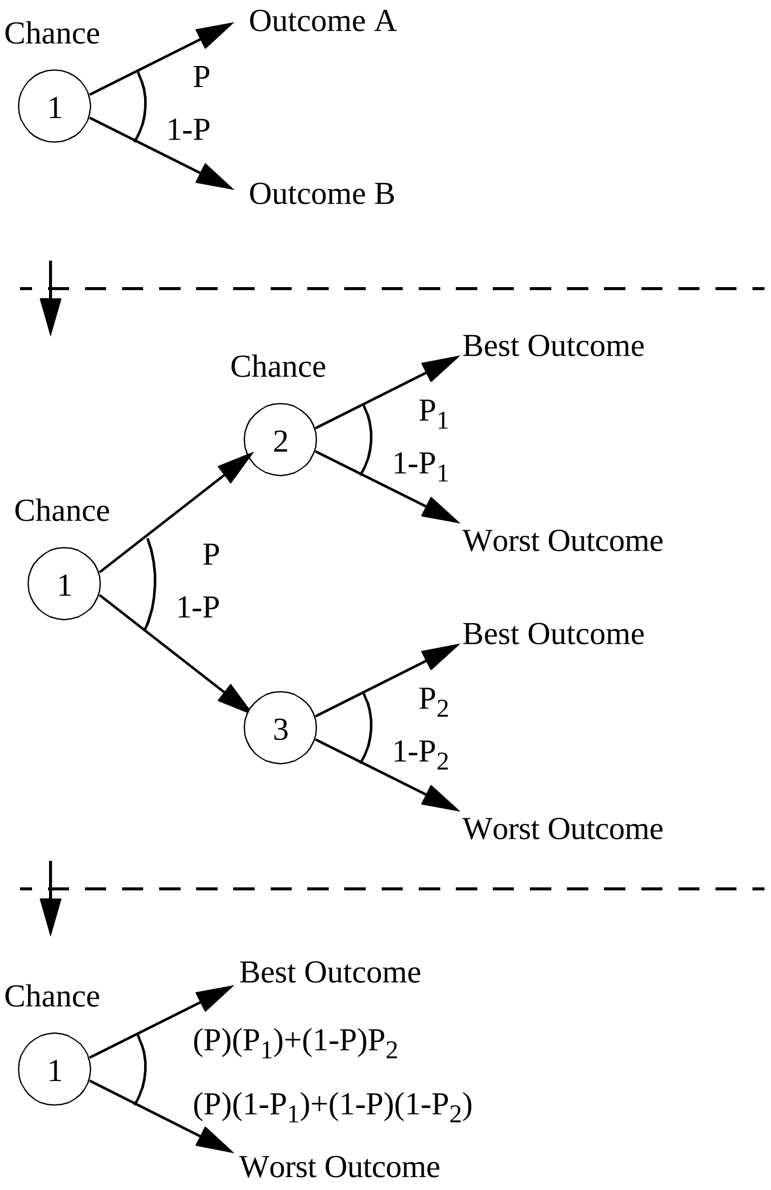

Figure 4-2. Using Utility Functions to Compare Gambles

The top frame illustrates a gamble, which we must compare to another gamble in order to make a decision. In the middle frame, we have replaced Outcome A and Outcome B by equivalent gambles involving the Best and Worst Outcomes. The distinction between Outcomes A and B is encoded in the preference probabilities $U(A)=P_1$ and $U(B)=P_2$. In the bottom frame, we have algebraically simplified the resulting gamble. This reduces the original gamble to a preference probability (in this case $P•(P_1)+(1-P)•P_2$), whose form indicates that it can also be calculated as the *expected utility* for the original gamble $(P•U(A)+(1-P)•U(B))$.

If heuristic functions satisfied the requirements of utility functions, we would be done. But they do not, and can not, for three reasons. The first is that heuristic functions are encoded in applications, and do not change from user to user: but utility functions are user-specific. The second is that multiple heuristic functions might be useful, but the utility axioms require that they be combined into a single function. The third is that heuristic functions commonly confound control information (search this subtree first) with preference information (solutions with this property are better) in a way that unnecessarily complicates the task of designing “good” heuristic functions (although it is possible to resolve this complication by deriving control information through decision analysis and learning about the heuristic function’s behavior [116]).

Because of these difficulties, our research approach in the Bayesian Problem-Solver project [52, 53, 54, 56, 83] has been to treat heuristic functions and so-called objective functions as observed data—statistical evidence—that *can* be used to condition estimates of utility. In other words, algorithms use the information provided by heuristic functions to estimate expected utility. This places no restrictions *per se* on a designer of heuristic functions, but I conjecture that the most informative heuristic functions are those that accurately estimate a single utility attribute (such as computation time, schedule cost, or value of scheduled tasks). Such functions are easily calibrated and combined, both with other heuristic functions and with varying user preferences.

In the Bayesian Problem-Solver approach, search is merely a mechanism for gathering data to use in calculating expected utilities for a decision problem. The search tree—its structure and heuristic function values—provides raw observation data that is used as conditioning information.

If the reader remembers nothing else from this chapter, I hope that the Bayesian Problem-Solver project's metaphor of search as decision-making remains after the engineering details have faded from memory.

4.4. Applying Decision Theory to Search

The question addressed by this chapter is how to implement a Bayesian Problem-Solver for scheduling problems: in other words, how to make rational decisions within a heuristic search algorithm suited for scheduling search spaces. As described above, the axioms of decision theory lead to the result that a *rational* decision-maker, i.e. one who is consistent with the axioms, will choose the gamble with highest expected utility.

To ground this discussion, I will analyze a canonical decision situation in heuristic search: where should we direct our search.

But first, we must consider what kind of utility function is involved in these analyses.

4.4.1. The Utility of Computation Time

In a previous section, I noted that to terminate search early, or set an aspiration level for branch-and-bound, is to gamble that more investment in search time would not yield a better solution. But that seems like a contrived example. If we have a current-best-solution in hand with expected utility *u*, then further search will not take it away.* Therefore, where is the “gamble”? It would seem that further search is *always* useful: it can *only* improve upon the solution we have in hand.

But in fact, the passage of time *degrades* the value of a solution. If this were not true, then users would be satisfied with a scheduling system that promised a dazzling solution after an arbitrarily long computation: but they are not satisfied with that. A more formal argument goes as follows: a solution g_1 at time t cannot be *less* desirable than the same solution g_1 at time t+δ, as the former can be converted to the latter when a no-cost option of delaying is exercised. In most natural cases, g_1 at time t will be preferred to g_1 at time t+δ because of the opportunity cost of delays. The value of most solutions degrades over time.

AI research on modeling the utility of computation time in problem-solving is a recent, but quite diverse enterprise. Work on anytime algorithms is the best known [21, 136], but decision analysis has also informed the field more directly, in the work of Harada [59], Horvitz [63], Russell & Subramanian[114] and Russell & Wefald [116]. These researchers have mapped out basic modeling issues related to fixed deadlines, uncertain but absolute deadlines, etc.

In practice, the utility of time is likely to also involve a number of other issues: one is the distinction between rapid prototyping and a fielded application. When rapid prototyping the model for a problem, I would prefer to know *quickly* that the problem is soluble *at all*, because of the diagnostic value of this information when correcting a misspecified or

* Of course, further search may reveal problems with our current best solution, thus lowering its expected utility. Chess programs, for example, alter their assessment of “good” moves and “bad” moves as new variations are uncovered by deeper search. But our current expected utility for the solution should include our own estimates of these risks. For example, it should not be the case that we currently expect that our future estimate of an action’s expected utility will only decrease with further search (ignoring the cost of time).

incomplete model. So poor solutions achieved in short order will be more valuable during system prototyping than as the output of a fielded application. After the problem specification is complete, and the system fielded for use in an application, the time utility function might be changed radically. A similar argument could be made for the demands on a programming-language compiler with a slow code-optimizer: during rapid-prototyping and debugging, compilation should be fast but the resulting executable programs need not be, whereas during performance testing and delivery, one wants fast programs and can afford slow compilation.

There is much to be done in understanding preferences for computation time, and it seems best to begin with simple utility functions. For simplicity, I follow Russell and Wefald [116] in assuming that the utility function is separable into two components $U[s] = U_a [a(s)] + U_t [t(s)]$, one for the time-independent evaluation of the solution based on its utility attributes $a(s)$, and one for the time $t(s)$ at which solution s is provided as output by the system. This assumption implies that time and the other attributes exhibit *mutual preferential independence* for the decision-maker [30, p. 579]: for example, always preferring less time to more, no matter what the other solution attribute values are. A stronger condition is necessary and sufficient for separability: *mutual utility independence* [30, p. 580].

To understand the mutual utility independence property, it is useful to realize that any uncertain gamble on solution attributes A_1 and A_2 is completely summarized by the joint probability distribution, p, over A_1 and A_2. Call the corresponding marginal distributions $p(A_1)$ and $p(A_2)$. If the two attributes are mutually utility independent, then if I prefer a gamble involving p_x to one involving p_y, such that $p_x(A_2) = p_y(A_2)$, then I will also prefer a gamble involving p_j to one involving p_k, as long as $p_x(A_1) = p_j(A_1)$ and $p_y(A_1) = p_k(A_1)$ and $p_j(A_2) = p_k(A_2)$. In other words, my risk aversion for attribute A_1 is not influenced by the value of A_2.

This is not always the case, of course: if A_1 and A_2 are the satisfaction of two customers, then the importance of my treatment of one customer may change if the other is gravely dissatisfied (particularly if I have no other customers). As should be apparent, the problem here might be resolved by the addition of extra attributes (in this case, perhaps the sum of customers' satisfaction). In many cases, the property of mutual utility independence does not hold, or does not hold without such additional attribute assessments. But as it is a common working assumption for decision analysts, and the focus of this dissertation is not utility assessment and modeling, I will adopt the assumption from now on.

I have modeled deadline situations using two types of *exponential utility function* (pictured in Figure 4-3). In the *risk-averse* case, utility prior to the deadline declines gradually as time increases, then declines sharply at the deadline (the first derivative is negative and the absolute value of the first derivative is increasing). In the *risk-prone* case, utility declines sharply as time increases beyond zero, and then the decline tapers off (the first derivative is negative and the absolute value of the first derivative is decreasing).

In the risk-prone case, the only hope for a high utility result is to find a solution quickly, and so we would expect that a utility-driven search algorithm would search small subtrees first. In the risk-averse case, the important point is to not exceed the deadline, and so search should be ordered more conservatively. Another function of interest, which will not be discussed in the dissertation, is the sigmoid function: risk-averse prior to the deadline, and then risk-prone to get as close to the deadline as possible. (The basic sigmoid shape can be achieved by adding two exponential functions, one which is risk-averse and another which is risk-prone.)

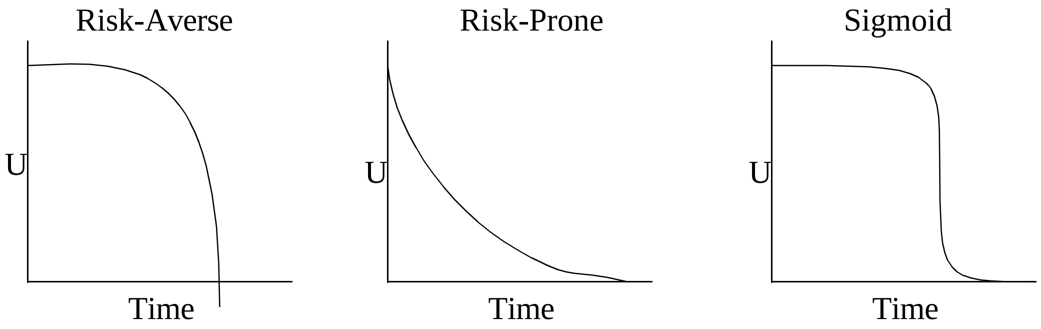

Figure 4-3. Utility Functions for Computation Time.

4.5. The Search Ordering Decision

I begin the analysis of the search ordering decision by describing a related problem: test-sequencing.

In its simplest form, test-sequencing [88, 121] concerns the optimal ordering of a sequence of *n* tests in order to find a fault in a system, composed of *n* subsystems. In the fault-diagnosis literature, test-sequencing is sometimes called troubleshooting. It is typically assumed that there may be more than one fault, but we are interested in finding the first fault as rapidly as possible. It is also assumed that all tests are independent and conclusive. The only difficulty is in the varying costs of the tests, and the fault probabilities for each subsystem. The problem is to order the tests so as to minimize the expected time to uncover the fault.

This problem is intriguingly similar to common problems in heuristic search. For example, consider the problem of ordering subtrees in a depth-first search, where the search stops after finding the first solution that satisfies some condition. In this case, each test is a subtree search, the faults are the set of acceptable solutions, and the problem is to find the solution with a search order that has lowest expected cost. Note that this process does not find the best solution (by some utility measure), but merely any solution that satisfies some predetermined “aspiration level” of utility: this is often called satisficing search.

Consider a simple search problem. Several subtrees (*A*,*B*,*C*,...) can be searched to find a solution. I will focus on the pairwise ordering question: is it better to search *A* before *B*, or *vice versa*? I will assume a linear utility function for computation time. Because we have an aspiration-level utility model, I will assume that the utility of each solution is equal. By this utility model, we are concerned with finding a solution, if it exists, in minimal expected time.

The decision tree for this problem is illustrated in Figure 4-4. For simplicity, I assume that the search cost (C(*A*), C(*B*), etc.) for each tree is fixed. The solution probabilities within each subtree are *not* assumed to be independent.

As illustrated in the figure, the choice is between *ABZ* and *BAZ*, where *Z* indicates the “prospect” we are faced with if *A* and *B* have both been searched unsuccessfully (the prospect would typically be a choice between further search or termination). The fact that this

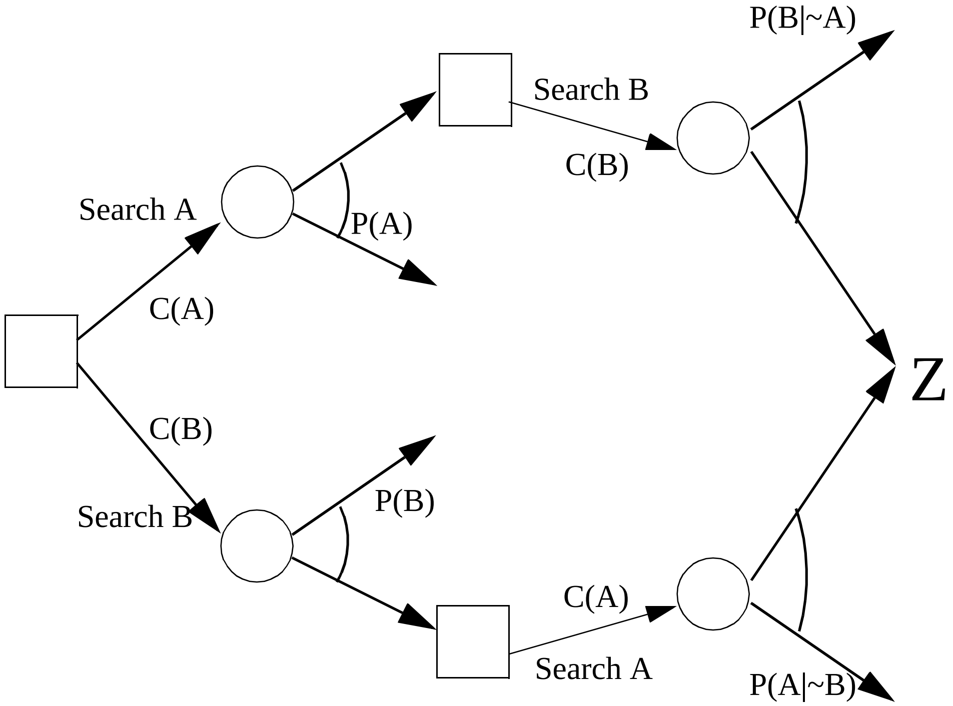

Figure 4-4. Pairwise Search Ordering Problem.

Square nodes indicate choice nodes (searching subtree A or B), and round nodes indicate chance nodes (whether each subtree contains a solution). The subtree labeled Z indicates the "prospect" we face if we search both A and B and have not yet found a solution. The fact that Z is shared by both subtrees is the key to the analysis: we do not need to compute the expected utility of the prospect Z directly.

prospect is identical (i.e., is a shared subtree in the decision tree) could greatly simplify the decision analysis, if we can avoid evaluating *EU(Z)*, the expected utility of the prospect. This happens to be the case, as the following analysis shows.

Consider the expected utility of the two branches of the initial decision:

$$\begin{aligned} EU(ABZ) &= C(A) + P(\neg A)C(B) + P(\neg A, \neg B)EU(Z) \\ EU(BAZ) &= C(B) + P(\neg B)C(A) + P(\neg A, \neg B)EU(Z) \end{aligned}$$

Now subtract to compare the two expected utilities:

$$EU(ABZ) - EU(BAZ) = C(A) + P(\neg A)C(B) - C(B) - P(\neg B)C(A)$$

This difference simplifies to $P(B)C(A) - P(A)C(B)$, which is greater than zero iff $P(B)C(A) > P(A)C(B)$, or equivalently, iff $P(B) / C(B) > P(A) / C(A)$.

Thus we can order search by sorting the subtrees *i* in increasing order based on the measure P(*i*) / C(*i*), and searching the subtrees in that order. (Note that *C*(*i*) is negative.) This result is recounted throughout the test-sequencing literature [88, 121].

4.5.1. Test-Sequencing with Deadlines

But *C(B)* and *C(A)*, the utility penalty of searching *B* and *A*, may depend on elapsed time. Assuming that they do not amounts to assuming a linear utility function for computation time. However, we are often in deadline situations, where utility for time changes dramatically near the deadline.

I will analyze the case of exponential utility functions (Figure 4-3) for the problem of ordering two subtrees. Recall that we are assuming a time-separable utility function over the solution *s* provided by the search algorithm: *U[s] = U[a(s)]-U[t(s)]*. If we find a solution *s* after searching only subtree A, then *U[t(s)] = U[C(A)]*.

For simplicity, assume that there is no uncertainty over C(B) and C(A). This assumption is unrealistic, but permits us to understand the desired BPS behavior under risk-averse and risk-prone utility functions.

Again we begin by writing the expected utility for each ordering.

$$\begin{aligned} EU(ABZ) &= P(A)U(C(A)) + P(\neg A, B)U(C(AB)) + P(\neg A, \neg B)EU(Z,\mathrm{C(AB)}) \\ EU(BAZ) &= P(B)U(C(B)) + P(\neg B, A)U(C(BA)) + P(\neg B, \neg A)EU(Z,\mathrm{C(BA)}) \end{aligned}$$

We use the equalities $P(\neg A, B) = P(B) - P(A, B)$, $P(\neg B, A) = P(A) - P(A, B)$, and $C(AB) = C(BA)$ and then subtract in order to compare the expected utilities.

$$\begin{aligned} & EU(ABZ) - EU(BAZ) \\ &= P(A)U(C(A)) + P(\neg A, B)U(C(AB)) \\ &\quad -P(B)U(C(B)) - P(\neg B, A)U(C(AB)) \\ &= P(A)U(C(A)) + (P(B) - P(A, B))U(C(AB)) \\ &\quad -P(B)U(C(B)) - (P(A) - P(A, B))U(C(AB)) \\ &= P(B)[U(C(AB)) - U(C(B))] - P(A)[U(C(AB)) - U(C(A))] \end{aligned}$$

Now we consider the form of the exponential utility function.

$$\begin{aligned} U(x) &= \beta e^{x/\lambda} \\ C(AB) &= C(A) + C(B) \\ U(x+y) - U(x) &= \beta e^{(x+y)/\lambda} - \beta e^{x/\lambda} \\ &= \beta e^{x/\lambda}[e^{y/\lambda} - 1] \end{aligned}$$

Substituting in the formula for EU(ABZ) - EU(BAZ) produces a simple subtree-ordering formula.

$$\begin{aligned} & EU(ABZ) - EU(BAZ) \\ &= \beta[P(B)e^{C(B)/\lambda}(e^{C(A)/\lambda} - 1) - P(A)e^{C(A)/\lambda}(e^{C(B)/\lambda} - 1)] \\ &= \beta\left[P(B)\frac{e^{C(B)/\lambda}}{e^{C(B)/\lambda} - 1} - P(A)\frac{e^{C(A)/\lambda}}{e^{C(A)/\lambda} - 1}\right] \\ &= \frac{\beta P(B)}{1 - e^{-C(B)/\lambda}} - \frac{\beta P(A)}{1 - e^{-C(A)/\lambda}} \\ &= f(B, \beta, \lambda) - f(A, \beta, \lambda) \end{aligned}$$

This implies that the two subtrees, *A* and *B*, can be ordered by comparing a simple function of each subtree: thus a list of subtrees can be ordered by sorting them according to this function.

To illustrate the intuition behind this equation, consider a set of subtrees with *N* different probabilities and *N* different search costs. As shown in Table 4-2, the optimal ordering varies as the parameters λ and β change (only the sign of β is important). The table uses two utility functions (risk-prone and risk-averse, as illustrated in Table 4-1), and then applies the ordering function (derived above) to rank subtrees whose probability and search time range from 0 to 1. As expected, zero-probability subtrees rank lowest for either utility function. Subtree order increases as probability increases or time decreases.

The italicized values (along the main diagonal of the bottom two tables in Table 4-2) illustrate the difference between risk-averseness and risk-proneness. Generally speaking, a risk-prone decision-maker will search small subtrees despite a low probability of success (upper-left italicized values in the top table), in the hopes of finding a fast solution. A risk-averse decision-maker will instead search large, high-probability subtrees (lower-right italicized values in bottom table) before small, low-probability subtrees. Note that a riskneutral decision-maker would be indifferent between the subtrees represented by the main diagonal, as probability/cost is equal along the diagonal.

Utility Functions			Time						
	β	λ	0	.1	.3	.5	.7	.9	1.0
Risk-Prone	-20	-3	1.00	.73	.38	.18	.08	.02	0
Risk-Averse	1	4	1.00	.99	.96	.88	.71	.34	0

Table 4-1. Risk-Prone and Risk-Averse Utility Functions.

This table presents scaled utility values for two exponential utility functions. Note that the risk-prone function reflects a sharp drop in utility as time increases, even from 0 to .1.

Risk-Prone Subtree Ordering

		Time						
		.01	.1	.3	.5	.7	.9	1.0
Probability	0	0	0	0	0	0	0	0
	.1	66	*5.7*	1.4	.6	.3	.1	.1
	.3	197	17	*4.1*	1.7	.8	.4	.3
	.5	328	29	6.9	*2.9*	1.4	.7	.5
	.7	460	40	9.6	4.0	*2.0*	1.0	.7
	.9	591	51	12	5.2	2.5	*1.3*	.9
	1.0	657	57	14	5.7	2.8	1.4	*1.1*

Risk-Averse Subtree Ordering

		Time						
		.01	.1	.3	.5	.7	.9	1.0
Probability	0	0	0	0	0	0	0	0
	.1	2.6	*.30*	.14	.12	.11	.10	.10
	.3	7.7	.91	*.43*	.35	.32	.31	.31
	.5	13	1.5	.72	*.58*	.53	.51	.51
	.7	18	2.1	1.0	.81	*.75*	.72	.71
	.9	23	2.7	1.3	1.0	.96	*.93*	.92
	1.0	26	3.0	1.4	1.2	1.1	1.0	*1.0*

Table 4-2. Examples for Test-Sequencing with Deadlines.

Each cell in these two tables represents the subtree ordering “priority” for a given search time and probability. Higher-numbered cells indicate subtrees which would be searched first.

4.6. Evaluation

I have tested the effectiveness of the BPS search-ordering techniques using POSIT, a highly-optimized propositional satisfiability tester developed by Freeman [48, 49], and based on the DPL algorithm (the Davis-Putnam algorithm [37] in Loveland's form, as described by Davis, Logemann and Loveland [36]). POSIT provides a simple validation of BPS search-ordering in the context of a realistic search-based problem-solver. Because POSIT relies on backtracking search and a specialized form of constraint satisfaction, it reflects potential advantages of using BPS in scheduling applications. In these experiments, I assume a linear utility function for computation time. I also describe earlier experiments I performed on a scheduling application [56].

4.6.1. Search Ordering in POSIT

Traditionally, constraint-satisfaction solvers rely on a variable-ordering and a value-ordering heuristic. The variable-ordering heuristic determines which variable will be instantiated next. The value-ordering heuristic determines the sequence in which to search the possible values for that variable.

In the case of POSIT, variable-ordering corresponds to choosing a propositional variable or "premise" *p*. Value-ordering corresponds to deciding whether to instantiate *p* or ~*p* first. If the problem is unsatisfiable, the choice is unimportant, because both the *p* and ~*p* subtrees (which I call the positive and negative subtrees) much be searched to verify that there is no satisfying assignment. But if there is a satisfying variable assignment, the value-ordering choice can have an effect on performance, if the first subtree searched contains a satisfying assignment.

Many different value-ordering heuristics have been proposed for the DPL algorithm which POSIT implements. Freeman [48, p. 82] gives a brief summary of these. Some researchers always choose the positive subtree first, and Freeman notes that Crawford and Auton [34] did so “only because they could not convince themselves that the branching order should make a difference.”

Other researchers have used a few basic features in constructing heuristics for value-ordering. The most common feature is a count of how many times *p* and ~*p* occur in the shortest unsatisfied clauses in the formula (these counts are known as *pos_cost* and *neg_cost*). If *p* occurs more often, POSIT chooses to instantiate *p* first (i.e., searches the positive subtree first). Another feature that is used by other SAT testers is whether or not there are any unsatisfied binary clauses in the formula (this feature is known as *bin*).

Dubois et al. [44] used a variation on this. If the formula is believed to be satisfiable, they used a version of their system (S-SAT, based on an incomplete search algorithm) which would first choose *p* iff *p* satisfied more clauses than ~*p* satisfied. Otherwise, they use a second version of the system (C-SAT), which uses the *opposite* ordering. Dubois et al. determine whether a formula is believed to be satisfiable based on the well-known “hard problems” threshold for random SAT problems [29].

4.6.2. Search Ordering using BPS

To provide a simple demonstration of BPS search ordering, I applied it only to the top level value-ordering decision in POSIT. Every subsequent value-ordering decision is made using POSIT’s built-in heuristics. I gathered data for several benchmark problem sets, solving 10000 problems for each one (this is broken in two equal halves: a training set and a test set).

The essence of the BPS approach is to estimate utility attributes from heuristic values. Specifically, the training data was used to construct estimates of P(*satisfiable* | *heuristics*) and E[*cost* | *heuristics*] for both the positive and negative subtrees. A simple uniform bucketing estimator was constructed over 3 existing POSIT heuristic features (*bin*, *pos_cost*, *neg_cost*). Each training problem contributes to the estimates in a single bucket, and the sample means for *cost* and *satisfiable* are used as the estimates. Where a bucket contained less than 50 samples, BPS reverts to random value-ordering.

I compare BPS search ordering to five other heuristics.

- POSIT: the built-in value-ordering heuristic.
- NEGAT, which chooses the opposite of POSIT's built-in choice.
- RANDOM, which chooses the positive subtree with probability 1/2.
- OMNISCIENT, an artificial algorithm which always makes the ideal ordering choice, given the available heuristic information. (OMNISCIENT is the perfect "table-lookup" algorithm, given a table indexed by the available heuristics. For each cell in the table, OMNISCIENT tabulates expected time over the training set for the two possible ordering choices. When run on the test set, OMNISCIENT chooses the ordering which minimizes expected cost on the corresponding cell of training set data.)
- ADVERSARY, an artificial algorithm which chooses the opposite of OMNISCIENT's choice, i.e., the worst possible ordering given the available heuristic information.

Table 4-3 summarizes the mean search time results for these 6 heuristics on two families of standard randomized benchmarks. The Hard Random K-SAT benchmark is due to Mitchell, Selman and Levesque [87]. Each K-SAT problem is generated according to three parameters (clause length, proposition count, clause count). For each clause, the member propositions are chosen at random, and each is negated with probability 1/2.

The graph coloring benchmark is part of the POSIT software distribution. Each graph-coloring problem is generated according to three parameters (vertex count, edge probability, color count). The first two parameters specify a random graph, and the third provides the number of legal colors. The SAT encoding uses a propositional variable for each vertex/ color pair.

Because we are only analyzing the top-level value-ordering decision in the search tree, the differences from Random value-ordering are small and measured in single percentage points, as we see in the table. Nonetheless, the BPS value-ordering is within 1% of optimal (OMNISCIENT) in all but one case. In contrast, the POSIT and NEGAT heuristics are each beneficial in *only one* of the two benchmark families. Freeman himself was puzzled by the odd behavior of the POSIT heuristic on Hard Random K-SAT problems:

> My experiments on a wide range of benchmark problems overwhelmingly supported the conclusion that *p* should first receive the truth value which *falsifies* as many of its associated literals as possible. But my experiments on hard random 3-SAT problems indicated that *p* should receive the truth value which *satisfies* as many of these literals as possible. (I have no explanation for this phenomenon.) [48, p. 55]

I suggest that the BPS search ordering analysis offers the explanation for this phenomenon. We can study the problem with the POSIT heuristic by analyzing two of the BPS ordering decisions. For the GCOL-13 benchmark, where the POSIT heuristic does well, its preference matches that of the BPS heuristic, if we train BPS on the single binary feature *pos_cost* > *neg_cost* (this is the binary feature used in POSIT’s value-ordering). When POSIT prefers the positive subtree, BPS scores the negative subtree as (p/c)=(.444/ 2.1)=.2, while the positive subtree is scored as (p/c)=(.445/1.0)=.4. The positive and nega-

		Random	Omniscient	BPS	Posit	Negat	Adversary
Benchmark (parameters)	% Satisfiable	(ms)	(as percentage of Random search time)				
ksat-6 *((3,125,536)*	48%	40.6	96.7%	97.2%	*102.5%*	97.6%	*103.4%*
ksat-7 *(3,150,642)*	48%	112.6	96.6%	97.1%	*102.5%*	97.5%	*103.3%*
ksat-8 *(3,175,748)*	48%	304.0	96.3%	96.7%	*103.3%*	96.8%	*103.7%*
ksat-9 *(3,200,854)*	49%	921.0	94.9%	95.8%	*104.8%*	95.1%	*105.0%*
gcol-10 *(20,.5,4)*	32%	39.9	98.7%	99.8%	99.8%	*100.3%*	*101.3%*
gcol-11 *(30,.25,4)*	48%	9.6	96.5%	96.6%	96.9%	*103.3%*	*103.6%*
gcol-12 *(40,.2,4)*	36%	24.5	98.0%	98.6%	98.5%	*101.5%*	*102.0%*
gcol-13 *(15,.45,4)*	44%	2.5	95.5%	96.3%	96.4%	*103.6%*	*104.5%*

Table 4-3. POSIT and BPS Results: Search Ordering.

This table presents search times for 8 benchmark problem sets for 6 value-ordering heuristics applied to the POSIT SAT tester. Average search time over 5000 problems is given for the Random heuristic (in milliseconds). Search time for the remaining 5 algorithms is given as a percentage of the Random search time. Italicized percentages indicate value-ordering results that are worse than Random choice for the given benchmark problem set.

tive subtrees have almost equal chance of containing solutions, but the negative subtree is over twice as large on average.

However, in the KSAT-9 benchmark, POSIT does poorly. When the POSIT heuristic prefers the positive subtree, BPS scores the negative subtree as (probability/cost) = (0.407/572) = .0007, while the positive subtree is scored as (p/c) = (.281/505)=.0006. Here the tradeoff is more subtle, but according to BPS, the decrease in size of the positive subtree (505 ms compared to 572 ms) is not worth the reduction in the odds of finding a solution (.281 compared to .407).

To see how these top-level decision improvements “expand” when the different heuristics are applied to each ordering decision in the search, I re-tested the KSAT-9 benchmark using four algorithms. A version of POSIT with the random-choice heuristic produced an average search time (over 5000 problems) of 928 milliseconds. The NEGAT heuristic led to a 9% reduction in search time, to 842 ms. The BPS heuristic produces an 8% reduction, to 857 ms. But as expected, the POSIT heuristic was 1% worse than random, at 936 ms/problem.

4.6.3. Earlier Experiments

I have also experimented with BPS in scheduling and other constraint-satisfaction problems (some very early work is described in a 1994 paper [56]).

In these early experiments, I saw that one advantage of the BPS approach is that multiple heuristics can be combined to form single estimates of probability (P) and search cost (C). Artificial intelligence techniques have never offered powerful methods for combining heuristics. For example, in branch-and-bound algorithms, the most common approach is to take the maximum of different admissible heuristics (admissible heuristics are guaranteed to underestimate the actual solution cost). In the POSIT work, I combine two heuristic features (*pos_cost* and *neg_cost*), and have experimented with other heuristics proposed for SAT problems.

Combining heuristics is important because individual heuristics may provide adequate estimates of cost (C) or satisfiability (P) alone. For example, the "most constraining variable" heuristic is a measure of subtree size and search cost: a variable which heavily constrains unassigned variables should generate a smaller search tree. The "least constraining value" heuristic, on the other hand, preserves as many solutions as possible: a value with few constraints on other variables should maintain the probability that a solution will be found.

Furthermore, the relationship between heuristic values and estimates of P or C may be subtle. For example, the Minimum Domain heuristic suggests choosing the variable with the minimum number of remaining values. But experimental results suggested that Minimum Domain should really be broken into two cases: variables with a single remaining legal value should be selected (this is an instance of the general "unit propagation" or "unit resolution" or "forced move" heuristic seen in other search problems), but otherwise, larger domains were preferred in at least one benchmark. The performance advantage of unit propagation may be so great that it skews the interpretation of the heuristic overall.

4.7. Related Work

Decision theory's founders include Ramsey [109], von Neumann and Morgenstern [127], Wald [130], Cox [33], de Finetti [40] and Savage [118]. For more information on the current state-of-the-art, see the excellent texts by Clemens [30], Keeney and Raiffa [70], Raiffa [108], von Winterfeldt and Edwards [128], or Watson and Buede [132], or the shorter articles by Henrion *et al*. [64], or Howard [65, 66].

Many researchers have applied decision theory and probability to heuristic search. Abramson [5] devised a heuristic for two-player games using an assumption of random-play beyond the search frontier: he trained a leaf-node heuristic estimator to predict the result of the game given random-play from that leaf. Baum and Smith [11] have developed the "best play for imperfect players" approach, which uses probability to model the imperfect decision-making of players (in direct contrast to Minimax), and to model the fact that more information will be available by the time that a searched game-tree node has become the current node. Boyan [25] applied predictive statistical techniques to stochastic search algorithms: his STAGE algorithm uses reinforcement learning to predict the performance of a hillclimbing search, and uses this information to find good starting points for hillclimbing. Mayer's dissertation [83] evaluated the BPS approach on single-agent problems such as the Eight Puzzle. Palay [99] developed a probabilistic version of Berliner's B* algorithm for two-player algorithms: his PB* algorithm represents and manipulates probability distributions over game position evaluations between an upper and lower bound provided by heuristic estimates. Rather than using the strict bounds to choose actions, PB* can commit to a move based on the probability that it is better than other moves. Finally, Russell and Wefald [116] developed an extensive framework for metareasoning in search control: they developed and evaluated algorithms for game-playing and problem-solving search.

Beyond the problem of search control, many other AI researchers have drawn on decision theory in the past decade: a good portion of this work can be found in the proceedings of the annual conference on Uncertainty in Artificial Intelligence. One particularly related topic is the efficient approximation of decision-theoretic reasoning, although most of that work is focused on probabilities alone, and not expected utilities.

Others in AI have used test-sequencing concepts to control problem-solving, particularly diagnosis (Heckerman et al. [60]), but also search (Simon and Kadane [121]).

4.8. Summary

I have applied the Bayesian Problem-Solving approach to the search-ordering problem in backtracking search for solving scheduling and constraint-satisfaction. I have shown how test-sequencing results fit into the BPS approach, with both linear and exponential utility functions for computation time.

I have implemented the BPS search-ordering approach to control POSIT, a state-of-the-art propositional satisfiability tester [48, 49], and demonstrated the behavior and effectiveness of BPS search ordering in this setting. By estimating two utility attributes (probability that a solution exists, and computation time) based on the evidence provided by heuristic functions, BPS is able to overcome a domain-specific performance anomaly in the built-in POSIT heuristic.

5 Summary and Future Directions

5.1. Summary

This dissertation has described several advances to the theory and practice of artificial intelligence scheduling techniques, originally developed and implemented during the construction of DTS, the Decision-Theoretic Scheduler, and its successor, SchedKit, a toolkit of scheduling algorithms and data structures.

The dissertation's three main chapters have focused on improving the efficiency of scheduling systems by reducing the three terms in a stylized "problem-solving cost equation":

$$\frac{\text{seconds}}{\text{problem}} = \frac{\text{seconds}}{\text{state expansion}} \cdot \frac{\text{state expansions}}{\text{state space size}} \cdot \frac{\text{state space size}}{\text{problem}}$$

- The first term is "seconds/state expansion", a measure of search cost per state expanded: in Chapter Three, I presented techniques for formalizing incremental computation of heuristics and constraints in a scheduling search system.
- The second term is "state expansions/state space size," a measure of the effectiveness of selective search control: in Chapter Four, I have applied the Bayesian Problem-Solver approach to derive a search ordering heuristic for backtracking search.
- The third term, "state space size/problem," is a broad characterization of the goal of problem formulation and preprocessing: in Chapter Two, I presented two techniques that preprocess conjunctions of disjunctive resource constraints in order to improve constraint tightness prior to search.

Chapter Two describes my research on *reducing the size of the scheduling state space*: I have developed two new preprocessing algorithms, MPC1 and MPC2, designed to exploit resource constraints and resource capacity prior to search.

- MPC1 extends the familiar critical path method to incorporate capacity constraints. The output of MPC1 is a set of tighter time constraints between pairs of events in the scheduling problem, based on the resource capacity requirements of tasks that occur between each event pair.
- Experimental results with MPC1 show that it is effective on its own, but when combined with existing temporal constraint processing algorithms, further improves their effectiveness in reducing the size of the search tree. When the output of these preprocessors is fed into an off-the-shelf constraint solver (clp(FD)), MPC1 is shown to provide a net factor of two improvement even on small problems, where the cost of preprocessing is relatively higher.
- MPC2 reasons about the aggregate resource constraints of subprojects (groups of tightly constrained tasks). Such subprojects are a common feature of NASA scheduling problems, where they represent the time-phased experiments or observations requested by an individual space scientist. The output of MPC2 is a set of additional ordering constraints at the level of subprojects, based on lower bounds of resource requirements and an analysis of possible overlap between pairs of subprojects.
- The MPC2 algorithm uses the interval-tree data structures described in Chapter Three. For example, MPC2 requires an $O(N^2 a(N) log(N))$ step[1] of constructing an "intermediate resource profile" data structure for each project, where N is the number of tasks in each subproject. This profile is added to another profile in time $O(N^2 a(N)\ log(N a(N)))$, which is the time complexity of checking for an MPC2 ordering constraint between two subprojects with N tasks each.

[1] $a(N)$ is the extremely slowly growing inverse of Ackerman's function.

- Experimental results with MPC2 show that it is very likely to discover additional ordering constraints even with moderate resource utilization. When the output of MPC2 is fed into an off-the-shelf constraint solver (clp(FD)), MPC2 provides a net speedup of between 2 and 10 times that of temporal preprocessing alone.
- I described a number of other preprocessing algorithms inspired by the basic MPC approach.
- Disjunction is generally a source of complexity for automated reasoning systems. The MPC techniques described here can be viewed as grouping the disjunctive constraints into bundles that can be treated conjunctively. I expect that future work will reveal better techniques for finding and exploiting tractable conjuncts of disjunctive constraints.

Chapter Three describes my research on *reducing the cost of state generation in scheduling search*: I have introduced the use of data structures that optimize heuristic evaluation, constraint-checking and state-variable maintenance by exploiting incremental computation. The data structures combine technology from computational geometry and attribute grammars.

- One approach to rapid state generation is to incrementally calculate heuristic evaluation functions and constraint checks. I describe how an augmented interval tree (a data structure from the computational geometry literature) is particularly appropriate for computation of scheduling heuristics and constraints. If properly coded, the heuristic value or constraint status can be read off the root of the tree.
- Hand-coded interval tree data structures resulted in significant performance speedups (three orders of magnitude) in early versions of DTS, because of the O(N)/O(log N) speedup due to the data structure.

- The augmented interval trees in DTS can be formally specified using attribute grammars, a tool developed by compiler theorists for formally specifying the parsing of programs. I describe and adapt the attribute grammar formalism for my purposes.
- I present several examples of attribute grammar specifications of heuristics and constraint checks. The specifications can also represent state variables (e.g., battery power, machine configuration).
- I discuss how the attribute grammar technology might be used to design application-specific schedule editors that check constraints interactively but efficiently.

Finally, Chapter Four describes my research on *reducing the number of states examined during search*. Continuing my earlier research on using decision theory to control search, I have applied the Bayesian Problem-Solver (BPS) approach to control search ordering in a backtracking algorithm.

- For search ordering, I have shown how test-sequencing results fit into the BPS approach, and extended these results to handle exponential utility functions for handling computational deadlines.
- I have implemented the BPS search-ordering heuristic to control POSIT, a state-of-the-art propositional satisfiability tester [48, 49] built upon backtracking search. These experiments demonstrate the effectiveness of BPS search ordering in this setting, although the possible improvement in search ordering is small in these benchmarks.
- By explicitly estimating and combining P(satisfiability | heuristics) and E[search cost | heuristics] from training problems, BPS is able to closely approximate the ideal ordering across eight problem sets drawn from two benchmarks (Hard Random K-SAT and graph-coloring). In contrast, POSIT’s built-in search ordering heuristic fails on half of these problem sets, producing results that are worse than random choice for Hard Random K-SAT.

5.2. Future Directions

5.2.1. Preprocessing Resource Constraints

The results of Chapter Two can be extended in a number of ways. The MPC2 algorithm reasons about the resource requirements of pairs of subprojects in the hopes of finding new ordering constraints implied by their internal temporal and resource constraints. MPC2 could be improved by incorporating additional constraints. For example, a bound on the schedule's "makespan" (schedule length) could tighten the bounds offered by the start-profile, end-profile and intermediate-profile, because inter-task delays in the subproject might have to be compressed. An extension of this would be a test for whether a *set* of subprojects can be completed if they must be scheduled within a specified time interval.

It would also be interesting to extend MPC2 to deal with ordering decisions over groups of subprojects. One possibility is to first consider the minimum delay between the start of two subprojects: if subproject A starts before subproject B, how much delay must there be between the two start times? Given this information, one might then be able to prove that a given subproject *cannot be* or *must be* started before all others, because of these delays (following the example of Carlier and Pinson [*27*] in the job-shop scheduling problem).

5.2.2. State-Generation and Resource Management

Chapter Three introduced an interval-tree data structure and a specification mechanism for designing efficient state generators. There are several challenges in making these techniques more useful for scheduling applications.

The interval tree provides a means for representing state variables (e.g., the charge level of a battery) during scheduling. This would be more practical if we could import existing state-machine models directly into the attribute grammar model. For example, in the NASA DS1 Remote Agent system [94], the most detailed model of spacecraft state transitions exists in the diagnostic system, not in the scheduling system. Reusing these diagnostic models would require a translator from the Remote Agent concurrent transition system models into the simpler models supported by the attribute grammar.

The interval-tree data structures are well-suited to support reasoning about resources in scheduling systems, i.e., to support queries such as the following.

- Are *C* units of resource *R* available at any time?
- What is the earliest time during interval *I* that *C* units of resource *R* are available?
- At what time during interval *I* is resource *R* most overallocated?
- Are there any underflows (overallocations) on resource *R* during interval *I*?
- In what state is resource *R* at time *T*?

These queries are essential in checking constraints, expanding search nodes, and evaluating scheduling heuristics. Queries such as these are used by both constructive (e.g., branch-and-bound) and iterative improvement (e.g., repair-based) scheduling algorithms.

In future work, I hope to extend the existing mechanisms in SchedKit and DTS into a general system for reasoning about resources: a Resource Map Manager. The name borrows from the Time Map Manager developed by Dean and McDermott [38], one of several systems which provides such omnibus support for temporal reasoning. But such systems do not provide general reasoning about resource availability, capacity, and state. A Resource Map Manager would be a valuable, reusable component for use in scheduling systems.

5.2.3. Decision Analytic Search Control

Chapter Four presented a search-ordering heuristic that is based on an explicit decision-theoretic analysis. In earlier research, this Bayesian Problem-Solving approach was applied to limited-time search decisions in single-agent problem-solving [83].

There are two major technical obstacles to the Bayesian Problem-Solving approach:

- BPS has high statistical estimation and computational requirements, because of the underlying costs of exact probabilistic inference [101].
- Reasoning about the expected utility of continued search is difficult because, among other reasons, the decision tree is very large (the sequence of search computations is potentially infinite). Other approaches to decision-theoretic search have used a *myopic* decision-making assumption to address this problem, but this typically underestimates the expected utility of continued search [116].

In light of these obstacles, the search-ordering analysis in Chapter Four is interesting because it evaluates two infinite decision trees by finding a closed form for the *difference* between the expected utilities. The expected utility of searching subtree *A* before subtree *B* is never computed directly (doing so would require a means of dealing with the large subtree that results if neither *A* nor *B* contain a solution).

Decision analysts and others have long known that it is not *necessary* to compute expected utilities directly in order to find the maximum expected utility decision (just as it is not necessary to sort a list in order to find the maximum element). Practicing decision analysts would not be able to carry out or explain their analyses to clients if exact computations were required: only by appropriate bounding and dominance arguments can they find short “proofs” to support the recommended decision. More generally, statisticians have long relied on bounding and approximation techniques to make statistical techniques practical (e.g., see the textbooks by Press [106], Holloway [61], Kennedy and Gentile [73], or Thisted [124]). And the fact that bounds can be used to limit computation has been exploited by previous search algorithms such as B* [14] and the decision-theoretic algorithms of Russell and Wefald [116].

Inspired by these lines of research, my current focus is to develop a decision-theoretic search algorithm that employs bounding and dominance arguments, in the hopes of overcoming the obstacles of myopia and the computational cost of exact probabilistic inference. There is much work to be done before the merits of this approach can be determined, but so far I have identified a number of promising techniques for computing upper and lower bounds on expected utility. These include reordering decision trees, adding or removing choices to decision trees, using upper- and lower-bound integration techniques such as the Riemann rule, and bounding the utility function directly.

I will sketch an analysis of “myopic” reasoning to give a flavor of the use of upper and lower bounds. When using decision theory to decide whether to continue search, one needs to evaluate a potentially long sequence of search computations: the myopic simplification is to consider only the first chunk of computation. This is called the “single-step” assumption in Russell and Wefald’s MGSS* algorithm. What effect does myopic reasoning have on decision-theoretic control of search? The key point is that myopic reasoning is a way of *underestimating* the value of a sequence of computations. Figure 5-1 describes the relation between myopia and lower-bounds. That myopic reasoning produces a lower bound is seen by the fact that it removes choices from the decision tree, which can never increase expected utility

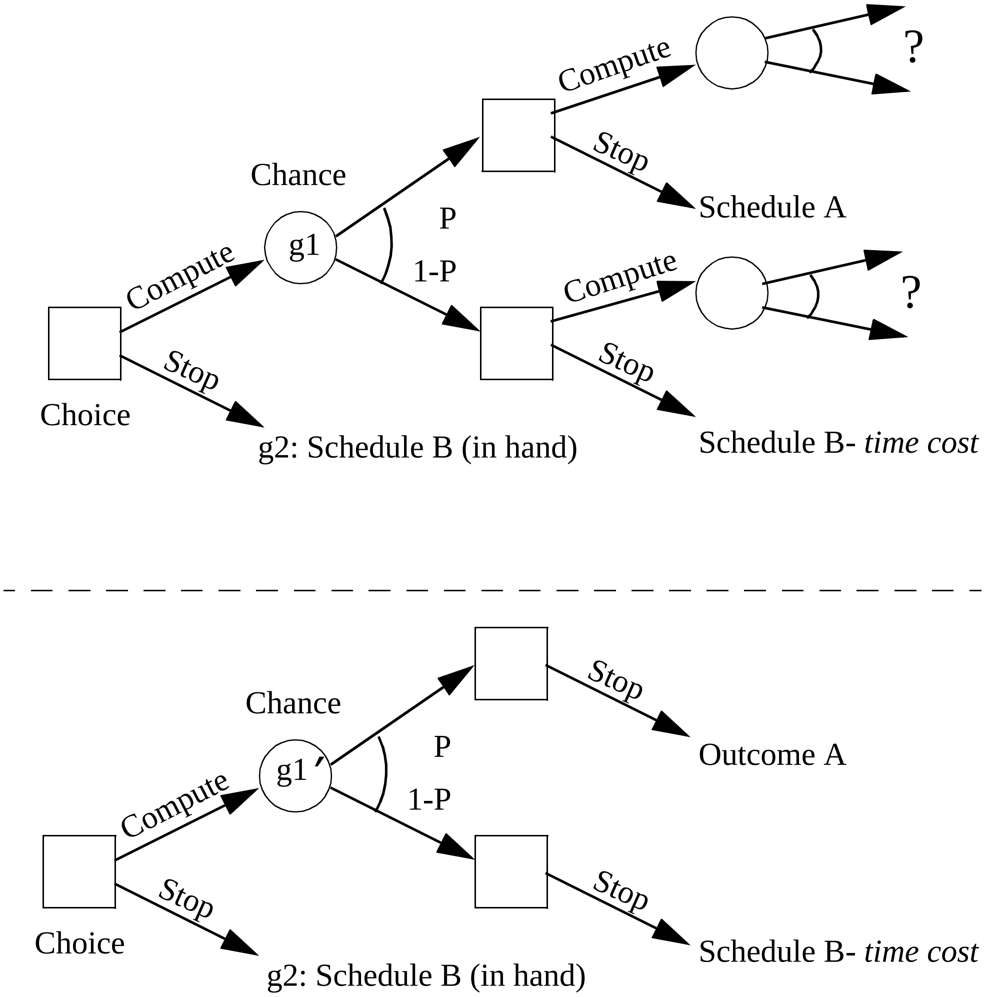

Figure 5-1. Lower-Bound on the EU of an Infinite Computation Sequence.

In this example we have found a feasible schedule B, and are deciding whether to continue searching in the hopes of finding a better schedule A. If we search and fail, we still have schedule B, but with a penalty of the time cost of searching.

The bottom decision tree is the "single-step assumption" simplification of the first, such that gamble g_1 has expected utility at least as great as gamble $g_1{}'$, $\mathrm{EU}(g_1) \geq \mathrm{EU}(g_1{}')$, because choice nodes have been restricted in the second diagram. Thus, if $\mathrm{EU}(g_1{}') \geq \mathrm{EU}(2)$, then $\mathrm{EU}(g_1) \geq \mathrm{EU}(g_2)$. Note that $\mathrm{EU}(g_1{}')$ is easily evaluated, while $\mathrm{EU}(g_1)$ requires the evaluation of the infinite sequences denoted by question marks.

My initial testbed for the bounding approach is the MAXSAT problem. MAXSAT is the generalization of propositional satisfiability where we seek to maximize the number of satisfied clauses. Because of its objective function, the problem is more difficult and computationally intensive for the BPS approach. Searching a subtree in SAT either produces a satisfying assignment or not, and so we need only estimate P[*satisfiable* | *heuristics*]. In MAXSAT, searching a subtree must be analyzed in terms of the probability distribution over possible increases in the number of satisfied clauses. One of the many approaches to bounding the expected utility of searching a subtree is to use an aspiration level *K* for the number of satisfied clauses, and estimate P[*K clauses satisfied* | *heuristics*].

A second line of further work is to use decision analysis to re-analyze the heuristics built into high-performance search algorithms. For example, POSIT uses two heuristics, but "the second, BCP-based, heuristic is much more powerful but too slow to run on every open proposition at every node of the search tree" [48, p. 42]. Accordingly, POSIT's designer established cutoffs to trade off the benefit of the better heuristic against its increased cost. But these cutoffs are fragile: "POSIT's running time is very sensitive to the exact value of *unfailed-lit-limit* elsewhere in the search tree; I have found that a value of 3 is just about right for most problems" [48, p. 52]. It would be challenging to formalize and perhaps improve such *ad hoc* design decisions using an explicit decision analysis.

References

Papers by Othar Hansson can be obtained by sending electronic mail to othar@acm.org. An online version of the bibliography is also available. Some references are available on the Internet: for these, a URL (Uniform Resource Locator) is provided in brackets.

4] E. H. L. Aarts, P. J. M. van Laarhoven, J. K. Lenstra and N. L. J. Ulder.
"A Computational Study of Local Search Algorithms for Job Shop Scheduling."
ORSA Journal on Computing, 6(2):118-125, 1994.

5] Bruce D. Abramson.
The expected-outcome model of two-player games.
Morgan Kaufmann, San Mateo, CA, 1991.

6] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, Reading, MA, 1986.

7] Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin.
Network Flows: Theory, Algorithms, and Applications.
Prentice Hall, Englewood Cliffs, NJ, 1993.

8] Kenneth J. Arrow.
Social Choice and Individual Values.
(Cowles Commission Monograph 12).
Wiley, New York, 1951.

9] Philippe Baptiste and Claude Le Pape.
"Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems."
In *Principles and Practice of Constraint Programming - CP97*, Linz, Austria, 1997.

10] Anthony Barrett and Daniel S. Weld.
"Partial-Order Planning: Evaluating Possible Efficiency Gains."
Technical Report 92-05-01 Expanded Version, Dept. of Computer Science and Engineering, Univ. of Washington, 1993.

11] Eric B. Baum and Warren D. Smith.
"A Bayesian Approach to Relevance in Game Playing."
Artificial Intelligence, 97(1-2):195-242, 1997.

12] Howard A. Beck.
"Constraint Monitoring in TOSCA."
In *Working Papers of AAAI Spring Symposium: Practical Approaches in Planning and Scheduling*, Stanford, CA, March 1992.
Also available from Artificial Intelligence Applications Institute (Edinburgh), Technical Report AIAI-TR-118, December 1992.
(http://www.aiai.ed.ac.uk/~hab/Papers/aaai92.ps)

13] Richard E. Bellman.
"On a routing problem."
Quarterly of Applied Mathematics, 16(1):87-90, 1958.

14] Hans J. Berliner.
"The B* Tree Search Algorithm: A Best-First Proof Procedure."
Artificial Intelligence, 12(1):23-40, 1979.

15] Hans J. Berliner and Carl Ebeling.
"Pattern Knowledge and Search: The SUPREM Architecture."
Artificial Intelligence, 38(2):161-198, 1989.

16] K. Bhaskaran and Michael Pinedo.
"Dispatching."
In G. Salvendy (ed.), *Handbook of Industrial Engineering*.
Wiley, New York, 1991.

17] Kurt M. Bischoff.
"Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C: Tutorial Introduction."
Department of Computer Science, Iowa State University, Ames, Iowa, 1994.
[ftp://ftp.cs.iastate.edu/pub/ox/*]

18] Kurt M. Bischoff.
"Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C: User Reference Manual."
Department of Computer Science, Iowa State University, Ames, Iowa, 1994.
[ftp://ftp.cs.iastate.edu/pub/ox/*]

19] J. T. Black.
The Design of the Factory With a Future.
McGraw-Hill, 1991.

20] J. H. Blackstone, D. T. Phillips and G. L. Hogg.
"A state-of-the-art survey of dispatching rules for manufacturing job shop operations."
International Journal of Production Research, 20:27-45, 1982.

21] Mark Boddy and Thomas Dean.
"Deliberation Scheduling for Problem Solving in Time-Constrained Environments."
Artificial Intelligence, 67(2):245-285, 1994.

22] Jean-Daniel Boissonnat and Mariette Yvinec.
Algorithmic Geometry.
Cambridge University Press, 1998.

23] Stuart Bowyer.
"The Extreme Ultraviolet Explorer Mission."
In T. Kondo (ed.), *Observatories in Earth Orbit and Beyond*, Kluwer Academic Publishers, 1990.

24] Stuart Bowyer.
"Extreme Ultraviolet Astronomy."
Scientific American, 271(2):32 ff., 1994.

25] Justin A. Boyan.
Learning Evaluation Functions for Global Optimization.
PhD Dissertation, Carnegie-Mellon University, 1998.

26] Gilles Brassard and Paul Bratley.
Algorithmics: Theory and Practice.
Prentice-Hall, Englewood Cliffs, NJ, 1988.

27] Jacques Carlier and E. Pinson.
"An algorithm for solving the job-shop problem."
Management Science, 35:164-176, 1989.

28] Center for Extreme Ultraviolet Astrophysics [CEA].
EUVE Guest Observer Handbook.
Appendix G of NASA NRA 92-OSSA-5, Berkeley, CA, January 1992.
[Other information is available through the World Wide Web, http://www.cea.berkeley.edu]

29] Peter Cheeseman, Bob Kanefsky and William M. Taylor.
"Where the Really Hard Problems Are."
In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, 1991.

30] Robert Clemen.
Making Hard Decisions. Second Edition.
Duxbury Press, Wadsworth Publishing, Belmont, CA, 1995.

31] Philippe Codognet and Daniel Diaz.
"Compiling Constraints in clp(FD)."
Journal of Logic Programming, 27(3):185-226, 1996.

32] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest.
Introduction to Algorithms.
McGraw Hill and MIT Press, 1990.

33] R. T. Cox.
"Probability, Frequency and Reasonable Expectation."
American Journal of Physics, vol. 14, 1946.

34] James M. Crawford and Larry D. Auton.
"Experimental Results on the Crossover Point in Satisfiability Problems."
In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC., 1993.

35] G. Dantzig.
"Linear Programming."
pp. 19-31 in Lenstra, Rinnooy Kan and Schrijver, eds. [79].

36] Martin Davis, George Logemann and Donald Loveland.
"A Machine Program for Theorem Proving."
Communications of the ACM, 5:394-397, 1962.

37] Martin Davis and Hilary Putnam.
"A Computing Procedure for Quantification Theory."
Journal of the ACM, 7:201-215, 1960.

38] Thomas Dean and Drew V. McDermott.
"Temporal Data Base Management."
Artificial Intelligence, 32(1): 1-55, 1987.

39] Rina Dechter, Itay Meiri and Judea Pearl.
"Temporal Constraint Networks."
Artificial Intelligence, 49(1-3):61-95.

40] Bruno de Finetti.
Theory of Probability.
Wiley, 1974.

41] Daniel Diaz.
clp(FD)
Software version 2.22, 1998.
[ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd/]

42] James R. Driscoll, Neil Sarnak, Daniel D. Sleator and Robert E. Tarjan.
"Making Data Structures Persistent."
In *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing* [STOC], 1986.

43] Dingzhu Du, Jun Gu and Panos M. Pardalos.
Satisfiability Problem: Theory and Applications.
DIMACS Series in Discrete Mathematics and Theoretical Computer Science: vol 35.
American Mathematical Society, Providence, RI, 1997.

44] O. Dubois, P. Andre, Y. Boufkhad and J Carlier.
"SAT versus UNSAT."
pp. 415-436 in Johnson and Trick [67].

45] Oren Etzioni and Daniel S. Weld.
"Intelligent Agents on the Internet: Fact, Fiction, and Forecast."
IEEE Expert, 10(4):44-49, 1995.

46] Mark S. Fox.
"ISIS: A Retrospective."
Chapter 1 in Zweben and Fox [138].

47] Mark S. Fox and Norman Sadeh.
"Why is Scheduling Difficult? A CSP Perspective."
In *Proceedings of the European Conference on Artificial Intelligence*, Stockholm, 1990.

48] Jon W. Freeman.
Improvements to Propositional Satisfiability Search Algorithms.
PhD Dissertation, University of Pennsylvania, 1995.

49] Jon W. Freeman.
POSIT: Propositional Satisfiability Testbed.
Software Version 1.0, 1994.
[ftp://ftp.cis.upenn.edu/pub/freeman]

50] Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W.H. Freeman, San Francisco, 1979.

51] Eliyahu M. Goldratt and Jeff Cox.
The Goal: A Process of Ongoing Improvement.
North River Press, Croton-on-Hudson, NY, 1986. Revised Edition.

52] Othar Hansson and Andrew Mayer.
"Probabilistic Heuristic Estimates."
Annals of Mathematics and Artificial Intelligence, 2:209-220, 1990.

53] Othar Hansson and Andrew Mayer.
"The Optimality of Satisficing Solutions."
In *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence*, Minneapolis, 1988.

54] Othar Hansson and Andrew Mayer.
"Heuristic Search as Evidential Reasoning."
In *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence*, Windsor, Ontario, 1989.

55] Othar Hansson and Andrew Mayer.
"A New and Improved Product Rule."
In *Proceedings of the Eighth Internatonal Congress of Cybernetics & Systems*, New York, 1990.

56] Othar Hansson and Andrew Mayer.
"DTS: A Decision-Theoretic Scheduler for Space Telescope Applications."
Chapter 13 in Zweben and Fox [138].

57] Othar Hansson and Andrew Mayer.
"DTS: A Toolkit for Building Custom, Intelligent Schedulers."
paper presented at i-SAIRAS conference, Pasadena, CA, October, 1994.

58] Othar Hansson, Andrew Mayer and Stuart J. Russell.
"Decision-Theoretic Planning in BPS."
In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, Stanford, CA, 1990.

59] Daishi Harada.
"Reinforcement Learning with Time."
In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, RI, 1997.

60] David Heckerman, John S. Breese and Koos Rommelse.
"Decision-Theoretic Troubleshooting."
Communications of the ACM, 38(3):49-57, 1995.

61] C. A. Holloway.
Decision Making under Uncertainty: Models and Choices.
Prentice-Hall, Englewood Cliffs, NJ, 1979.

62] Roger Scott Hoover.
Incremental Graph Evaluation.
PhD Dissertation, Cornell University, 1987.
[Available as CS Technical Report 87-836.]

63] Eric J. Horvitz.
Computation and Action under Bounded Resources.
PhD Thesis, Stanford University, 1990.
[Available as Knowledge Systems Laboratory Report KSL-90-76.]

64] E.J. Horvitz, J.S. Breese and M. Henrion.
"Decision Theory in Expert Systems and Artificial Intelligence."
Journal of Approximate Reasoning, Special Issue on Uncertainty in Artificial Intelligence, 2:247-302, 1988.

65] Ronald A. Howard.
"Information Value Theory."
IEEE Transactions on Systems, Man, and Cybernetics, vol. SSC-2, 1965.

66] Ronald A. Howard.
"Decision Analysis: Practice and Promise."
Management Science, 34:679-695.

67] David S. Johnson and Michael Trick (eds.).
Cliques, Coloring, and Satisfiability.
DIMACS Series in Discrete Mathematics and Theoretical Computer Science: vol 26.
American Mathematical Society, Providence, RI, 1996.

68] Laveen Kanal and Vipin Kumar (eds.).
Search in Artificial Intelligence.
Springer-Verlag, New York, 1988

69] Randy H. Katz.
Section 12.3 (Jump Counter) in *Contemporary Logic Design.*
Benjamin Cummings/Addison Wesley Publishing Company, 1993.

70] Ralph L. Keeney and Howard Raiffa.
Decisions with Multiple Objectives: Preferences and Value Tradeoffs.
Wiley, 1976.

71] J. E. Kelley, Jr.
"Critical Path Planning and Scheduling, Mathematical Basis."
Operations Research, 9(3):296-320, 1961.

72] J. E. Kelley, Jr.
"The Critical Path Method: Resources Planning and Scheduling."
Chapter 21 in Muth & Thompson [96].

73] William J. Kennedy, Jr. and James E. Gentile.
Statistical Computing.
Marcel Dekker, New York, 1980.

74] Donald E. Knuth.
"Semantics of Context-Free Languages."
Mathematical Systems Theory, 2(2):127-145, 1968.

75] Amy L. Lansky.
"Localized Planning with Diversified Plan Construction Methods."
Technical Report FIA-93-17, NASA Ames Research Center, Artificial Intelligence Research Branch, June 1993.

76] Eugene L. Lawler.
"Old Stories."
pp. 97-106 in Lenstra, Rinnooy Kan and Schrijver [79].

77] Eugene L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan and David B. Shmoys.
"Sequencing and Scheduling: Algorithms and Complexity."
Technical Report BS-R8909, Centre for Mathematics and Computer Science, Amsterdam, 1989.
[Also appeared as pp. 445-522 in S. C. Graves et al. (eds.), *Handbooks in OR & MS, Vol. 4*, Elsevier, 1993.]

78] Kai-Fu Lee.
Automatic Speech Recognition: The Development of the SPHINX System.
Kluwer Academic, Boston, 1989.

79] Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, Alexander Schrijver, editors.
History of Mathematical Programming: collection of personal reminiscences.
North-Holland, New York, 1991.

80] John R. Levine, Tony Mason and Doug Brown.
lex & yacc.
O'Reilly & Associates, Sebastopol, CA, 1992.

81] David Levy and Monty Newborn.
How Computers Play Chess.
Computer Science Press (W. H. Freeman), New York, 1991.

82] Kim Marriott and Peter J. Stuckey.
Programming with Constraints: An Introduction.
MIT Press, 1998.

83] Andrew Mayer.
Rational Search.
Ph.D. Dissertation, University of California, Berkeley, 1994.

84] Itay Meiri.
Temporal Reasoning: A Constraint-Based Approach.
Ph.D. Dissertation, University of California, Los Angeles, 1991.
[Available as Technical Report CSD-920019, Computer Science Department.]

85] Steven Minton, John Bresina and Mark Drummond.
"Commitment strategies in planning: A comparative analysis."
In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, 1991.

86] Steven Minton, Mark Johnston, Andrew Philips and Philip Laird.
"Solving Large-Scale Constraint-Satisfaction and Scheduling Problems using a Heuristic Repair Method."
In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, 1990.

87] David Mitchell, Bart Selman and Hector Levesque.
"Hard and easy distributions of SAT problems."
In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, 1992.

88] L. G. Mitten.
"An Analytic Solution to the Least Cost Testing Sequence Problem."
Journal of Industrial Engineering, vol. 11, 1960.

89] U. Montanari.
"Networks of Constraints: Fundamental Properties and Applications to Picture Processing."
Information Processing Letters, vol. 7, 1974.

90] Thomas E. Morton and David W. Pentico.
Heuristic scheduling systems : with applications to production systems and project management.
Wiley, 1993.

91] Nicola Muscettola, Stephen F. Smith.
"A Probabilistic Framework for Resource-Constrained Multi-Agent Planning."
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, 1987.

92] Nicola Muscettola and Stephen F. Smith.
"Integrating Planning and Scheduling to Solve Space Mission Scheduling Problems."
In *Proceedings of the DARPA Planning Workshop*.
Morgan Kaufmann, San Mateo, CA, 1990.

93] Nicola Muscettola.
"HSTS: Integrating Planning and Scheduling."
Chapter 6 in Zweben and Fox [138].

94] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Brian C. Williams.
"Remote Agent: to boldly go where no AI system has gone before."
Artificial Intelligence, 103:5-47, 1998.

95] Nicola Muscettola, Barney Pell, Othar Hansson and Sunil Mohan.
"Automating mission scheduling for space-based observatories."
In G.W. Henry and J.A. Eaton (eds), *Robotic Telescopes: Current Capabilities, Present Developments, and Future Prospects for Automated Astronomy*. ASP Conference Series, Vol. 79.
Astronomical Society of the Pacific, San Francisco, 1995.

96] John F. Muth and Gerald L. Thompson, editors.
Industrial Scheduling.
Prentice-Hall, Englewood Cliffs, NJ, 1963.

97] Dana S. Nau, Vipin Kumar and Laveen Kanal.
"General Branch and Bound, and its Relation to A* and AO*."
Artificial Intelligence, 23(1): 29-58, 1984.

98] Monroe Newborn.
Kasparov Versus Deep Blue : Computer Chess Comes of Age.
Springer Verlag, 1996.

99] Andrew J. Palay.
Searching With Probabilities.
Morgan Kaufmann, San Mateo, CA, 1985.

100] Judea Pearl.
Heuristics: Intelligent Search Strategies for Computer Problem-Solving.
Addison-Wesley, Reading, MA, 1984.

101] Judea Pearl.
Probabilistic Reasoning in Intelligent Systems.
Morgan Kaufmann, San Mateo, CA, 1988.

102] Judea Pearl and Glenn Shafer.
Readings in Uncertain Reasoning.
Morgan Kaufmann, San Mateo, CA, 1988.

103] Joseph C. Pemberton and Flavius Galiber, III.
"A Constraint-Based Approach to Satellite Scheduling."
Paper presented at the DIMACS Workshop on Constraints and Large Scale Discrete Optimization, Rutgers University, Piscataway, NJ, September 1998.
[http://www.greas.com/papers/]

104] David Poole, Alan Mackworth and Randy Goebel.
Computational Intelligence: A Logical Approach.
Oxford University Press, 1998.

105] Franco Preparata and Michael I. Shamos.
Computational Geometry.
Springer-Verlag, New York, 1985.

106] James S. Press.
Bayesian Statistics : Principles, Models, and Applications.
Wiley, 1989.

107] Howard Raiffa.
In *The Harvard Guide to Influential Books*, C. Maury Devine (ed.), Harper, 1986.

108] Howard Raiffa.
Decision Analysis.
McGraw Hill, 1997.

109] Frank P. Ramsey.
The Foundations of Mathematics and other Logical Essays.
Littlefield Adams, London, 1960.

110] Thomas W. Reps.
Generating language-based environments.
MIT Press, 1984.

111] Thomas W. Reps and Tim Teitelbaum.
The Synthesizer Generator: a System for Constructing Language-Based Editors.
Springer-Verlag, 1989.

112] Thomas W. Reps and Tim Teitelbaum.
The Synthesizer Generator Reference Manual.
Springer-Verlag, 1989, 3rd edition.

113] Stuart J. Russell.
“Rationality and Intelligence.”
Artificial Intelligence, 94(1-2): 57-77, 1997.

114] Stuart Russell and Devika Subramanian.
“Provably bounded-optimal agents.”
Journal of Artificial Intelligence Research, 2, 1995.
[www.jair.org]

115] Stuart J. Russell and Eric Wefald.
“Principles of Metareasoning.”
Artificial Intelligence, 49(1-3): 361-395, 1991.

116] Stuart J. Russell and Eric Wefald.
Do the Right Thing: Studies in Limited Rationality.
MIT Press, Cambridge, MA, 1991.

117] Norman Sadeh.
Lookahead Techniques for Micro-Opportunistic Job-Shop Scheduling.
Ph.D. Dissertation, Carnegie Mellon University, 1991.

118] L. J. Savage.
The Foundations of Statistics.
Dover, New York, 1972.

119] Micha Sharir and Pankaj K. Agarwal.
Davenport-Schinzel Sequences and Their Geometric Applications.
Cambridge University Press, 1995.

120] Yoav Shoham.
"Agent-Oriented Programming."
Artificial Intelligence, 60(1): 51-92, 1993.

121] Herbert A. Simon and Joseph B. Kadane.
"Optimal Problem-Solving Search: All-or-None Solutions."
Artificial Intelligence, vol. 6, 1975.

122] Douglas R. Smith and Eduardo A. Parra.
"Transformational Approach to Transportation Scheduling."
In *Proceedings KBSE '93: The Eighth Knowledge-Based Software Engineering Conference*.
IEEE Computer Society Press, 1993.

123] Gerald Tesauro.
"Temporal Difference Learning and TD-Gammon."
Communications of the ACM, 38(3): 58-68, 1995.

124] Ronald A. Thisted.
Elements of Statistical Computing.
Chapman & Hall, New York, 1988.

125] Edward Tsang.
Foundations of Constraint Satisfaction.
Academic Press, New York, 1993.

126] Pascal van Hentenryck.
Constraint-Satisfaction in Logic Programming.
MIT Press, Cambridge, MA, 1989.

127] John von Neumann and Oskar Morgenstern.
Theory of Games and Economic Behavior.
Princeton University Press, 1944.

128] Detlof von Winterfeldt and Ward Edwards.
Decision Analysis and Behavioral Research.
Cambridge University Press, 1986.

129] William M. Waite.
Compiler Construction.
Springer Verlag, NY, 1984.

130] Abraham Wald.
Statistical Decision Functions.
Wiley, 1950.

131] David Waltz.
"Understanding line drawings of scenes with shadows."
In P. H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, 1975.

132] Stephen R. Watson and Dennis M. Buede.
Decision synthesis : the principles and practice of decision analysis.
Cambridge University Press, 1987.

133] H. P. Williams.
Model Solving in Mathematical Programming.
John Wiley & Sons, 1993.

134] Wayne L. Winston.
Introduction to Mathematical Programming: Applications and Algorithms.
PWS-Kent Publishing Co., Boston, 1991.

135] James P. Womack, Daniel T. Jones and Daniel Roos.
The Machine that Changed the World: How Japan's Secret Weapon in the Global Auto Wars will Revolutionize Western Industry.
Harper Perennial, New York, 1990.

136] Shlomo Zilberstein.
Operational Rationality through Compilation of Anytime Algorithms.
Phd Dissertation, University of California, Berkeley, 1993.

137] Monte Zweben, Brian Daun, Eugene Davis, and Michael Deale.
"Scheduling and Rescheduling with Iterative Repair."
Chapter 8 in Zweben and Fox [138].

138] Monte Zweben and Mark S. Fox (eds.).
Intelligent Scheduling.
Morgan Kaufmann, San Francisco, 1994.